The Exploration Workspace

Interleaving the Implementation and Use of Plain Objects in Smalltalk

Patrick Rein
Hasso Plattner Institute
University of Potsdam
Potsdam, Germany
patrick.rein@hpi.uni-potsdam.de

Robert Hirschfeld
Hasso Plattner Institute
University of Potsdam
Potsdam, Germany
robert.hirschfeld@hpi.uni-potsdam.de

ABSTRACT

When engaging in exploratory programming, the activities of exploring, understanding, implementing, and using objects of a particular domain should ideally be tightly interwoven to allow for short feedback cycles and continuous progress towards desired levels of comprehension and knowledge. However, when working with plain data objects using state-of-the-art development tools, programmers often have to resort to a more sequential workflow in which they first need to understand the model, then implement it, and only afterwards can start using and exploring it.

We propose the *exploration workspace* tool which enables programmers to interleave these activities to better support the exploration of objects they are not fully acquainted with. We achieve this by trying to tolerate deviations of the assumed protocol from the protocol actually provided by the objects the programmers wish to interact with. For that, we strive for non-intrusive tool support that helps to semi-automatically build up the vocabulary preferred in interactions. We also suggest to consolidate both domain object implementations and exploration scripts once learning progressed.

Through an example scenario, we will illustrate various resolution strategies applied during the implementation of a geographical map showing social media posts and photos.

CCS CONCEPTS

• **Software and its engineering → Integrated and visual development environments**; *Classes and objects*;

KEYWORDS

exploratory programming, live programming, tool support, Smalltalk, Squeak

ACM Reference Format:
Patrick Rein and Robert Hirschfeld. 2018. The Exploration Workspace: Interleaving the Implementation and Use of Plain Objects in Smalltalk. In *Proceedings of 2nd International Conference on the Art, Science, and Engineering of Programming (<Programming'18> Companion).* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3191697.3214339

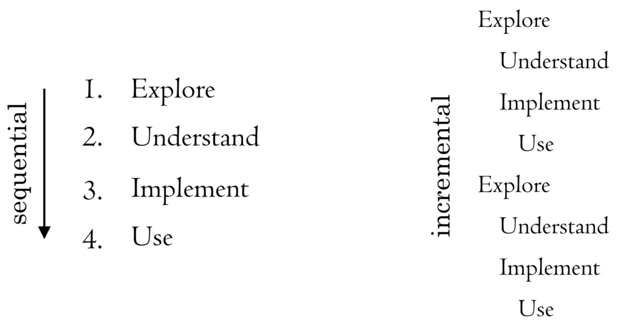


Figure 1: Two approaches to working with data objects in an exploratory environment. The first approach organizes rather big steps in a sequential manner. The second one favors to repeatedly interleave finer-grained activities for shorter feedback cycles and a more continuous exploration experience.

1 COMPREHENSION, IMPLEMENTATION, AND USAGE IN EXPLORATION

Exploratory programming describes a workflow for situations in which requirements and appropriate solutions are yet to be determined [6, 8]. Through this workflow, programmers can experiment with possible solutions to determine a suitable way forward. *Exploratory programming environments*, such as several Lisp and Smalltalk implementations, support this workflow by providing mechanisms and tools for the introspection of run-time behavior and state and by allowing programmers to change the system while it is running [2, 5, 7]. In the case of Smalltalk, programmers can explore the run-time state of individual objects, invoke behavior on these objects, and change their behavior by changing the method definition in the corresponding classes. The underlying activities of *using* objects, *understanding* the domain represented by the objects, and *implementing* new behavior and solutions are tightly interwoven and thereby support an exploratory style of development (see Figure 1). These activities are already well supported whenever the data interesting to programmers is already represented as domain objects, and the interface of these domain objects has a fairly complete implementation.

However, for example when working with data retrieved from modern web services, programmers often start with data not yet properly represented as domain objects with an extensive interface. In this case, the experimentation is put to a halt. For example, when loading objects directly from a JSON file into a Squeak/Smalltalk image (which will serve as an example exploratory programming environment), the resulting `JsonObject` instances only provide access to their data. Nevertheless, programmers want to script and explore solutions based on these data objects by using domain-specific operations, especially so, when the programmers have an intuition about the interfaces those domain objects should provide.

For example programmers might want to implement a media map showing social media posts and photos from external data sources. The social media posts and meta-data on the photos were loaded as data objects into a Squeak/Smalltalk environment. Both kinds of objects contain geographical positions. In this scenario, programmers might interpret a data object as a geographical location. When programmers explore a solution using these simple objects, they might, based on their intuition about the domain, want to ask for the geographical distance between two such objects. However, the data objects do not allow for that, yet. While they might allow programmers access to their internal state, they do not provide domain-specific methods. This discrepancy between the aassumed interface of these objects and the actually available interface has the potential to disrupt the train of thought of programmers. They have to switch from thinking about their current variation of a solution to reasoning about the technical details of the provided interface.

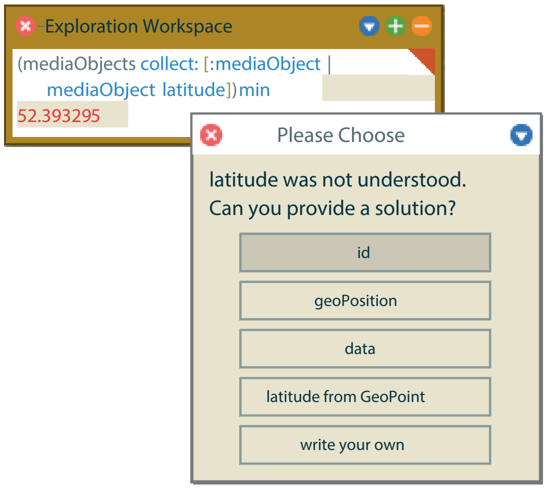
Consequently, programmers have to take a step back from the exploratory programming approach and resort to a sequential programming workflow (see Figure 1). First, they try to *understand* the internal structure of the incoming data objects. Then, they can *implement* a corresponding domain object class and the conversion of the data objects to domain objects. Finally, only after these steps are finished, the programmers can start *using* these objects in their exploration and for working on a solution within the actual problem domain. For example, programmers could first devise an appropriate model for geographical locations before loading the incoming data into objects of this model and then start exploring what could be done using the locations.

Ideally, programmers would start exploring a solution using the data objects right away, interleaving the activities of *understanding*, *using*, and *implementing* right from the start (see Figure 1). However, this interleaving might require programmers to constantly switch between using the domain objects, implementing basic domain logic, converting from data objects to domain objects. A major part of this basic domain logic is often the mapping of the names of fields in the data objects to domain vocabulary, as well as extending the interface with convenience methods. These *small* but *frequent* context switches can interrupt the exploration process and the flow of thought of programmers. Thus, we propose an *exploration workspace* for Squeak/Smalltalk [3] that integrates such fine-grained implementation tasks more tightly with the usage of data objects during an exploratory programming workflow.

2 THE EXPLORATION WORKSPACE

We propose a tool called *exploration workspace*. It is designed to ease the burden of context switches for minor implementation tasks during exploratory programming activities involving data objects.

To introduce the tool, we describe an exploration of the aforementioned media map that shows media objects (for example social media posts or photos) with geographical coordinates on a map. In this example, the programmers use data from two different data sources, a social network and a photo sharing platform. While the data objects from both sources include geographical coordinates, those coordinates are represented through different structures.



```
(mediaObjects collect: [:mediaObject |
    mediaObject latitude]) min
52.393295
```

latitude was not understood.
Can you provide a solution?

Figure 2: A screenshot of an exploration workspace in Squeak/Smalltalk. In addition to its regular workspace behavior, it also helps to automatically mitigate minor inconsistencies between messages sent and methods actually implemented. Here the message `latitude` is sent to but not understood by the `mediaObject`. Our exploration workspace offers a few options to help the programmer to proceed: The first three methods are already understood by the `mediaObject` and the fourth method can be added by a trait. And of course, if none of the proposals seem helpful, one can write the missing code by falling back to the regular programming infrastructure.

2.1 Features

The exploration workspace is an adapted Squeak/Smalltalk workspace (see Figure 2). The workspace assumes that the programmer calls methods which semantically fit the available objects and thus the workspace will automatically try to solve minor incompatibilities which would normally require programmers to write code. It uses a combination of two strategies: normalizing selector names (for example between camel-case and underscores) and mapping synonymous selectors using WordNet [9].

Further, if there is no obvious mapping from our desired message to a method already implemented by the receiver, the workspace will present a list of options on how to proceed. First, programmers can choose any existing method of the object as a replacement for the method being called. Second, programmers can select a method from a number of *traits* which might already be available in the environment [1]. The workspace only suggests a trait if the object provides the methods required by it. Last, programmers can choose to implement a method from within the workspace if no other option matches the desired functionality.

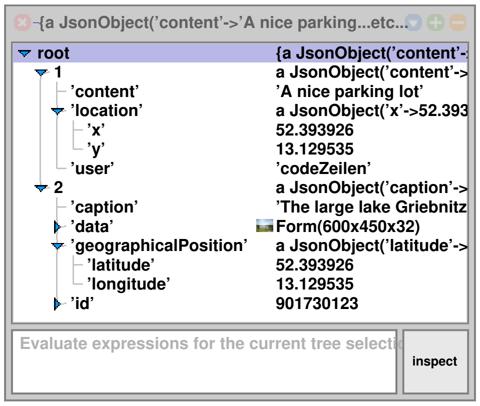
Any decision, taken by either the workspace or the user, is stored in the workspace. If a new mismatch arises, the exploration workspace first tries to resolve it by using the previous decisions. Further, the decisions are scoped to code executed from within that particular workspace. A second exploration workspace can contain different decisions. After an exploration programmers can consolidate the decisions and materialize them into source code as a starting point for a follow-up implementation.

2.2 Example: Building a Social Media Map

The following walkthrough illustrates the interactions between programmers and the exploration workspace. In our example, the programmers want to explore how a basic structure of a social media map could work. As a starting point, we will load the data objects from two JSON files into the exploration workspace and then start exploring.

```
map := KnotsMap new openInWorld; yourself.
tweets := Json loadFrom: (FileDirectory default
 readOnlyFileNamed: 'tweets.json') readStream.
photos := Json loadFrom: ...
```

In order to see what is available, the programmers first open an object explorer on a tweet and a photo:

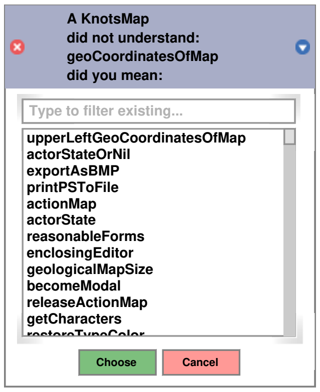


```
{a JsonObject('content'->'A nice parking...etc...
root                          {a JsonObject('content'-
  1                           a JsonObject('content'->
    'content'                 'A nice parking lot'
    'location'                a JsonObject('x'->52.393
      'x'                     52.393926
      'y'                     13.129535
    'user'                    'codeZeilen'
  2                           a JsonObject('caption'->
    'caption'                 'The large lake Griebnitz
    'data'                    Form(600x450x32)
    'geographicalPosition'    a JsonObject('latitude'->
      'latitude'              52.393926
      'longitude'             13.129535
    'id'                      901730123
Evaluate expressions for the current tree selecti
```

The programmers look at the first object and see that the object stored in the location field contains geographical coordinates. Thus, they start exploring a possible algorithm for displaying them on the map by trying the following:

```
tweets first location
 distanceTo: map geoCoordinatesOfMap.
```

It turns out that they forgot the correct selector for the geographical coordinates of the map and thus the exploration workspace shows the following dialog:

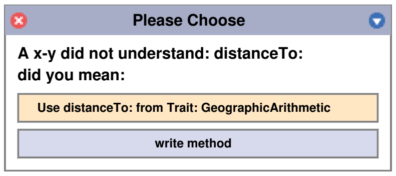


A KnotsMap
did not understand:
geoCoordinatesOfMap
did you mean:

Type to filter existing...

upperLeftGeoCoordinatesOfMap
actorStateOrNil
exportAsBMP
printPSToFile
actionMap
actorState
reasonableForms
enclosingEditor
geologicalMapSize
becomeModal
releaseActionMap
getCharacters

They select the first selector and the execution continues. As the object resulting from sending `location` to the tweet object is merely a `JsonObject` it does not understand the domain-specific message `distanceTo:`. However, the system includes a trait with methods for geographic arithmetic operations which includes a `distanceTo:` method. Thus it proposes the following:



Please Choose

A x-y did not understand: distanceTo:
did you mean:

The dialog shows the list of fields as the type of the object ("x-y"), as the object is not an instance of a domain-specific class. The programmers recognize that it refers to the location object of the tweet and as the `GeographicArithmentic` Trait sounds promising they select it. However, this trait requires an object to understand latitude and longitude, so subsequently two more dialogs open up asking for an interpretation of latitude and longitude in the context of the location object. After these questions have been answered, the expression evaluates successfully. Based on this expression they develop the following snippet to place objects on the map:

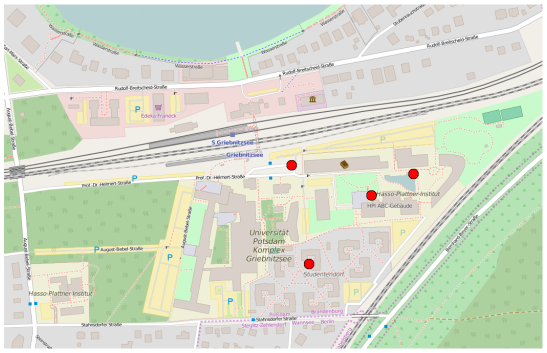
```
(tweets collect: [:o | o location distanceTo:
 map geoCoordinatesOfMap])
  do: [:point |
   map addMorph: (CircleMorph new
    position: map topLeft + (point * map mapScale)abs;
    color: Color red;
    openInWorld )].
```

For now the programmers are satisfied with the script and want to extend it to work with photos also. They think that the logic of the script should also apply to photo objects as they also have a geographic location. So, to visualize the photos they change the collection of objects to:

```
(tweets , photos collect: [:o | o location ...] ...)
```

From a technical perspective this script includes another interface mismatch, as photo objects store the geographical location in a field called `geographicalPosition`. However, when executing the script this time, the exploration workspace resolves the mismatch automatically because the two selectors are synonymous according to the synonym database. The only prompt presented to the programmers on this second execution is the question where `distanceTo:` should be taken from. This is necessary as the photo location objects have fields different from the tweet location objects and thereby the previous decision does not apply to them.

After successfully running the script the programmers see the map with red markers showing the locations of tweets and photos:



3 IMPLEMENTATION

The implementation of the exploration workspace is based on compatibility layers [4]. It consists of the mechanism to detect a mismatch and a number of resolution strategies.

We detect an interface mismatch by reacting to dispatch errors. Therefore, we adapted the `doesNotUnderstand:` method on the `Object` class. This adapted implementation checks whether the call originated from code executed in an exploration workspace. We determine this by traversing the stack looking for a call to an exploration workspace. If there was such a call, the workspace is returned and used as the scope for the following resolution process.

After detecting the mismatch the aforementioned resolution strategies are applied in the following order:

(1) Resolution using existing decisions
(2) Lexical normalization
(3) Synonym lookup through WordNet
(4) Proposing manual resolution: existing methods, matching traits, writing ad-hoc implementation

The decisions are stored per selector and class of the called object. For data objects without a domain-specific class the alphabetically sorted list of field names is used as a class identifier.

The lexical normalization strategy converts camel-case selectors with colons to a lowercase string separated by underscores. To determine synonyms, we also use a Smalltalk WordNet database [1] normalized by the same rules. We are conservative in looking up synonyms and only take synonyms from the synset for the desired selector. If there is no designated synset for that selector the synonym lookup fails.

4 CONCLUSION AND FUTURE WORK

We propose exploration workspaces to improve exploratory workflows that involve data objects. While our current focus is mostly on the exploration of data obtained from external sources, we believe that exploration workspaces can also be beneficial when programming with objects with extensive interfaces. Especially novice programmers might benefit from that approach as they might struggle with the strict interpretation of method selectors while already having an intuitive idea of how to talk to an object.

Future Work

In order to evolve the design of the exploration workspace, more information is required on the particular mismatches happening during exploratory programming. We plan on gaining these insights through a qualitative study of exploratory workflows.

Additionally, the user experience of the exploration workspaces can be improved. The scale of the exploration is limited as programmers have to mentally keep track of resolved mismatches. Further, if they take a wrong decision they can not selectively undo it. Thus, future designs should include some overview of past decisions and mechanisms to quickly undo decisions. As the explored solutions might become more extensive the resolutions might become more extensive, too. When faced with decisions programmers might require more context including information on which decisions have already been used in the current call stack and which decisions might arise depending on the current options. A tree visualization of the potential resolution paths might be of help here. Finally, the user interaction based on dialog boxes can disturb the exploration workflow through regular context-switches. A more integrated design similar to the way programmers interact with code completion might be beneficial.

ACKNOWLEDGMENTS

Sincere thanks also go to all PX workshop participants, who provided valuable feedback by discussing this topic thoroughly. We would like to thank Anton von Weltzien for his version of and perspective on the exploration workspace. We gratefully acknowledge the financial support of HPI's Research School[2] and the Hasso Plattner Design Thinking Research Program.[3]

REFERENCES

[1] Stéphane Ducasse, Oscar Nierstrasz, Nathanael Schärli, Roel Wuyts, and Andrew Black. 2006. Traits: A Mechanism for Fine-Grained Reuse. *ACM Transactions on Programming Languages and Systems* 28, 2 (2006), (331 to: 338). https://doi.org/10.1145/1119479.1119483
[2] Adele Goldberg and David Robson. 1983. *Smalltalk-80: The Language and Its Implementation.* Addison-Wesley Longman Publishing Co., Inc., Boston, USA.
[3] Daniel Ingalls, Ted Kaehler, John Maloney, Scott Wallace, and Alan Kay. 1997. Back to the Future: The Story of Squeak, a Practical Smalltalk Written in Itself. In *Smalltalk and Exploratory Programming*, Vol. 32. ACM, (318 to: 326). https://doi.org/10.1145/263698.263754
[4] Patrick Rein, Robert Hirschfeld, Stefan Lehmann, and Jens Lincke. 2016. Compatibility Layers for Interface Mediation at Run-time. In *Companion Proceedings of the 15th International Conference on Modularity (MODULARITY Companion 2016).* ACM, New York, NY, USA, 113–118. https://doi.org/10.1145/2892664.2892683
[5] David Sandberg. 1988. Smalltalk and Exploratory Programming. *SIGPLAN Not.* 23, 10 (1988), (85 to: 92). https://doi.org/10.1145/51607.51614
[6] Beau Sheil. 1983. Power Tools for Programmers. Datamation Magazine. (1983).
[7] Warren Teitelman. 2008. History of Interlisp. In *Celebrating the 50th Anniversary of Lisp (LISP50).* ACM, New York, NY, USA, (5 to: 5). https://doi.org/10.1145/1529966.1529971
[8] Jason Trenouth. 1991. A Survey of Exploratory Software Development. *Comput. J.* 34, 2 (1991), (153 to: 163).
[9] Princeton University. 2010. "About WordNet". (2010).

[1]Available at https://github.com/hpi-swa/SmalltalkWordNet

[2]www.hpi.uni-potsdam.de/research_school
[3]www.hpi.de/en/research/design-thinking-research-program