In Christoph Meinel and Larry Leifer (eds.).
Design Thinking Research: Achieving Real Innovation (pages 111-145)
Springer 2022 (doi.org/10.1007/978-3-031-09297-8_7)

A Pattern Language of an Exploratory Programming Workspace

Marcel Taeumel and Jens Lincke and Patrick Rein and Robert Hirschfeld

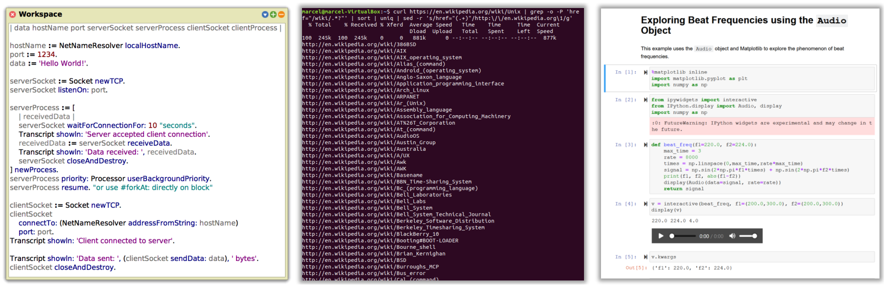
Abstract Software design and the underlying programming activities entail a great portion of exploration to better understand problem and solution spaces. There are programming tools and environments that support such exploratory programming practices exceptionally well. However, inexperienced programmers typically face a steep learning curve until they can reach the promised efficiency in such tools. They need a long time to study best practices firsthand in real projects. The tools in use might also need adjustments, given that modern programming languages are continually introducing new features or redesigning old ones. We want to apply the idea of *patterns* to capture traditional and modern practices of exploratory programming. In this chapter, we focus on the *workspace* tool, whose core ideas transcend many different programming communities such as the Smalltalk workspace, the Unix shell, and data-analysis notebooks. We extracted the essence into a novel *pattern language* around the *conversations* that programmers have with their environment. We believe that our work can help programmers to quickly understand and apply the idea of workspaces, as well as tool builders to increase the efficiency of their project team when facing exploratory challenges.

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

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



```
| data hostName port serverSocket serverProcess clientSocket clientProcess |

hostName := NetNameResolver localHostName.
port := 1234.
data := 'Hello World!'.

serverSocket := Socket newTCP.
serverSocket listenOn: port.

serverProcess := [
    | receivedData |
    serverSocket waitForConnectionFor: 10 "seconds".
    Transcript showln: 'Server accepted client connection'.
    receivedData := serverSocket receiveData.
    Transcript showln: 'Data received: ', receivedData.
    serverSocket closeAndDestroy.
] newProcess.
serverProcess priority: Processor userBackgroundPriority.
serverProcess resume. "or use #forkAt: directly on block"

clientSocket := Socket newTCP.
clientSocket
    connectTo: (NetNameResolver addressFromString: hostName)
    port: port.
Transcript showln: 'Client connected to server'.

Transcript showln: 'Data sent: ', (clientSocket sendData: data), ' bytes'.
clientSocket closeAndDestroy.
```

```
marcel@marcel-VirtualBox:~$ curl https://en.wikipedia.org/wiki/Unix | grep -o -P 'hre
f="/wiki/.*?"' | sort | uniq | sed -r 's/href="(.+)"/http:\/\/en.wikipedia.org\1/g'
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100  245k  100  245k    0     0   881k      0 --:--:-- --:--:-- --:--:--  877k
http://en.wikipedia.org/wiki/386BSD
http://en.wikipedia.org/wiki/AIX
http://en.wikipedia.org/wiki/AIX_operating_system
http://en.wikipedia.org/wiki/Alias_(command)
http://en.wikipedia.org/wiki/Android_(operating_system)
http://en.wikipedia.org/wiki/Anglo-Saxon_language
http://en.wikipedia.org/wiki/Application_programming_interface
http://en.wikipedia.org/wiki/Arch_Linux
http://en.wikipedia.org/wiki/ARPANET
http://en.wikipedia.org/wiki/Ar_(Unix)
http://en.wikipedia.org/wiki/Assembly_language
http://en.wikipedia.org/wiki/Association_for_Computing_Machinery
http://en.wikipedia.org/wiki/AT%26T_Corporation
http://en.wikipedia.org/wiki/At_(command)
http://en.wikipedia.org/wiki/AudioOS
http://en.wikipedia.org/wiki/Austin_Group
http://en.wikipedia.org/wiki/Australia
http://en.wikipedia.org/wiki/A/UX
http://en.wikipedia.org/wiki/Awk
http://en.wikipedia.org/wiki/AWK
http://en.wikipedia.org/wiki/Basename
http://en.wikipedia.org/wiki/BBN_Time-Sharing_System
http://en.wikipedia.org/wiki/Bc_(programming_language)
http://en.wikipedia.org/wiki/Bell_Laboratories
http://en.wikipedia.org/wiki/Bell_Labs
http://en.wikipedia.org/wiki/Bell_System
http://en.wikipedia.org/wiki/Bell_System_Technical_Journal
http://en.wikipedia.org/wiki/Berkeley_Software_Distribution
http://en.wikipedia.org/wiki/Berkeley_Timesharing_System
http://en.wikipedia.org/wiki/BlackBerry_10
http://en.wikipedia.org/wiki/Booting#BOOT-LOADER
http://en.wikipedia.org/wiki/Bourne_shell
http://en.wikipedia.org/wiki/Brian_Kernighan
http://en.wikipedia.org/wiki/BSD
http://en.wikipedia.org/wiki/Burroughs_MCP
http://en.wikipedia.org/wiki/Bus_error
http://en.wikipedia.org/wiki/Cal_(command)
```

Exploring Beat Frequencies using the `Audio` Object

This example uses the `Audio` object and Matplotlib to explore the phenomenon of beat frequencies.

```
In [1]:  %matplotlib inline
         import matplotlib.pyplot as plt
         import numpy as np

In [2]:  from ipywidgets import interactive
         from IPython.display import Audio, display
         import numpy as np

         :0: FutureWarning: IPython widgets are experimental and may change in t
         he future.

In [3]:  def beat_freq(f1=220.0, f2=224.0):
             max_time = 3
             rate = 8000
             times = np.linspace(0,max_time,rate*max_time)
             signal = np.sin(2*np.pi*f1*times) + np.sin(2*np.pi*f2*times)
             print(f1, f2, abs(f1-f2))
             display(Audio(data=signal, rate=rate))
             return signal

In [4]:  v = interactive(beat_freq, f1=(200.0,300.0), f2=(200.0,300.0))
         display(v)

         220.0 224.0 4.0

In [5]:  v.kwargs

Out[5]:  {'f1': 220.0, 'f2': 224.0}
```

Fig. 1 Workspace-like tools support effective and efficient conversations between programmers and their environments. They can be found in many different domains such as (f.l.t.r.) Squeak/Smalltalk (workspace), Ubuntu/Linux (shell), and Jupyter/IPython (notebooks).

1 Introduction

The exploratory mindset is an intensive form of user-to-software mediation where programmers are especially motivated to find a design that both works and inspires: "I know it when I see it." Programming languages are expressive enough to unfold anything from unsurprising complexity to surprisingly beautiful simplicity. The latter is key.

Programming tools play a crucial part in the exploratory journey to reach high-quality software. All programming activities have complementary tool support: reading, writing, debugging, testing, deploying, and so on. Programmers can *thrive* within environments that offer a sense of liveness [26, 20] and short feedback cycles through tools. Mistakes can be corrected on short notice – with the next iteration likely more productive than the previous one. While an exploratory mindset expects certain things from tools and their user interfaces, good tools for exploration can even foster such an exploratory mindset. As illustrated in Figure 1, exploratory tools have been around for a long time such as the Smalltalk workspace [6], the Unix shell [19], and notebooks for scientific computing [18].

However, it is not easy for inexperienced programmers to learn about and grow an exploratory mindset. Even if the appropriate tools are at hand, it can be challenging to use them effectively and efficiently. Many best practices are supported (or even promoted) through tools but not easily discovered during normal use. Without proper guidance, you may never leave the traditional path, that is, remaining pleased with the habits you acquired by chance, completing your tasks in a "good enough" fashion. Provided that you are a tool builder in a *foreign* environment, it can be challenging to pinpoint specific issues but rather to criticize *everything* that does not feel "like home." For example, if you prefer the programming experience in a Smalltalk environment, you might be tempted to re-create its entirety for your JavaScript workflow. While there is the Lively system for modern web development [12, 14] in the spirit of Smalltalk, it took a great effort and many resources to reach its productivity level as of today. Instead of specific tool implementations, we are looking for a way to capture the essence of exploratory programming practices – covering both mindset and tools – and that is easily accessible for learners and practitioners – for tool users and tool builders.

We apply the idea of *patterns* to capture the core of best practices for exploratory programming. Originally promoted for architectural design [2], patterns have proven useful for software design [5] as well as for guiding human behavior [9]. We like the *generative* aspects of the pattern form, which allows for variation while clearly separating the repetitive elements. For example, there is clearly an overlap between Smalltalk workspaces and Unix shells, but it is not easy to see for a non-expert. The patterns in this chapter build upon our previous work. Our experience with tools for exploration stems from the Squeak/Smalltalk programming system [11], which we have been using in teaching and research for more than a decade. Our first take on patterns approached the concepts of enabling and controlling exploration [25]. Our second take presented actions around direct window manipulation [24], which we collected during our work on the Vivide tool-building platform [22].

In this chapter, we contribute a pattern language around the conversation style that happens between programmers and their environments using workspace-like tools: the exploratory programming workspace. The central pattern “Conversation in Context” describes the typical question/response cycle that programmers follow to learn more during their programming task such as the location of a bug or the best way to implement a new feature. The three main components of that interaction lead to the other patterns: “Elaborate Inquiry” and “Coach Your Environment” dive into questions, “Concept in Shards” and “Proxy Transport” talk about the context, “Simple Response” and “Tangible Response” address responses. The last pattern “Pause and Explore” embeds a single conversation into the bigger picture, because programmers must often manage multiple conversations at the same time.

In section 2, we describe the pattern form and agents we use in each pattern. We also give an overview of all pattern’s intents. In section 3 to section 10, we present the eight patterns in this pattern language. Finally in section 11, we conclude this chapter with a discussion on pattern quality and possible next steps.

2 Pattern Form, Intents, and Agents

In this section, we describe the elements (or form) of each pattern, give an overview of all intents in our pattern language, and elaborate on the agents whose characteristics and actions we try to capture in each pattern. It is all about *you* working on a *project* in an *environment* while continually switching between different *interaction contexts*.

2.1 The Pattern Form

Building upon the original triad of context-problem-solution by Christopher Alexander [3], we took inspiration from both patterns of object-oriented design [5] and human behavior [10] to shape the pattern form for this chapter:

- **Intent** is a quick summary of the actions that are typically part of the pattern's solution in the manner of "Do this, then that."
- **A Desire for Exploration** motivates the pattern through a hands-on story. We typically refer to our experiences from the Squeak/Smalltalk system [11].
- **The Profile** is a brief summary of the pattern's context, problem, and solution. As in patterns on human actions [10], it includes a list of forces that the pattern tries to resolve.
- **The Structure** is a more detailed listing of what the programmer and its tools (or environment) must achieve to resolve the conflict (i.e., the forces). Less technical than object-oriented design patterns [5], it, however, includes an abstract visual model as in Iba's patterns [10].
- **Consequences** outlines possible trade-offs that tool builders face when implementing tools for the particular pattern, which might eventually also affect programmers as tool users.
- **Related Patterns** are the backbone in our pattern language. Without meaningful connections between our patterns, we would have a list of unrelated scenarios. Yet, our patterns all deal with the conversational style between programmers and their workspace-like tools.

Note that we will use the term 'context' differently throughout this chapter. We will not address a 'problem context' but an 'interaction context', which is the knowledge base full of technical artifacts leading toward the solution. See below.

2.2 Overview of All Patterns

Our pattern language includes eight patterns. The first and the last pattern frame the middle ones, which are organized in pairs dealing with questions, context, and responses, respectively. Each pattern's intent is as follows:

Conversation in Context *(section 3)* Ask the environment one or more questions to better understand the context you are currently working with. Iteratively revise each question until you are satisfied with the environment's response. In follow-up questions, refer to prior responses to make your wording (and the responses) more expressive and precise.

Elaborate Inquiry *(questions, section 4)* Decompose the topic of your conversation into parts. Then compose your central question from multiple sub-questions according to those parts. Let the environment manage the topic's complexity to reduce your cognitive load and hence the risk of making mistakes.

Coach Your Environment *(questions, section 5)* Extend the environment's knowledge by adding information to the current interaction context. Extend its language vocabulary by adding tools for reference in follow-up questions. Both kinds of additions can either come from your memory (and experience) or be loaded from external resources.

Concept in Shards *(context, section 6)* Relevant information might be scattered all over the environment. Do not be satisfied with the first useful response. Keep asking questions to reduce guesswork and to gain more substantial insight. If necessary, import external knowledge from outside the environment.

Proxy Transport *(context, section 7)* Design a flexible proxy to integrate external information into your interaction context. Map the external format (e.g., file contents) onto the internal one (e.g., object fields). Use *lazy loading* and *caching* to avoid high memory consumption and slow access times.

Simple Response *(responses, section 8)* Every question has a simple response. Do not hesitate but ask away; embrace trial-and-error. A text will appear, a picture will render, a sound will play – tackle complexity with a combination of these.

Tangible Response *(responses, section 9)* Directly engage (e.g., click on) a response as it is represented on screen. The environment will respond to the "What is this?" question with more details. Return to the particular conversation later on or spawn new ones.

Pause and Explore *(section 10)* Suspend an ongoing conversation if you need to explore and understand details of a prior response. Start a new conversation to learn more about these details. Later, resume the original response and make use of your freshly gained knowledge to pose better questions.

2.3 You and Your Project

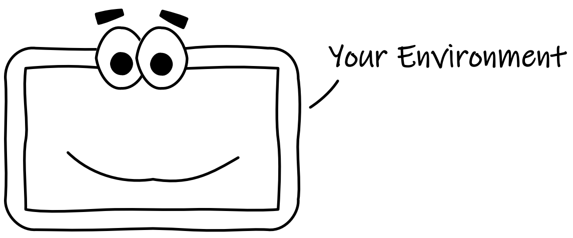
We are aware that readers of this chapter might not necessarily identify themselves as *programmers*. Even though non-professional, end-user programming is widespread [13, 4], we think that exploratory programming affects devoted programmers more than hobbyists. The exploratory mindset covers more than just "getting things done;" it entails an *intensive* relationship with the computational medium. Therefore, each pattern's "You" addresses the reader as if they were programmers who really want to dive into source code to make their software project not just working but simple and inspiring.

The exploratory programmer – meaning *you* – might be working on a multitude of programming tasks where conversations – and thus our pattern language – can provide guidance. Besides hunting down bugs or designing new features, programmers also add commentary to source code or write tests for untested modules. They might also teach new team members about existing system components directly through the live system, which is more credible than talking about (dead) documentation. In all these cases, programmers engage with the programming environment and its tools to investigate and learn.

Tool builders are programmers, too. While your project's goal is mainly about new software for your customers, there is other software to be built or adapted on the side: the programming tools. In our pattern language, you will notice requirements for specific services that the programming environment has to provide. It is the job of tool builders to ensure that there are tools in the environment that comply. If not, tool-using programmers cannot follow the particular pattern's guidance. So, when a problem force claims that "It is easy to ..." or the pattern structure dictates that "The environment should ...", tool builders have to take action. Note that every programmer can take on the role of a builder, even ad-hoc during a task.

2.4 The Environment

This is your conversational partner on your exploratory journey:



The environment is an *active entity* because it offers various ways of interaction. As a programmer, it helps you author program source code. As an explorer, it helps you find and connect relevant information. It is your *tool shed* because it offers many services through interactive tools. You can use keyboard, mouse, or touch input to express your goals. It will acknowledge your input and try to fulfill your expectations through text, graphics, or sound.

The environment has one or more languages, which you must use during conversations. A language is not necessarily a *programming language* but sometimes just a graphical interface with clickable, labeled buttons. Then, you must click the right buttons to express your goals. In general, a language's vocabulary is *extensible* so that you can teach new phrases for a more efficient communication.

There are other environments, each with its own boundaries. For one thing, your environment might be *embedded* within a (parent) environment. Then again, that parent might embed more environments besides yours, which makes them siblings. Practical examples include operating systems, which host applications that may complement each others use cases. Being aware of other environments helps you overcome limitations in your knowledge and that of your environment. You can always learn from things outside your own boundaries.

2.5 The Interaction Context

The environment has a knowledge base, which you will constantly probe and expand during conversations: the interaction context. The context is the *passive* counterpart to the *active* environment. It is separate from the language vocabulary and contains all kinds of domain-specific information. Within the environment's boundaries, information is represented in a common (transport) format. In a Smalltalk system, everything is an object. In a Unix-based system, everything is a file. Thus, Smalltalk workspaces talk about objects (and structured fields) while Unix shells talk about files (and byte streams).

Accessing information from the interaction context is not as flexible as using your brain to think of something. The context is typically *scoped* to a particular tool. That is, the information from one conversation is not necessarily accessible in another one. Yet, there are means of combining such *local scopes* or even promoting them to being *globally available* in the entire environment. Note that you can typically refer to pieces of information in a (scoped) context *by name*. Examples include the "environment" in a Unix shell or the "bindings dictionary" in a Smalltalk workspace. To sum it up, your environment might have *all the knowledge*, but you have to find an *appropriate context* to access the information you are looking for.

3 Conversation in Context

Ask the environment one or more questions to better understand the context you are currently working with. Iteratively revise each question until you are satisfied with the environment's response. In follow-up questions, refer to prior responses to make your wording (and the responses) more expressive and precise.

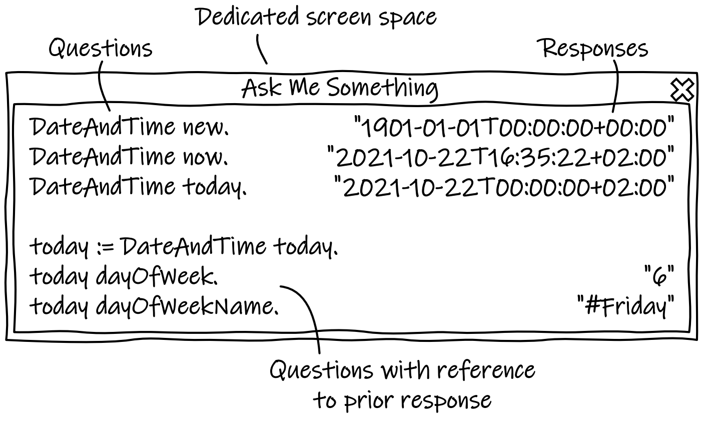
3.1 A Desire for Exploration

In exploratory programming, conversations with the computational medium are paramount. The environment represents a gateway to all digitally available information, which includes all kinds of software-related artifacts. Starting with a simple question, the environment can help you learn more about your current opportunities within your current task context. Even if your overall knowledge feels satisfactory, exploring responses to concrete questions can reduce the risks of relying on false hypotheses. If you are lucky, responses are provided without noticeable delay so that you can follow up and establish a conversation that feels live and direct. At the end of such an interaction, you will know more than before, and the environment might also have learned something about your context-specific inquiries.

So, instead of asking yourself, you ask the environment. Provided that you are working within a task-specific context, you might want to ask something about a module's interface or explore the structure of an artifact: Does the property X hold in this context? How many kinds of X objects are there? What are the fields of X? How does the service X work? Is this module X still functional? Responses can range from simple Booleans, numbers, or texts to deeply structured information.

Note that the actual representations of questions and responses are not captured in this pattern. We assume that you want to engage with the environment in a direct conversation to help you move forward in your programming task. There can be many reasons for why you would want to keep up a conversation, such as unclear wording in your prior question or confusing details in a prior response. Representations might be textual, visual, or even audible, which depends on your human-computer interface.

Imagine that you read about the notion of `DateAndTime` in a piece of documentation. Now you want to learn more about it. By example, you reason like this: "Today is Friday. The environment should know that. Let's ask it." Visually, the conversation might have happened as follows:



```
DateAndTime new.          "1901-01-01T00:00:00+00:00"
DateAndTime now.          "2021-10-22T16:35:22+02:00"
DateAndTime today.        "2021-10-22T00:00:00+02:00"

today := DateAndTime today.
today dayOfWeek.                                  "6"
today dayOfWeekName.                        "#Friday"
```

First, you found a space on screen where the environment could listen to your questions. Second, you posed several questions using the (programming) language that the environment could understand. In this case, simple text-based representations of both questions and responses drove the conversation successfully. Finally, you learned more about the interface of `DateAndTime`, which is knowledge that you can now apply in your task (and program).

3.2 The Profile

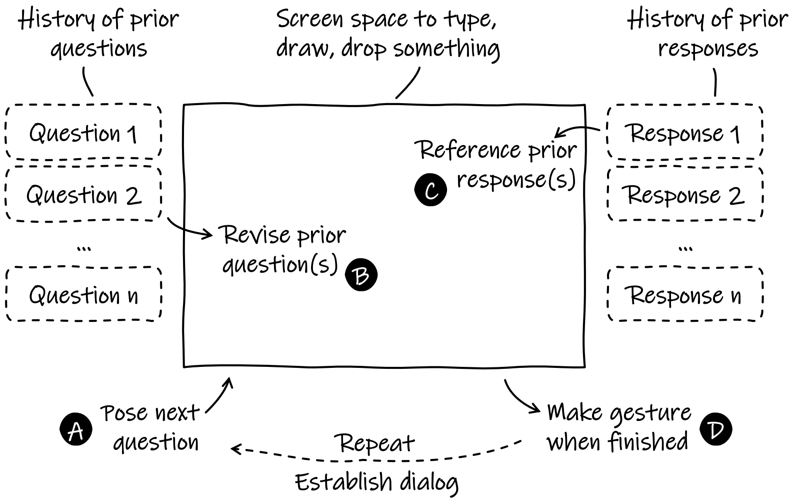
You want to ask the environment a *question* about something you have on your mind, that is, within a *context*. You also know about the (textual or visual) language that the environment can understand, that is, evaluate to present a notable response.

However, you cannot directly express your thoughts even though you have the feeling that the environment has all the information it needs to answer your questions, to help you move forward.

- It is easy to type, draw, or drop something using your input devices (e.g., keyboard, mouse, or touch screen).
- It is difficult to find a space where (1) you can focus on phrasing the question and (2) the environment will (try to) respond.
- It is difficult to find the right words for a question the first time.
- The environment's responses can be too technical or verbose, thus not comprehensible immediately.

Therefore, you engage with the environment and establish a conversation to learn more about the current context. First, you locate a space where the environment waits for your direct input. Then you type, draw, or drop something that complies with the environment's language. After posing your question, you make a gesture to let the environment evaluate your words and present a response. After your initial interaction, you can establish a conversation through (revised) follow-up questions to clarify misunderstandings or unpack complexity. Consequently, you can rely on the environment to maintain a history of prior questions and responses. On the one hand, you can easily revise your prior wording. On the other hand, you can precisely refer to selected pieces of information you just acquired. Your goal is to continue this conversation until you fully understand the environment's response(s).

3.3 The Structure


History of prior questions
Screen space to type, draw, drop something
History of prior responses
Question 1
Question 2
...
Question n
Reference prior response(s) C
Revise prior question(s) B
Response 1
Response 2
...
Response n
A Pose next question
Repeat
Establish dialog
Make gesture when finished D

The environment should provide the following services:

- **Screen space** that is dedicated to direct user input (i.e., the questions) and the presentation of results (i.e., the responses).
- **Context** that is accessible through the environment's known language (and vocabulary) such as identifiers (or names) of relevant software artifacts.

- **History of prior questions** that is automatically written with every new question posed. The history's contents are directly accessible for the user in a way that allows for effortless revisions.
- **History of prior responses** that is automatically written with every new response delivered. The history's contents are directly accessible for the user in a way that allows for references selected pieces in follow-up questions.

You should adhere to the following interaction (loop):

A) Pose a new question by typing, drawing, or dropping something into the dedicated space that the environment can understand.
B) Optionally, access one of the prior questions to repeat or revise its wording.
C) Optionally, access one of the prior responses to integrate references to its contents into your next question.
D) Finish your input with a dedicated gesture, let the environment evaluate your question and present a response, and then explore that response.

3.4 Consequences

Direct conversations with the environment reduce the need for taking offline notes. Both histories of questions and responses document your thoughts and can thus help you remember them later in the process. Plus, the notes you would take outside the environment would be detached from what actually goes on in your project, challenging to synchronize. Being immersed within your environment, instead, you can easily build up trust in your knowledge and your decisions.

A conversation can only be as effective as its representations for questions and responses. For example, if you cannot express your thoughts in text, you might wish to draw a shape instead. Yet, the environment's language might not support a visual syntax. At worst, inappropriate representations can be misleading and thus time-consuming.

The environment's response might be "rich" and interactive in the sense that it promotes you to a different tool (window) and experience. In a similar fashion, your questions might also be formed through an elaborate interaction using keyboard, mouse, or touch. That is, even though there are many text-based command-line interfaces following the "Conversation in Context" pattern, you might also be able to establish conversations through a graphical tool's interactive widgets.

The environment benefits from an efficient *window manager* to help you organize your questions and responses on the screen. Screen space is limited, and you might struggle finding a "quiet place" where you can focus on your next conversation. Multiple other conversations might also be in progress at the same time. The integration of multiple tools (and windows) might become an issue. Otherwise, you have to repeat your query in every new conversation, risking mistakes on the way.

The environment should respond as fast as possible; it should create a feeling of *liveness* during conversations. Otherwise, there is a chance of you losing interest in engaging with the environment in the first place. To avoid unnecessary workload in the environment, you finish your question with an extra gesture. Yet, unforeseen complications might increase the time before you can receive and read a response. Maybe you even want a *pause button* to be able to further explore the environment's current progress to then cancel and revise your question.

The history of prior questions and responses might consume many resources, thus possibly interfering with your other tasks. You might not be able to decide when to discard older information. You might not even know about the actual footprint of a question or response. So, the environment should probably figure out a way to discard deprecated information and compress redundant details to save resources. It might be able to inform you about the most demanding artifacts to help you make critical decisions.

3.5 Related Patterns

This pattern is the umbrella for all other patterns in this chapter:

- The *questions* in a conversation are covered in "Elaborate Inquiry" and "Coach Your Environment."
- The *context* that the environment needs to evaluate each question is covered in "Concept in Shards" and "Proxy Transport."
- The *responses* that the environment produces is covered in "Simple Response" and "Tangible Response."

The final pattern "Pause and Explore" complements the liveness and flow (that we strive for in each conversation) with the importance to manage conversations at a higher level and be able to reflect on every little detail.

4 Elaborate Inquiry

Decompose the topic of your conversation into parts. Then compose your central question from multiple sub-questions according to those parts. Let the environment manage the topic's complexity to reduce your cognitive load and hence the risk of making mistakes.

4.1 A Desire for Exploration

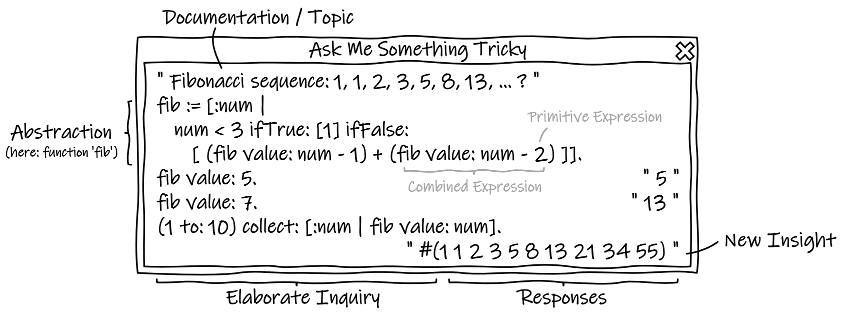
Simple questions are good conversation starters. You learn about the environment's vocabulary and can easily browse the current interaction context. Soon you begin to wonder whether it is possible to combine selected pieces of information, that is, prior questions or responses. There are some operations that you could do in your head or on a piece of paper such as picking the longest entry from a longer list, sorting a shorter list alphabetically, or removing all odd numbers from a list. However, all of these operations are tedious and error-prone, especially if done repeatedly and manually. And your exploratory journey has just begun. Many interesting artifacts might lie ahead, waiting to be explored. Computers are generally good at sorting, filtering, and transforming millions of data points repeatedly and efficiently. You just have to apply the environment's means of combination and abstraction.

We assume that the environment's language is either a general-purpose or domain-specific programming language that allows for combining simple expressions. Such a language does not have to be textual but usually is. Provided that you are working in a Unix shell, you can easily combine small (filter) programs through pipes:

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

This program fetches a website, extracts URLs from its contents in a sorted and unique fashion, and finally writes everything into the `urls.txt` file. You can now use this "Elaborate Inquiry" to continue exploration at a higher level, that is, for extracting URLs from arbitrary websites.

A language's means of abstraction can help you wrap complex questions (i.e., the combinations) into modules (e.g., functions) which you can then refer to *by name* in follow-up questions. For example, think about the beginning of the Fibonacci sequence: 1, 1, 2, 3. You want to know the tenth number but not calculate it by hand. So, you begin a conversation and describe the underlying rule of the sequence, which is "Add the two prior numbers to get the next one." Visually, the conversation might have happened as follows:



```
" Fibonacci sequence: 1, 1, 2, 3, 5, 8, 13, ... ? "
fib := [:num |
  num < 3 ifTrue: [1] ifFalse:
    [ (fib value: num - 1) + (fib value: num - 2) ]].
fib value: 5.                                          " 5 "
fib value: 7.                                         " 13 "
(1 to: 10) collect: [:num | fib value: num].
                            " #(1 1 2 3 5 8 13 21 34 55) "
```

First, you wrote a little reminder about what you want to discuss here. Then you designed the underlying rule as a recursive function, which you labeled "fib." You tried out the function multiple times until you got it right. Finally, you collected the sequence up to ten and found your definitive answer: 55. Given that function, you asked for the 100th number, but after waiting for a while you decided to cancel the computation without getting a response.

4.2 The Profile

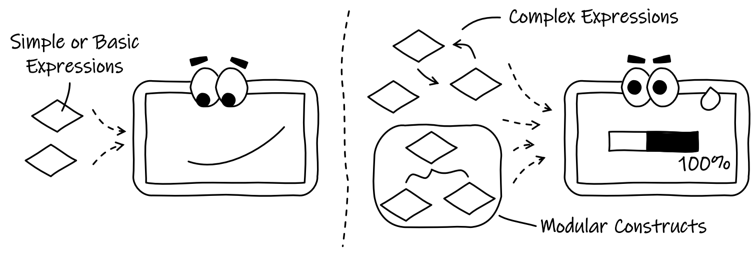
You are in the middle of a "Conversation in Context." It turns out that the topic you are interested in has several aspects with rather complex relationships. You notice a substantial risk of making mistakes, relying on guesswork.

However, you struggle with comprehending the environment's responses because their complexity outranks your questions' simplicity. Follow-up questions only increase your cognitive load because you start to combine facts in your head rather than letting the environment combine the facts for you.

- It is easy to ask follow-up questions that refer to prior responses.
- Such follow-up questions benefit from a good understanding of prior responses.
- It is difficult to *manually* interpret *complex* responses.
- Mistakes in understanding can infect follow-up questions.
- It is difficult to express your *complex* thoughts using *primitive* vocabulary.

Therefore, you adhere to a *modular* approach when (com-)posing your (follow-up) questions. At best, the final revision of your central question will satisfy your entire informational need. First, you decompose the topic of your conversation into – more or less isolated – parts. Second, you compose your question from sub-questions, each addressing one important part. The combination of sub-questions will naturally process the environment's intermediate responses. Consequently, the *manual*, error-prone act of posing multiple follow-up question will become an *automated*, robust one. The environment will immediately tell you when you make a mistake. Eventually, you will feel like an author who can safely prepare the script for a play, rather than a person who is trapped in the middle of a heated discussion.

4.3 The Structure



Considering a common trichotomy for programming language design [1, pp. 4–31], the environment's language should have the following properties:

- **Primitive expressions** that allow for posing simple questions and a stepwise unpacking of complex responses.
- **Means of combination** that allow for composing multiple (sub-)questions to automatically integrate intermediate responses.

- **Means of abstraction** that allow for encapsulating recurrent parts of the conversation to further reduce cognitive load and the risk of making mistakes.

You should make use of the language's properties as follows:

1. Begin your conversation with simple questions using only basic vocabulary.
2. Employ means of combination as soon as you need to refer to multiple (prior) responses in a single follow-up question.
3. Employ means of abstraction to avoid repetition and verbosity in your questions.
4. As you learn more about the topic of your conversation along the way, decompose important parts into sub-questions and compose follow-up questions from these.
5. Your final question should encompass *everything* you want to get out of the current conversation.

4.4 Consequences

If you can manage to reduce the conversation to a single (or few) composite question(s), your cognitive load will be reduced, too, because the environment can process even millions of data points without making any mistake. Yet, you have to be sure to ask the correct questions or you will be faced with millions of misleading results. Overall, an "Elaborate Inquiry" is more prone to errors than a couple of simple(r) questions.

While the environment can be optimized to ensure quick response times for simple questions using only standard vocabulary, complex questions might take longer to process. Without taking extra care, your "Elaborate Inquiry" can easily include operations whose time-to-process grows exponentially. A progress indication can help you assess the impact of your question, but then you need to be able to pause or cancel the computation to fix your mistake.

4.5 Related Patterns

Especially if you want to "Coach Your Environment," following "Elaborate Inquiry" can help you author side effects.

5 Coach Your Environment

Extend the environment's knowledge by adding information to the current interaction context. Extend its language vocabulary by adding tools for reference in follow-up questions. Both kinds of additions can either come from your memory (and experience) or be loaded from external resources.

5.1 A Desire for Exploration

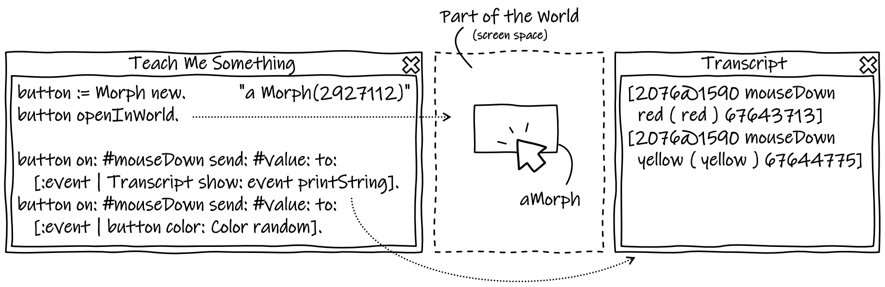
Think about a conversation with another person. There, you can give instructions like “Imagine that you are visiting a foreign city.” You can keep on describing the situation like “Now your smartphone stops working.” And finally, you can ask questions like “How would you find the next good Italian restaurant?” The other person is able to learn about (imaginary) constraints to then reason within those constraints while answering questions. Your programming environment can do that, too. You just have to coach it.

Your questions can be as expressive as the environment’s vocabulary. Thus, if you install new tools, that vocabulary can grow. For example, there are tools for static and dynamic analysis that allow for very specific questions about your system’s modular structure.

The environment’s responses are only as informative as the data they are based on. Thus, if you load more information from external sources into the current interaction context, the same questions can yield more meaningful responses. For example, if you ask questions about photographs (e.g., number of recognized faces), having more photos available will increase the responses’ quality.

You can also coach your environment by writing prototypical programs. In object-oriented systems, programmers often explore the interfaces of objects. They instantiate classes, send messages to the instances, and observe the (side) effects. It is still a conversation with the environment, but with a focus on selected objects in particular. Side effects play an important role when “setting up the scene” before any meaningful question can be asked.

Provided that you wanted to learn more about event handling, you prepared the scene as follows:



```
button := Morph new.          "a Morph(2927112)"
button openInWorld.

button on: #mouseDown send: #value: to:
  [:event | Transcript show: event printString].
button on: #mouseDown send: #value: to:
  [:event | button color: Color random].
```

```
[2076@1590 mouseDown
 red ( red ) 67643713]
[2076@1590 mouseDown
 yellow ( yellow ) 67644775]
```

First, you instantiated a `Morph` (i.e., a basic shape) and positioned it on the screen (here: the world). Then you configured an event handler to react on mouse clicks. The handler will either log information on the `Transcript` (here: `event printString`) or change the `Morph` to a random color (here: `Color random`). Clicking this prototypical button is still part of the conversation about understanding event handling. Every click is a question, the change of color a response. You taught the environment how to do this.

5.2 The Profile

You are in the middle of a “Conversation in Context.” You repeatedly question the quality of the environment’s responses. You get the feeling that the environment’s knowledge about the current topic is too limited.

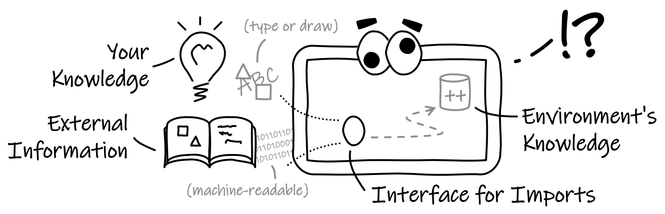
However, you cannot think of a better way to phrase your questions so that the environment might provide a satisfactory response.

- It is difficult to know everything about a topic.
- It is easy to learn something new during a conversation.
- Your questions must use the vocabulary from the environment’s language.
- The quality of responses depends on the environment’s knowledge (i.e., current interaction context).
- Both language/vocabulary and knowledge/context can be extended during conversations.

Therefore, you tell the environment to remember new things so that it can react differently to your follow-up questions. Technically speaking, your interactions with the environment should have *side effects*. You can add information to the current context by writing down *what you know* in a primitive form (e.g., text, numbers, records). You can also tell the environment to load *external information* from outside its boundaries (e.g., host file system, network storage, Internet/Web content).

Considering the environment’s language, you can add new vocabulary like you would in an “Elaborate Inquiry” using, for example, labels (or names) for new classes/methods or the information you just loaded into the interaction context. Note that you can also load *external tools* into the environment, which usually bring along additional vocabulary as well. This extra vocabulary can help you make your follow-up questions more precise.

5.3 The Structure



The environment should provide the following services:

- **Extensible Context**, which is the common representation of knowledge in the environment (e.g., files, objects, tables); thus, it should remain open for ad-hoc additions during your exploratory journey.

- **Extensible Vocabulary**, which is your primary source of "building blocks" for posing (or revising) questions; thus, it should remain open for ad-hoc additions during the conversations with the environment.
- **Interface for Imports**, which allows for loading both machine-readable and directly/manually entered information from outside the environment.

You are responsible for coaching your environment in time:

- Type or draw the facts you already know directly from *your* memory to then build upon that knowledge in follow-up questions.
- Investigate the world outside the environment to learn about relevant external information.
- Be precise: do not load everything you find directly into your environment but consider the *relevance* of your current "Concept in Shards."
- Be thoughtful: construct a "Proxy Transport" for data that is hardly reachable or memory-heavy.

5.4 Consequences

If you are not careful, you can break the environment's current context. For example, you can get punished for your coaching if you overwrite existing knowledge with useless (or wrong) facts. Follow-up questions will then yield misleading results. You can also tamper with the language vocabulary in the sense that once meaningful questions cannot be processed anymore. Fortunately, many environments support "undoing" your latest mistakes to then continue the conversation in a fruitful manner.

The state of having too limited knowledge can quickly turn into an oversupply of information. It can then be difficult to narrow down a question to get a response that you can actually understand. Some external sources might be of poor quality, which will further increase your cognitive effort because you will constantly have to scrutinize the value of the environment's responses. Having no information on a topic is easy to check, an oversupply, however, time-consuming to explore.

Your questions might head into the wrong direction, which might be why the environment seems to have no valuable responses for you. Instead of loading new external information, you might want to backtrack and double-check the point where your environment became less helpful. Maybe you made a mistake earlier that needs to be corrected.

5.5 Related Patterns

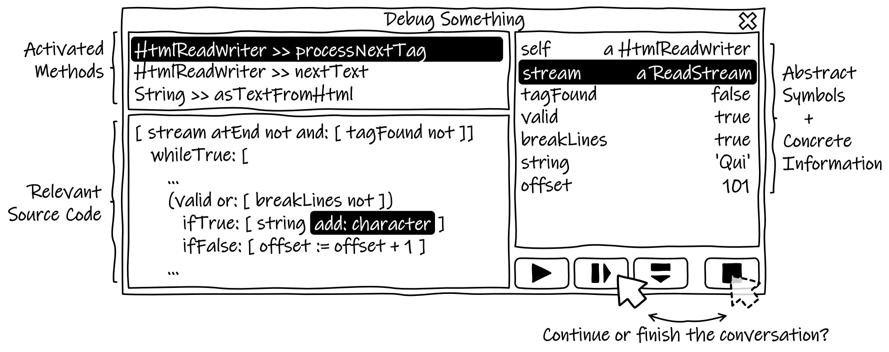
Both "Coach Your Environment" and "Elaborate Inquiry" are close friends.

6 Concept in Shards

Relevant information might be scattered all over the environment. Do not be satisfied with the first useful response. Keep asking questions to reduce guesswork and to gain more substantial insight. If necessary, import external knowledge from outside the environment.

6.1 A Desire for Exploration

The *symbolic debugger* is a conversational tool that manages to connect pieces of information that can be otherwise scattered across modular boundaries (e.g., packages, classes, methods). Debuggers interrupt the running system and offer a conversation about the system's dynamic *control flow* – a snapshot that entails many concrete details about abstract code. While details can be overwhelming at times, programmers have the chance to explore first-hand, reliable facts if they manage to persevere. Here is a typical interface of a debugger in an object-oriented environment:


Debug Something
Activated Methods
Relevant Source Code
Abstract Symbols + Concrete Information
Continue or finish the conversation?

```
HtmlReadWriter >> processNextTag
HtmlReadWriter >> nextText
String >> asTextFromHtml
```

```
[ stream atEnd not and: [ tagFound not ]]
  whileTrue: [
    ...
    (valid or: [ breakLines not ])
      ifTrue: [ string add: character ]
      ifFalse: [ offset := offset + 1 ]
    ...
```

```
self        a HtmlReadWriter
stream        aReadStream
tagFound            false
valid                true
breakLines           true
string              'Qui'
offset                101
```

However, it can be tempting to prematurely jump to conclusions. Exploring the control flow step-by-step is not without efforts. Programmers have to collect and assess the pieces they find. Thus, the conversation might become cumbersome, depending on how the debugging tool helps you manage your findings. Sometimes, you can only go forward and not backwards, which is problematic when exploring systems that have many (stateful) side effects. Sometimes, you cannot restart (or repeat) the conversation if the circumstances (i.e., the system state) are difficult to recreate. Therefore, you should reflect on the value of every debugging situation that you encounter. Do not give up too soon.

6.2 The Profile

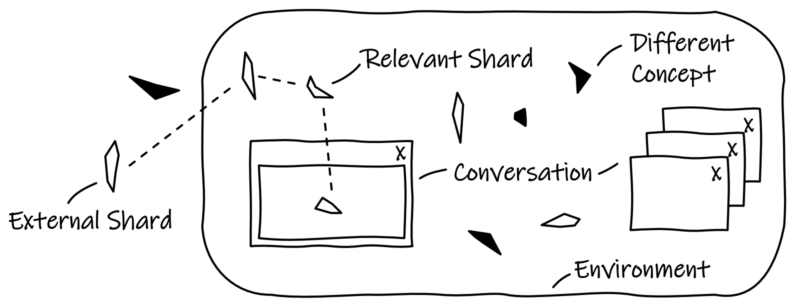
You are in the middle of a “Conversation in Context.” You just got the first response that has information you are looking for.

However, you notice that several important details of the conversation’s topic are still unclear or missing. You wonder whether you should start to “Coach Your Environment” about what you know from *your own* experience. But then you remember:

- The environment’s knowledge (i.e., current interaction context) encompasses many (software) artifacts.
- A concept (or topic) might reach across artifact (or module) boundaries.
- It is easy to ask follow-up questions.
- It is possible to combine multiple responses in an “Elaborate Inquiry.”

Therefore, you keep on having this conversation to collect more details. The concept you are talking about lies “in shards” virtually in front of you. You just have to pick up each individual piece to appreciate its value. Do not stop at the first piece but reason over a *comprehensive* amount of information. While you might not be able to collect *all* shards of a concept, the relevant fraction will bring you further. Note that there might be relevant knowledge located *outside* your environment. Therefore, you might have to “Coach Your Environment,” after all, in how to access the external knowledge.

6.3 The Structure



The environment should provide the following services:

- **Means of Collection** that allow you to manage “the relevant whole” you have identified so far.
- **Means of Comparison** that allow you to quickly determine the added value for each new piece you find.
- **Means of Loading** that allow you to load external information from outside the environment.

You should design your exploratory journey as follows:

1. Stay nosy: keep asking questions and don't be satisfied with the first promising response.
2. Finish the conversation when it becomes tricky to find anything new and relevant, yet don't give up too early.
3. Don't worry: you can often bring up the same topic later in a different conversation.

6.4 Consequences

When *collecting* pieces of information, you rely on efficient window management since each piece might come with a dedicated view that cannot be embedded into the conversation directly. When you then "Pause and Explore" such views, the extra conversations could derail your train of thought. You have to be extra careful not to lose focus.

When *comparing* pieces of information, it can be challenging to assess the quality of similar pieces coming from different (external) sources. If you accumulate too much similar information in your conversation's interaction context, follow-up questions can become tricky to pose or their responses cumbersome to handle.

When *loading* external information into the environment, you can face consequences similar to the ones of "Coach You Environment." An oversupply of information can noticeably increase your cognitive effort.

6.5 Related Patterns

While you "Coach Your Environment" to reach a shard outside the environment, a "Proxy Transport" can help you reduce memory consumption within the environment. An "Elaborate Inquiry" can help you connect all relationships in a compact form.

7 Proxy Transport

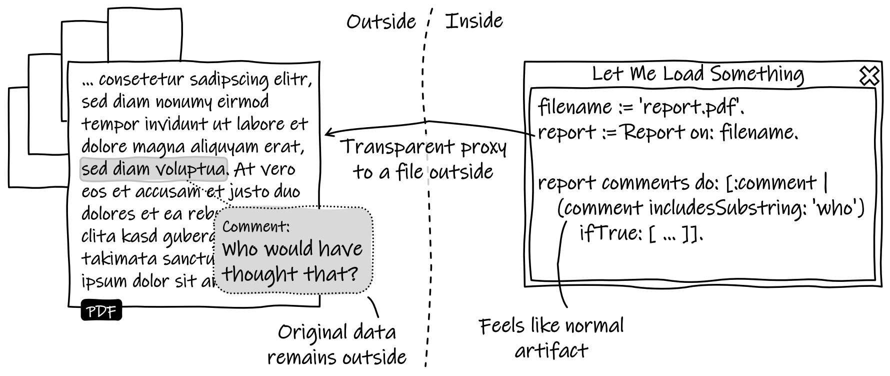
Design a flexible proxy to integrate external information into your interaction context. Map the external format (e.g., file contents) onto the internal one (e.g., object fields). Use *lazy loading* and *caching* to avoid high memory consumption and slow access times.

7.1 A Desire for Exploration

The environment's knowledge is limited. You can surely talk about all the code artifacts that you have created. After all, code management is among the primary features of the environment's tools. However, there is much information only indirectly related to the software system under construction: tickets in a bug tracker, emails in your inbox, or documents in a forum. Chances are that your environment does not manage all these artifacts itself but that other environments take care of it.

There are two main reasons why you should not just *copy* external knowledge into your environment. First, its actuality can quickly be compromised if outside tools keep on updating the information without informing your copy. Second, your resources and your environment's resources are limited. It can take a while to copy data from external environments, which can conflict with the time you estimated for your current task. At some point, your local machine might be out of memory, or the tools need what seems like an eternity to provide meaningful results. Consequently, you have to make external information accessible to be selectively loaded on-the-fly and cached efficiently.

Think about a collection of documents persisted as PDF files on your disk. During the last months, you collected several files and also added much commentary to their contents. Now you want to explore and process this information from within your environment. You have to map the files and its structured contents to objects and fields, which your environment can understand. You might also want to edit selected commentary but keep everything stored in external files to let external tools work on those, too. Eventually, you create a `Report` proxy that is able to read and write PDF files:


Outside
Inside
... consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea reb clita kasd guber takimata sanct ipsum dolor sit a
Comment:
Who would have thought that?
PDF
Transparent proxy to a file outside
Original data remains outside
Let Me Load Something
Feels like normal artifact

```
filename := 'report.pdf'.
report := Report on: filename.

report comments do: [:comment |
   (comment includesSubstring: 'who')
      ifTrue: [ ... ]].
```

You can now have conversations in the same way as you would about artifacts stored inside the environment. Yet, if the external files get moved without notice, the flow in your conversation will break. You have to make the trade-off between availability and efficiency.

7.2 The Profile

You are in the middle of a “Conversation in Context.” You want to “Coach Your Environment” to load external information, maybe because your central “Concept in Shards” pointed to a shard outside the environment.

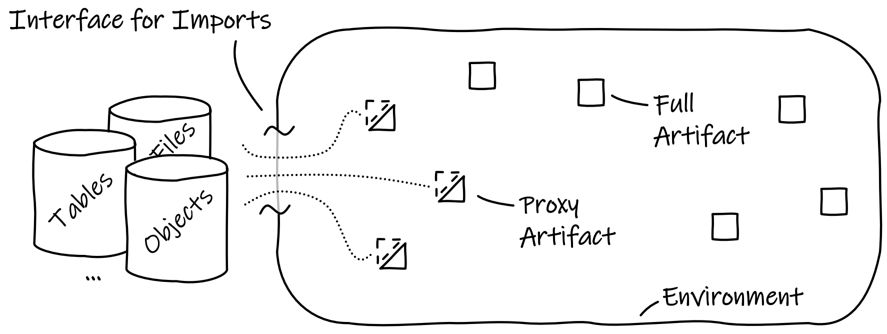
However, you realize that the external information sources you selected are rather demanding in terms of *access time* and/or *memory consumption*.

- It is easy to load external information into the current interaction context.
- The environment relies on the context’s common format (e.g., objects, files, tables).
- It is easy to map an external format (e.g., files) onto the context’s format (e.g., objects).
- It is difficult to find a *good* mapping, meaning, one that fits your conversation’s needs (e.g., file’s contents to object’s fields).

Therefore, you “Coach Your Environment” to make that external information available through *proxies* as part of the current interaction context. A proxy is an informational artifact that has special properties:

- It is *transparent*, which makes it look like normal information stored within the environment.
- It provides *lazy loading*, which only loads the pieces required to process a specific question to avoid high memory consumption.
- It provides *caching*, which stores pieces required for one question to be used in another one to avoid slow access/response times.
- It remains *adaptable* so that you can revise its mapping rules as you revise your questions.

7.3 The Structure



The environment should provide the following services:

- **Means for "Proxification"**, which should be a convenient way for you to "Coach Your Environment" about the useful but costly information outside.
- **Interface for Imports**, which is constantly accessed "behind the curtains" when working with the proxy artifact in conversations. Examples include file handles, database sessions, and network sockets.
- **Means for Debugging**, which is needed when a proxy artifact disturbs the flow in your conversation. The quality of external resources (availability/reliability) is always hard to control from within the environment. The proxy abstraction might "leak."

You should work with external information the same way you do with internal information:

- "Coach Your Environment" to load external information but consider its costs (e.g., access time, memory consumption).
- Stay focused: try to ignore the special characteristics of proxies in your conversations, even if you know about it.
- Be flexible: if a proxy "leaks," plug it and quickly return to your conversational flow.
- If you misjudged the costs, turn the proxy into a fully loaded artifact of the current interaction context.

7.4 Consequences

The biggest issue with proxies is a *leaky abstraction* in the middle of a conversation. For example, if you talk about an artifact that you did not know was a proxy, a sudden increase of response times – because an external resource became unavailable – can break the entire flow of that conversation. You can then either choose to become a *tool builder* to fix it or you can exclude that artifact from your conversation and risk guesswork.

The second biggest issue is "proxification," because in many environments you have to construct a proxy transport for external artifacts yourself. It is typically not a single button click but involves actual programming effort. Depending on the quality of the external source, you might risk not having a caching strategy to be able to quickly return to your conversation. However, a leaking proxy might bring you back to the construction site more quickly than you want. Eventually, programmers often try to bring external information closer to the environment if it cannot be loaded entirely into the environment. For example, information from the Web can be cached in a database on your computer. Your environment can then access that (still external) database without any noticeable lag of a slow network connection.

7.5 Related Patterns

You will have to “Coach Your Environment” to set up the proxies for external information, which can itself be part of your conversation’s central “Concept in Shards.” Note that the *Proxy* pattern is common in object-oriented software design [5], which provides implementation guidelines for tool builders.

8 Simple Response

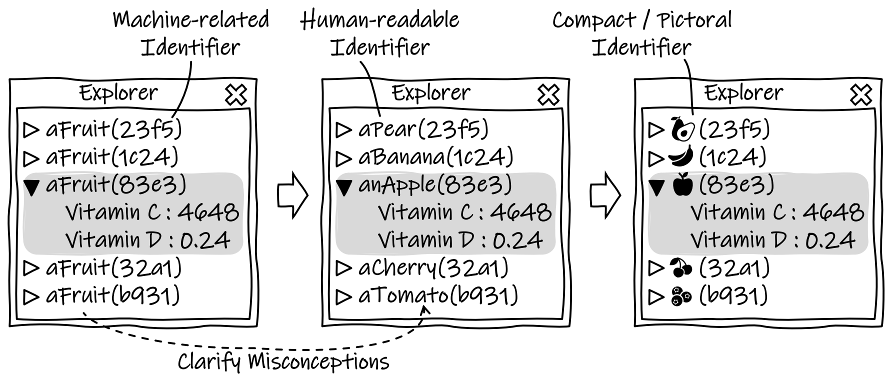
Every question has a simple response. Do not hesitate but ask away; embrace trial-and-error. A text will appear, a picture will render, a sound will play – tackle complexity with a combination of these.

8.1 A Desire for Exploration

Every screen cluttered with a multitude of windows, each one filled with plenty of information, is made of simpler, comprehensible elements. Not just the windows (or views) themselves, but also *their* contents are made of smaller pieces. Color, shape, animation, and interaction help users to distinguish a single piece from its surroundings. We know what buttons look like and that we can click on them. We know that textual labels in a list view can be selected. We know that such labels represent abstract handles to more structured information. Think of your address book as an example with all its details about your friends. Of course, at some point, we had to learn about these elements, but now we know and expect certain things from a graphical interface.

During conversations, the environment will give you responses that look rather simple, even if they entail more complexity. In object-oriented systems, for example, every object has a *print-string*, which is a textual representation of its most prominent (and identifying) features. That print-string is used in many tools to provide you with familiarity during exploration. You will read it and might think: “Ah! I know this object from before.” Therefore, simple responses will keep you oriented and focused during conversations.

However, simple responses can be misleading. If you forgot to “Coach Your Environment” about the identifying features of a new kind of domain object, the environment can only present generic features. Take the following list of fruit objects as an example. On the left, there is only a generic representation, which is difficult to explore. In the middle, the kind of fruit got exposed, which made you realize that `aTomato` might be in the wrong class. On the right, you see a more visual take on each object’s features:



Consequently, it is your responsibility to "Coach Your Environment" about an appropriate representation of informational artifacts throughout the conversation. Only then can you find your path through cluttered screens and be successful on your exploratory journey.

Remember the *Visual Information Seeking Mantra* [21]: overview first, zoom and filter, details on demand. In the end, you browse and explore through a multitude of simple responses. Complex structures are only composites.

8.2 The Profile

You are about to start a "Conversation in Context." A couple of possible first questions begin to form in your mind.

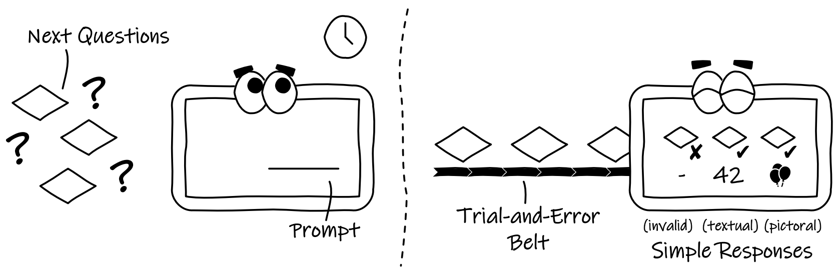
However, you cannot decide on a good first question. You fear that the environment's response might overwhelm you with details you are not yet ready for.

- It is difficult to get a satisfactory response on the first try.
- The environment will not punish you for bad questions.
- It is easy to revise your wording in follow-up questions.
- The environment will provide *some* response.

Therefore, you do ask away your first question and wait for the response. The environment will present a piece of information from the current interaction context in a *simple form*. All artifacts have some kind of textual label, a *name* if you will. Such names help you identify and compare different pieces of information. Naturally, texts will be represented as themselves and so will pictures. Sounds are audible through your speakers. All these simple forms help you assess the quality of your question.

Note that if your question does not make sense to the environment, it will complain in a simple manner, too. Usually, you will be instructed to change your wording or explain the unknown vocabulary. Embrace simplicity; fail early to move on quickly.

8.3 The Structure



The environment should provide the following services:

- Acknowledge any user input, even if malformed.
- Keep it simple: show text, render pictures, play sound or music.
- Wrap complex data structures into simple representations for a pleasant first encounter with the user.

You should embrace the fact that the environment always listens:

- Embrace trial-and-error during conversations.
- Try asking for alternative representations.
- Ask for more details when struggling with unclear (or shallow) representations.

8.4 Consequences

In object-oriented systems, objects with no real-world counterpart can be difficult to represent in a meaningful, simple way. For example, people have names and photographs have colors. Even network sockets have addresses. All these properties can simply be shown on screen to identify the objects behind them. However, purely artificial objects such as observers [5, p. 293] and mediators [5, p. 273] merely have their class name and maybe a location in memory to distinguish individual instances. During conversations, the simple/generic representation of such artificial objects can be cumbersome to use (e.g., `anObserver(1234)` and `anObserver(5678)`).

Users might have accessibility issues with specific on-screen contents due to diminished vision or hearing. Especially an environment that is crowded with information – and multiple ongoing conversations – can be challenging for all kinds of users. The legibility of text might suffer, some colors in pictures might be hard to see, a specific sound might not be audible in a noisy setting.

You might want to change the representation of some artifacts during a conversation, but only for that conversation. For example, while the name of a person might not matter its favorite food does. You therefore "Coach Your Environment" to represent people differently. However, such changes must be scoped to the task at hand so as not to interfere with other conversations or your generic perspective on the domain artifact. We designed the Vivide environment [22] for exactly that purpose.

8.5 Related Patterns

Every “Simple Response” has a non-simple counterpart with a more technical structure. Thus, each response should be a “Tangible Response” since its simple representation usually hides many details that you might want to explore.

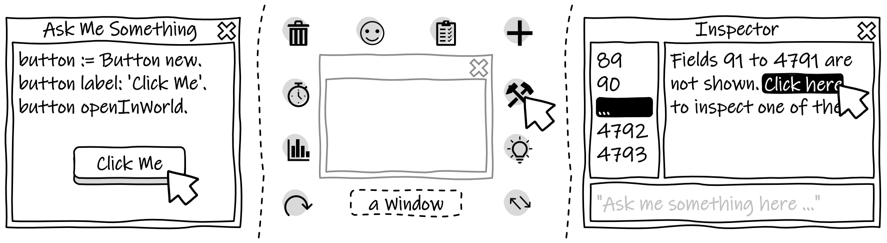
9 Tangible Response

Directly engage (e.g., click on) a response as it is represented on screen. The environment will respond to the “What is this?” question with more details. Return to the particular conversation later on or spawn new ones.

9.1 A Desire for Exploration

During conversations, you and the environment will fill the screen with (visual) information. Your questions take space, and so do the environment’s responses. Therefore, you have to keep track of the meaning of all those visuals. Yet, occasionally, the environment will show you something that you do not quite understand (or have already forgotten about). Then, the follow-up question is simple: “What is that?” You can use touch or mouse input to point at something; you can also use the keyboard to move a cursor to indicate curiosity. In any case, you want to look “behind the pixels” on screen. A textual label might hide details about an important artifact. A graphical composite might be connected to data as well.

In the Web, documents offer clickable (hyper-)links. We have gotten used to hovering over and clicking on anything colorful and suspicious. Yet, your programming environment can offer more tangibility than plain web browsers do. It can help you deconstruct and explore everything you see. The following examples indicate *tangible interactions* during conversations:



```
button := Button new.
button label: 'Click Me'.
button openInWorld.
```

Fields 91 to 4791 are not shown. Click here to inspect one of the

On the left, you told the environment to create and show a button that is now sitting on the screen, waiting to be clicked on. In the middle, you discovered a special gesture to open a *halo* [15] around a window to take a look at its composite structure. On the right, you found a link in a text field that you can click on to explore more information. Overall, your environment can help you explore and understand everything it knows and shows.

9.2 The Profile

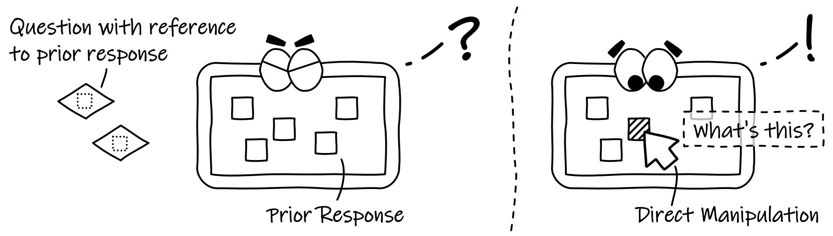
You are in the middle of a "Conversation in Context." You just got an interesting "Simple Response" and want to learn more about its details.

However, you cannot figure out how to ask the environment about details without knowing what lies behind that simple form of the latest response. You are not sure whether there is any vocabulary to express your intent.

- It is easy to touch or click on something that is visible on screen.
- It is more difficult (but possible) to indirectly reference prior responses in a follow-up question.
- A response can be a (generic) question on its own: what is this?

Therefore, you directly engage (i.e., click/touch/activate) with the response as it is represented on screen [8]. The environment will treat this interaction as a generic "What is this?" question, whose response will be more detailed and informative, yet keeping it simple in its parts. You then "Pause and Explore" until you are ready for your next follow-up question. You might already be satisfied and finish the conversation. You might also have many new different questions and thus spawn several new conversations. Note that the "Conversation in Context" maintains a history of prior responses so that you can engage with any of them later on as well.

9.3 The Structure



The environment should provide the following services:

- Provide visual cues that foster the direct engagement with an on-screen representation (e.g., text color).

- Help the user organize exploration paths for deeply structured artifacts.
- Keep track of any sideline conversations that emerge during exploration.

You should hover over, click on, or touch visual cues:

1. Treat any response as a tangible entity that you can engage with.
2. Effectively work with a multitude of simple forms *but* do not forget to probe details so as not to rely on guesswork.

9.4 Consequences

Direct manipulation *for exploration* needs dedicated gestures such as mouse clicks. Yet, graphical, interactive objects already need many gestures for normal interactions. Thus, for tool builders, the design challenge is to find a trade-off between extra/hidden/overlapping gestures to integrate exploratory and normal use. A special *exploration mode* can help to temporarily free reserved gestures from such objects.

Visual clutter affects the size of click targets, which can lead to input errors. It might be easier to indirectly reference a prior response in a follow-up question than to interact with it, even if it is visible on screen. Any “Conversation in Context” should reserve enough screen space for you to compose your question. Even if graphical objects get occluded, you can rely on programmatic references such as variable names.

9.5 Related Patterns

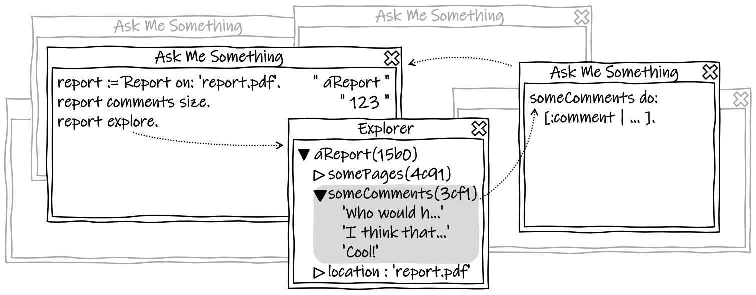
The direct inspection of a response leads to “Pause and Explore” and thus another conversation about the response’s details.

10 Pause and Explore

Suspend an ongoing conversation if you need to explore and understand details of a prior response. Start a new conversation to learn more about these details. Later, resume the original response and make use of your freshly gained knowledge to pose better questions.

10.1 A Desire for Exploration

A single conversation will probably not satisfy all the information needs that arise during an exploratory programming task. Chances are that you hit a block in one conversation that can only be resolved by approaching the topic from a different perspective in another conversation. The environment has several specialized tools that each offer their own conversational interactions. Therefore, you keep on jumping between tools to collect all their insights. The environment's window manager helps you organize screen contents and thus all open conversations:



```
report := Report on: 'report.pdf'.      " aReport "
report comments size.                   " 123 "
report explore.
```

```
▼ aReport(15b0)
  ▷ somePages(4c91)
  ▼ someComments(3cf1)
      'Who would h...'
      'I think that...'
      'Cool!'
  ▷ location : 'report.pdf'
```

```
someComments do:
  [:comment | ... ].
```

Note that, in the process, you can only focus on a single conversation at a time [16]. Also, resuming suspended conversations takes extra time [17]. On the bright side, you can ask very precise questions through specialized tools, whose constraints allow them to also provide very specific responses. After all, your exploratory journey benefits from a rich tool set and your ability to select and integrate appropriate tools as you go.

10.2 The Profile

You are in the middle of a "Conversation in Context." You just exploited a "Tangible Response" and the environment presented many new details you now want to dive into because they seem to be a crucial part of your current conversation. You proceed to start another "Conversation in Context."

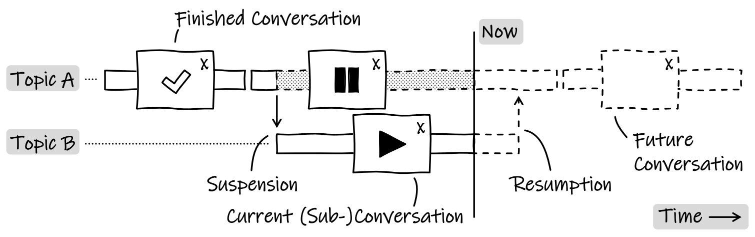
However, you get the feeling that the new information becomes outdated while you are studying it. You feel in a hurry to get back to your original conversation. But then you remember:

- The flow in a conversation will stagnate if you cannot understand crucial information. The risk of mistakes through guesswork will rise.
- Every sample in a pool of information is valuable; pieces allow for extrapolating to the whole concept.
- It is easy to suspend and later resume an ongoing conversation.
- It is easy to spawn new (sub-)conversations to learn more about crucial aspects of/for other conversations.

Therefore, you take a deep breath and dive into the new information you have at hand. You start a new "Conversation in Context" to ask different questions. You suspend the original conversation in the sense that it will be easy for you to resume it later on. The environment does not bother; it can wait and help you shift focus to your new topic on interest.

Note that highly dynamic (and live) systems exhibit quickly changing interaction contexts during conversations. Every response might only capture a single snapshot in time for you to explore. It might be difficult to generalize from a few snapshots. Luckily, it is possible to represent such liveness as *steady frames* [7], which can serve as a more comprehensive topic to talk about with your environment.

10.3 The Structure



The environment should provide the following services:

- Manage to have multiple ongoing conversations at the same time.
- Keep track of topic(s) per conversation.
- Support switching focus between conversations; suspension and resumption should feel effortless.

You should acquire the ability to work with exploratory conversations at a higher level:

- Start a new conversation if you want to dive into the details of a prior response in an ongoing conversation.
- Close (or archive) conversations that you see finished. New conversations can always tackle the same topic(s).

10.4 Consequences

Efficient conversation management relies on a good window manager, just like when collecting your "Concept in Shards." Too many (overlapping) windows will clutter the screen and thus affect click targets of a "Tangible Response." Overall, you might lose your focus and forget about which conversation was spawned because of which other conversation.

You might want to merge two conversations if you realize they lead you to complementary insights. Technically speaking, you might want to merge two (scoped) interaction contexts to then continue your conversation based on a comprehensive knowledge base.

10.5 Related Patterns

Every time you take a break from one conversation to explore data in the context of another one, you will find you way back to a "Conversation in Context" and with it all patterns that come along.

11 Conclusion

We presented the pattern language of an exploratory programming workspace, which captures the conversational style of workspace-like tools such as Smalltalk workspaces, Unix shells, and scientific notebooks for computation. Each pattern in the language offers guidance for programmers as tool users and tool builders. We are aware that there might not be a single implementation out there that covers each and every pattern perfectly. Looking at the consequences we discussed earlier, there can be many practical trade-offs involved in your favorite programming environment. For example, direct manipulation of a "Tangible Response" might be limited in a Unix shell but rather rich in Squeak's Morphic [23]. Programmers have to cope with such limitations on a daily basis.

The impact of a tool builder can be crucial to the success of a project. At best, there are dedicated programmers that have the resources to continuously evolve the tools their colleagues are using. Sometimes, however, it can be beneficial to just realize an idea yourself directly during the programming task without having to talk about it. There are environments that support evolving the tools directly during use such as Squeak/Smalltalk [11] and Lively/JavaScript [14]. Our pattern language can help programmers focus during such construction efforts.

In the future, we want to connect this chapter's patterns to more different real-world projects, tools, and environments. Following traditional patterns of object-oriented design [5], implementation sketches might help tool builders to better understand common pitfalls early on. We think that this pattern language will grow since we already identified more aspects to talk about such as "Surprise Response" and "Conversations Re-used." Change is omnipresent in the lifetime of a pattern as it gets applied and revised along the way.

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. Abelson, H., Sussman, G.J., Sussman, J.: Structure and Interpretation of Computer Programs, 2 edn. The MIT Press (1996)
2. Alexander, C.: The Timeless Way of Building. Oxford University Press (1979)
3. Alexander, C., Ishikawa, S., Silverstein, M., Jacobson, M., Fiksdahl-King, I., Angel, S.: A Pattern Language - Towns, Buildings, Construction. Oxford University Press (1977)
4. Burnett, M.M., Myers, B.A.: Future of end-user software engineering: Beyond the silos. In: Proceedings of the On Future of Software Engineering, pp. 201–211. ACM (2014). DOI 10.1145/2593882.2593896
5. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Longman Publishing Co., Inc., USA (1995)
6. Goldberg, A.: Smalltalk-80: The Interactive Programming Environment. Addi-son-Wesley (1983)
7. Hancock, C.M.: Real-time programming and the big ideas of computational literacy. Ph.D. thesis, Massachusetts Institute of Technology (2003)
8. Hutchins, E.L., Hollan, J.D., Norman, D.A.: Direct manipulation interfaces. Human-Computer Interaction **1**(4), 311–338 (1985). DOI 10.1207/s15327051hci0104_2
9. Iba, T., Isaku, T.: A pattern language for creating pattern languages: 364 patterns for pattern mining, writing, and symbolizing. In: Proceedings of the 23rd Conference on Pattern Languages of Programs, pp. 1–63 (2016)
10. 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`
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. Ingalls, D.H.H., Palacz, K., Uhler, S., Taivalsaari, A., Mikkonen, T.: The lively kernel a self-supporting system on a web page. In: Self-Sustaining Systems, pp. 31–50. Springer (2008). DOI 10.1007/978-3-540-89275-5_2
13. Ko, A.J., Abraham, R., Beckwith, L., Blackwell, A., Burnett, M.M., Erwig, M., Scaffidi, C., Lawrance, J., Lieberman, H., Myers, B.A., Rosson, M.B., Rothermel, G., Shaw, M., Wiedenbeck, S.: The state of the art in end-user software engineering. ACM Computing Surveys (CSUR) **43**(3), 21:1–21:44 (2011). DOI 10.1145/1922649.1922658
14. 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`
15. Maloney, J.H., Smith, R.B.: Directness and liveness in the morphic user interface construction environment. In: Proceedings of the 8th Annual ACM Symposium on User Interface and Software Technology, pp. 21–28. ACM (1995). DOI 10.1145/215585.215636
16. Miyata, Y., Norman, D.A.: Psychological issues in support of multiple activities. In: D.A. Norman, S.W. Draper (eds.) User Centered System Design: New Perspectives on Human-Computer Interaction, pp. 265–284. Lawrence Erlbaum Associates, Inc. (1986)
17. Parnin, C., Rugaber, S.: Resumption strategies for interrupted programming tasks. Software Quality Journal **19**(1), 5–34 (2011). DOI 10.1007/s11219-010-9104-9
18. Pimentel, J.a.F.N., Braganholo, V., Murta, L., Freire, J.: Collecting and analyzing provenance on interactive notebooks: When ipython meets noworkflow. In: Proceedings of the 7th USENIX Conference on Theory and Practice of Provenance, pp. 155–167. USENIX (2015). URL `http://dl.acm.org/citation.cfm?id=2814579.2814589`. `http://dl.acm.org/citation.cfm?id=2814579.2814589`, accessed 2018-11-10

19. Raymond, E.S.: The Art of UNIX Programming. Addison-Wesley (2004)
20. Rein, P., Ramson, S., Lincke, J., Hirschfeld, R., Pape, T.: Exploratory and live, programming and coding. The Art, Science, and Engineering of Programming **3**(1), 1:1–1:33 (2018)
21. Shneiderman, B.: The eyes have it: A task by data type taxonomy for information visualizations. In: Proceedings 1996 IEEE Symposium on Visual Languages, pp. 336–343. IEEE (1996). DOI 10.1109/VL.1996.545307
22. 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`
23. Taeumel, M., Hirschfeld, R.: Evolving user interfaces from within self-supporting programming environments: Exploring the project concept of squeak/smalltalk to bootstrap uis. In: Proceedings of the Programming Experience 2016 (PX/16) Workshop, pp. 43–59. ACM (2016). DOI 10.1145/2984380.2984386
24. Taeumel, M., Hirschfeld, R.: Exploring modal locking in window manipulation: Why programmers should stash, duplicate, split, and link composite views. In: Companion Proceedings of the 5th International Conference on the Art, Science, and Engineering of Programming, pp. 14–20 (2021)
25. Taeumel, M., Rein, P., Hirschfeld, R.: Toward patterns of exploratory programming practice. In: Design Thinking Research, pp. 127–150. Springer (2021)
26. Tanimoto, S.L.: A perspective on the evolution of live programming. In: 2013 1st International Workshop on Live Programming (LIVE), pp. 31–34. IEEE (2013). DOI 10.1109/LIVE.2013.6617346