Moldable, Context-Aware Searching with SPOTTER

(Preprint *)

Andrei Chiş[1], Tudor Gîrba[3], Juraj Kubelka[2],
Oscar Nierstrasz[1], Stefan Reichhart[4], and Aliaksei Syrel[1]

Software Composition Group, University of Bern, Switzerland, scg.unibe.ch[1],
PLEIAD Laboratory, University of Chile, Chile, pleiad.cl[2],
feenk.com, Switzerland[3], stefan.reichhart@gmail.com, Switzerland[4]

Abstract

Software systems involve many different kinds of domain-specific and interrelated software entities. A common strategy employed by developers to deal with this reality is to perform exploratory investigations by means of searching. Nevertheless, most integrated development environments (IDEs) support searching through generic and disconnected search tools. This impedes search tasks over domain-specific entities, as considerable effort is wasted by developers locating and linking data and concepts relevant to their application domains.

To tackle this problem we propose SPOTTER, a moldable framework for supporting contextual-aware searching in IDEs by enabling developers to easily create custom searches for domain objects. In this paper we motivate a set of requirements for SPOTTER and show, through usage scenarios, that SPOTTER improves program comprehension by reducing the effort required to find and search through concepts from a wide range of domains. Furthermore, we show that by taking code into account, SPOTTER can provide a single entry point for embedding search support within an IDE.

Categories and Subject Descriptors D.2.6 [*Software engineering*]: Programming Environments – Integrated environments, Interactive environments

Keywords search, navigation, IDEs, integration

* In Proceedings of the 2016 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! 2016), November 2–4, 2016, Amsterdam, Netherlands.
DOI: 10.1145/2986012.2986023

[Copyright notice will appear here once 'preprint' option is removed.]

1. Introduction

Program comprehension requires developers to reason about many kinds of interconnected software entities (*e.g.*, code, annotations, packages/namespaces, documentation, configuration files, resource files, bugs, change sets, run-time data structures) [25] often stored in different locations [7]. Dealing with this reality prompts developers to form and maintain task contexts [14] by continuously searching for relevant entities and navigating their dependencies [9, 11, 21]. Cognitive tasks analyses describe this process as a foraging loop in which developers seek, understand, and relate information [18].

Depending on the application domain, software entities are further assigned domain-specific concepts. This improves program comprehension as domain concepts play an important role in human knowledge and software development [13, 19]. For example, an event-based system can use run-time objects to model events, a server can rely on XML files to model descriptors for web services, and a parser can model grammars using methods.

Hence, instead of reasoning just in generic and low-level terms (*e.g.*, *What files named* web.xml *contain within a* <security-role> *tag a* <role-name> *tag with the value "manager"?*), developers commonly formulate their queries using concepts and abstractions from their application domains (*e.g.*, *What web applications use the security role* "manager"*?*).

Nevertheless, although searching is pervasive in software development and maintenance tasks, it is supported in IDEs mainly by means of disconnected and generic search tools. On the one hand, the lack of search tool integration forces developers to manually locate and construct domain abstractions by piecing together information from various sources (*e.g.*, *What XML tags represent security roles? In what files are they defined?*). On the other hand, it impedes discoverability: one has to be aware of a domain abstraction to know what to look for. Nevertheless, given the size of today's systems, awareness of all domain abstractions is not

feasible [17]. Hence, a generic and disconnected approach of integrating searching into IDEs leads to information foraging loops where significant effort is wasted recovering concepts instead of directly reasoning in term of those concepts.

To address this problem and improve program comprehension during information foraging loops we propose that search tools directly enable developers to discover and search through domain concepts. This goal can be achieved if IDEs support developers in creating and managing custom ways to search through their domains. Towards this goal we propose SPOTTER, a moldable framework for enabling contextual domain-aware searching in IDEs by putting customization in the foreground and enabling developers to:

(i) easily create custom searches for domain objects;
(ii) automatically discover searches for domain objects.

SPOTTER targets the foraging loop described by Beck *et al.* (*i.e.*, Search and Filter, Read and Extract, Follow Relations) [2] and leverages a simple object-oriented model for expressing search tools by composing search processors. First, a search processor is a run-time object that expresses an individual search query. Second, every search processor is associated with a software entity from an application (*i.e.*, its target object). To achieve this, all software entities from an application that can have an associated search processor are modeled as run-time objects. On the one hand, this includes domain objects. On the other hand, this also includes other software entities like packages, classes, methods, annotations, files, source code, bug reports, documentation, examples, repositories, configurations, *etc.* This way processors can be uniformly attached to domain objects, source code entities and external resources, given that modern IDEs already provide an object-oriented model for representing code and project related data (*e.g.*, JDT in Eclipse). A developer creates new custom processors by configuring the following attributes of a processor object:

Provider: extracts the data on which the processor operates from the system (*e.g.*, the productions from a parser, the shapes from a visualization).

Preprocessor: alters, if needed, the user-supplied query (*e.g.*, fixes typos, removes white spaces, compiles a regular expression) or improves the query by following various heuristics (*e.g.*, based on natural language processing).

Query engine: extracts a subset of elements from the data provider based on the preprocessed query (*e.g.*, substring matching, regular expressions, similarity threshold).

Sorter (optional): can reorder the filtered results (*e.g.*, based on the frequency of their usage [26]).

Hence, a search processor is an object that knows how certain domain concepts related to its target object are reflected in an application and can restrict textual searches to software entities appropriate for those domain concepts. However, a search processor models just an individual query. To model a complete foraging loop, SPOTTER relies on search steps and exploration sessions: a *search step* captures a step in a foraging loop; an *exploration session* consists of a series of connected search steps. Each step takes as input an object and loads all processors associated with that object. When a user opens a step or enters a textual query all loaded search processors are executed in parallel. For example, opening a search step on a web server loads processors for searching through exposed services and security roles in use. Search results are displayed using a user interface that follows the guidelines for improving search tools proposed by Starke *et al.*: *(i)* skimming through search results, *(ii)* ranking and grouping of results, and *(iii)* exploring result sets [27].

To determine the extent to which SPOTTER can enable domain-aware searching we have implemented it in Pharo[1], as part of the GToolkit project.[2] Due to positive feedback from software developers using our prototype, we have continuously improved it and integrated a stable version into the Pharo 4 release[3]. Currently SPOTTER supports more than 100 different search processors for 30 different data types, created both by us and by the developers of several frameworks and libraries from the Pharo ecosystem. On average, extending Spotter with a new type of search requires 9 lines of code. We discuss these extensions as well as what kinds of search tools can be integrated in SPOTTER in Section 8. We further present a pilot user study exploring how developers create extensions for SPOTTER (Section 6) and a survey looking into how developers perceive and use SPOTTER in practice (Section 7).

Through usage scenarios we show that SPOTTER can address a wide range of domain-specific questions from various domains (*e.g.*, parsing, GUIs, event-bases systems, profilers, compilers, HTTP servers). We also show that SPOTTER can be extended to also support generic searches through code, classes methods, bug reports, run-time objects, *etc.* By doing this SPOTTER can provide a single entry point for embedding search support within IDEs.

The contributions of this paper are as follows:

- Extracting and motivating a set of requirements for enabling domain-aware searching within an IDE;
- Presenting and discussing SPOTTER, a model for integrating domain-aware searching within an IDE;
- Discussing the practical applicability of SPOTTER in providing domain-aware searching and improving foraging loops based on real-world examples;
- An analysis investigating the cost of creating custom extensions for SPOTTER.

[1] `pharo.org`
[2] `gt.moosetechnology.org`
[3] `pharo.org/news/pharo-4.0-released`

2. Requirements

To illustrate how generic approaches lead to wasted effort during information foraging loops, we start with two motivating examples. We then propose and motivate a set of requirements for addressing this problem and discuss how they are currently supported in related approaches.

2.1 Motivating Scenarios

2.1.1 Searching Through a Parser Grammar

PetitParser is a framework for creating parsers that makes it easy to dynamically reuse, compose, transform and extend grammars [20]. Developers create parsers by specifying a set of grammar productions in a class or in a class hierarchy. To specify a grammar production a developer needs to: *(i)* create a method that constructs and returns a parser object for that part of the grammar; *(ii)* define, in the same class, an attribute having the same name as the method. Productions are referred to in other productions by accessing object attributes. Developers can add other helper methods and attributes to a parser class.

Finding a grammar production is a common task during the development of a parser. As grammar productions have associated methods, one way to find a production within a parser class is to use a generic search for methods. Nevertheless, this approach will find methods defined in that class that are not productions; developers need to further check that an attribute with that exact name also exists. If the parser is organized in a class hierarchy, developers are required to browse through the superclass chain when a production is not found in the current class. Another task that often arises when working with grammars is to locate those productions that use a given production. A developer can use a generic approach and start browsing all method that access an attribute, however, as in the previous situation, she will have to check if the accessing methods are indeed grammar productions.

These issues can be directly addressed with the help of two domain-specific searches that allow developers to discover and search through productions in a parser and through productions using a given production. For example, a search through the productions of a PetitParser parser can be instantiated as follows using a search processor:

> *Provider*: extracts from a parser class those methods that are grammar productions (*i.e.*, methods where there exists an attribute with the same name as the method);
>
> *Preprocessor*: parses and compiles the query supplied by the user into a regular expression;
>
> *Query engine*: extracts those productions whose name matches the regular expression;
>
> *Sorter*: orders productions based on the frequency of their usage within the current grammar.

The results of using this processor to search for productions containing the string *"hex"* in a parser for Java code is displayed in Figure 1a. The presented scenario is not unique to PetitParser. Similar situations arise every time code elements (*e.g.*, methods, classes, annotations) have a domain specific semantic, since generic search tools cannot filter out unrelated entities.

2.1.2 Searching Through a Visualization

Roassal is an engine for building visualizations defined in terms of objects and their relations [1]. Developers create a new visualization starting from a set of domain objects by *(i)* mapping different types of shapes and relations to those objects, *(ii)* choosing a layout algorithm, and *(iii)* specifying how properties of shapes and of the layout are computed from the domain objects.

Reasoning about how model elements are rendered, a common task consists in locating those parts of a visualization responsible for rendering that model object. A visualization in Roassal is a run-time object consisting of a composite (*i.e.*, tree) of shape objects. Answering our question requires a developer to search through the composite and locate shapes that render that object. As a visualization is an object, one way to address this question is to navigate through object state using a generic object inspector. Nevertheless, Roassal visualizations are complex objects containing many other attributes, unrelated to the task at hand, leading to a significant effort just for navigating through the object graph.

Providing a domain-specific search that enables developers to directly determine what shapes render a domain object can reduce navigation overhead. This search can be instantiated using a search processor as follows:

> *Provider*: extracts all graphical objects that render the target object associated with the processor from a Roassal visualization;
>
> *Preprocessor*: parses and compiles the query supplied by the user into a regular expression;
>
> *Query engine*: extracts those graphical objects whose class name matches the regular expression;
>
> *Sorter*: orders graphical objects alphabetically based on their class name.

Figure 3b shows an example for this search processor. Such types of searches are not limited to Roassal; they are common when domain objects of interest for a developer are spread across an object graph.

2.2 Requirements Discussion

The scenarios presented in Section 2.1 cover different types of developer questions that can be efficiently addressed through custom domain-specific searches. For this approach to be possible, IDEs should enable developers to create and

work with domain-specific searches. Starting from the presented scenarios we propose the following as a set of minimum requirements towards this goal: *inexpensive creation of search processors*, *support for multiple data sources* and *context-aware searches*.

2.2.1 Inexpensive Creation of Search Processors (REQ1)

Given the wide rage of development tasks and applications, foreseeing all usage contexts of a tool is not possible [25]. A fixed set of searches limits the applicability of a search tool. Enabling developers to create custom searches for their domain entities addresses this problem. Nevertheless, the difficulty of creating a custom search directly influences the usability of such an approach. On the one hand, a domain-specific language for creating custom searches can significantly reduce the cost for certain types of extensions. On the other hand, supporting custom searches through a general-purpose programming language allows for any type of extension. To provide a quick entry point and not limit the types of possible extensions, an infrastructure for domain-specific searching should *support inexpensive creation of common types of searches, while allowing developers to fall back to a general-purpose programming language when advanced extensions are needed.*

2.2.2 Multiple Data-sources Support (REQ2)

The two questions discussed in the previous section require information from two different data sources: source code and runtime. External data (*e.g.*, files) is another form of data source also frequently encountered in developer questions. Giving the wide range of heterogenous data used in software application enabling successful domain-specific searching requires *support that integrates and presents to developers data from multiple data sources.*

2.2.3 Context-aware Searches (REQ3)

The questions discussed in Section 2.1 are sometimes addressed in IDEs through standalone search tools (*e.g.*, tools for query-based debugging, dedicated tools for working with parsers). To take advantage of them, developers have to be aware of their presence and know when they are applicable. Modern IDEs, however, can contain hundreds if not thousands of tools and commands. Finding which are applicable to a given entity in a given context can be a challenging task [15]. An infrastructure encouraging developers to create and work with custom searches should address this issue by *enabling developers to automatically find custom searches applicable for a domain entity, based on the entity, the task at hand, and the developer's task context.*

2.3 Current Approaches

There exists a wide range of software tools focusing on improving program comprehension by combining and integrating multiple search tools and techniques. In this section we present and discuss several tools that support custom extensions for searching through a software system:

Tool	Data model	Extension language	Requirement		
			1	*2*	*3*
JQuery	logic database	TyRuBa	✓*	—	✓*
SEXTANT	XML database	XQuery	✓*	—	✓*
Ferret	sphere model	algebra	?	✓	✓*
Sando	text-based files	internal DSL	✓	—	—
SPOOL	OO model	internal DSL	✓	—	—
SPOTTER	reuses the IDE model	internal DSL	✓	✓	✓

— no support, ✓ full support, ✓* partial support, *?* unknown

Table 1. Feature comparison.

a) JQuery is a code browsing tool that combines the advantages of query-based and hierarchical browser tools [10]. JQuery relies on a knowledge database generated dynamically using the Eclipse API and queried using TyRuBa, a logical programming language augmented with a library of helper predicates for searching through source code.

b) SEXTANT is a software exploration tool that leverages a custom graph-based model [23]. In SEXTANT all sources of a project are transformed to XML, stored in a database and queried using XQuery.

c) Ferret is a tool for answering conceptual queries that integrates different sources of information, referred to as spheres, into a queryable knowledge-base [5]. Each sphere performs its queries using a component/plugin of the Eclipse IDE (*e.g.*, static Java searches are executed using JDT).

d) Sando is code search tool and framework that embodies a general and extensible local code search model [24]. Sando focuses on enabling researches to easily implement and compare approaches for local code search.

e) SPOOL is a reverse engineering environment combining searching and browsing. SPOOL has at its core a repository that stores source code models and provides a query mechanism through which a user can query the model [22].

2.3.1 Customization

JQuery and SEXTANT address customization by selecting a language that best fits the model they use to represent the queried data. Nevertheless, this requires developers to write queries for object-oriented programs using a language based on a different paradigm (*i.e.*, logical programming, functional programming). The query language used in JQuery, while allowing for expressive queries, makes the creation of complex queries difficult even for advanced users. Sando and SPOOL use a different approach: developers create custom extensions by implementing a predefined interface and writing code in the host language (*i.e.*, Java, C#) leveraging a given API. This does not require developers to become fa-

(a) Searching for productions within a grammar

(b) Searching through the users of a production.

Figure 1. Exploring a PetitParser grammar for Java code.

miliar with another language. Sando shows that with this approach complex searches can be implemented in less than 100 lines of code. We were unable to find a discussion regarding the ease of creating conceptual queries for Ferret.

2.3.2 Data-sources

Ferret is the only one of the five selected tools that fully addresses *REQ2* by taking source code, dynamic and historical data into account; SEXTANT, JQuery, Sando and SPOOL are targeted towards the analysis of source code artifacts and do not take the run-time into account. Apart from the discussed tools, many other approaches from the area of feature location combine multiple types of information to improve their results but do not focus on customization. Dit *et al.* provide a comprehensive review [6]. Search tools from current IDEs also allow developers to search through multiple data. For example, *Global Search* in IntelliJ makes it possible for developers to search through files, methods, preferences, tools, menus, *etc.*

2.3.3 Context-aware Searches

SPOOL only allows custom searches to be selected based on the type of an entity (*e.g.*, method, file). SEXTANT, JQuery and Ferret go one step further and attach searches to software entities; applicable searches can than be dynamically selected based on various properties of those entities. They however do not attempt to model the developer's context and select searches also by taking into account that context. A different approach is taken by recommender systems which aim to suggest developers useful tools by recording and mining usage histories of software tools [15]; this however requires usage history information.

2.3.4 Summary

The requirements identified in Section 2.2 are addressed to various degrees in current approaches that focus on integrating searches over multiple data types. This indicates a need for a search framework that focuses on unifying search support within an IDE by addressing all three requirements.

3. SPOTTER in a Nutshell

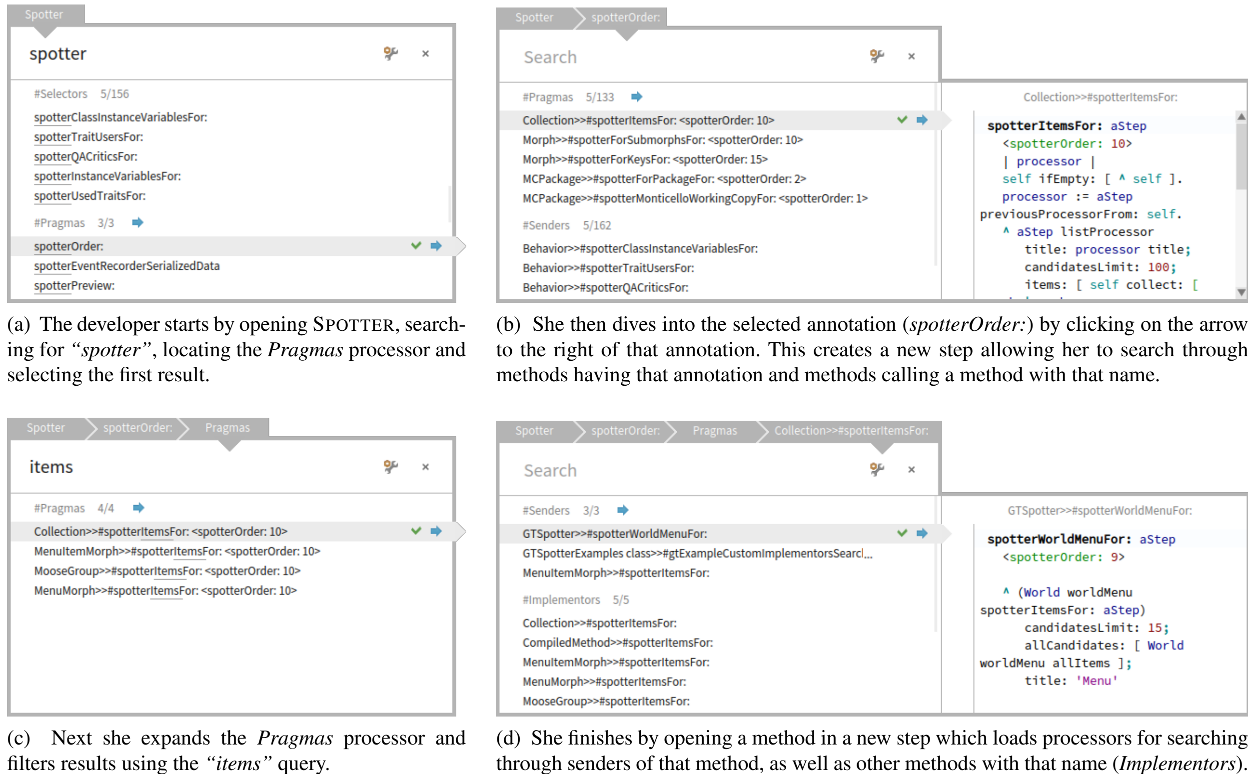
In this section we show how SPOTTER addresses the foraging loop described by Beck *et al.*, we introduce a user interface to support the presented model, and we discuss how SPOTTER models a search context.

3.1 Search Context

As discussed in Section 2.2.3, developers need support for automatically selecting searches relevant for their current needs. In the context of domain-specific object inspection we introduced a solution for modeling a developer's context based on the following operators: *tags*, *sessions* and *activation predicates* [4]. We propose reusing the same approach for enabling SPOTTER to automatically select relevant processors. For completeness, we present these operators and show how they apply to SPOTTER.

Tags identify and group together processors applicable for a development task or application domain. For example, the processors related to PetitParser have the *parsing* tag. Generic processors are grouped using the *default* tag. Only processors that have a tag currently present in the search context are made available to developers. A *session* stores the objects found by a developer using SPOTTER, together with the order in which they were searched for and the search processors used to find those objects. An *activation predicate* is a boolean condition associated with a search processor that is applied on the current search context before loading that processor. Hence, an activation predicate can filter searches based on the state of the current object as well as based on the entire session. For example, the search processor for searching through production in a parser class described in Section 2.1.1, should only be available if the developer opened SPOTTER on a class that represents a PetitParser parser; this processor needs an activation predicate that checks the type of the class.

A search context consists in a set of tags and a session. Only processors whose activation predicate return *true* when applied on the current context are made available to developers.



(a) The developer starts by opening SPOTTER, searching for *"spotter"*, locating the *Pragmas* processor and selecting the first result.

(b) She then dives into the selected annotation (*spotterOrder:*) by clicking on the arrow to the right of that annotation. This creates a new step allowing her to search through methods having that annotation and methods calling a method with that name.

(c) Next she expands the *Pragmas* processor and filters results using the *"items"* query.

(d) She finishes by opening a method in a new step which loads processors for searching through senders of that method, as well as other methods with that name (*Implementors*).

Figure 2. Exploring callers of methods having the *spotterOrder:* annotation (in Pharo annotations are referred to as *pragmas*).

3.2 Supporting an Information Foraging Loop

Starting from the foraging loop for intelligence analysis proposed by Pirolli and Card [18], Beck *et al.* describe a foraging loop for feature location [2] having three main activities: *search and filter*, *read and extract* and *follow relations*. We show next how SPOTTER supports these activities.

3.2.1 Search and Filter

To support this activity, SPOTTER uses *search steps*. A step encapsulates a search on an object in a given context. Each step takes as input a target object and loads all processors that apply to objects of that type in the current context. For example, a search step opened on a method (*i.e.*, on a method object) can load processors for searching through both the callers and the callees of that method. A step opened on a class representing a PetitParser parser in a context that contains the *parsing* tag loads specific processors related to PetitParser (*e.g.*, a search through grammar productions — Figure 1a). If the *default* tag is also in the current context, then generic processors related to classes are loaded (*e.g.*, searches through subclasses, superclasses or instances when the runtime is available).

When a developer opens a step on an object all data providers from the loaded processors are executed and their results are presented to the user according to the order defined by the sorter. For example, when a developer enters the query *'hex'* in Figure 1a, all loaded processors receive the query and update their results. This is the search phase of the foraging loop; developers do not have to select a priori what processors they want to execute. To implement the filter phase, each time the developer provides a textual query, the query engines are used to filter the presented data in each processor.

3.2.2 Read and Extract

Developers can extract initial informations by reading a short textual description of each element present in the results list. As this only gives a limited amount of information, SPOTTER supports a contextual preview for each type of result (*e.g.*, source code for a method, graphical representation for a graphical object or a *png* file). Hence, developers can choose to view each element using the preview. For example when searching for a production in a parser (Figure 1a) the preview of each shape shows the source code of that production. If the preview is not enough, developer can open a result in another tool from the IDE; these tools are selected based on the type of the result entity: code editor for methods, object inspector for objects, *etc.*

3.2.3 Follow Relations

From a search step a developer can choose to continue navigation by selecting a search result. This adds the selected element to the search session and opens a new search step

on the selected result. The creation of a new search step is exemplified in Figure 1: initially the developer searched for productions containing the text *'hex'* and selected the production *hexNumeral* (Figure 1a). Then the developer dived into the *hexNumeral* production, creating a new search step (Figure 1b); in the new search step she can search for productions using the *hexNumeral* production. The search session preserves the navigation history through multiple search steps, and allows developers to reason about how they got to the current step, as well as to go back to previous steps and try alternative searches. Furthermore, processors in the current step can also be filtered based on objects from the previous search steps, not only based on the current object.

3.3 A User Interface for the SPOTTER Model

While the SPOTTER model can express a wide range of searches, the user interface (UI) plays a crucial role in the usability of a search tool. Hence, in this section we introduce and discuss a UI design for SPOTTER based on the three guidelines proposed by Starke *et al.*: *(i)* group and rank results, *(ii)* support rapid skimming through result sets, and *(iii)* enable in-depth exploration of result sets [27]. To illustrate the UI we use a running example in which a developer is interested in exploring callers of methods having a certain annotation.

3.3.1 Grouping and Ranking Results

Running all search processors loaded in a step is at the core of the SPOTTER model. This raises the need of displaying multiple heterogenous result sets at once. We address this by displaying each result set using one level trees: the roots of the trees are the processors that return results and the children are the actual results; results are presented in the order retuned by the processor (Figure 2a).

3.3.2 Skimming Through Result Sets

Given that processors can produce a large number of results, for each processor we display the first n results, where n is customizable per processor (Figure 2a); root nodes are expanded automatically. The label of each result set includes the number of displayed results and the total number of results from that result set. To help developers understand why a result is displayed, the querying engine can embed visual cues when displaying results (*e.g.*, highlight the text matching the query).

3.3.3 Exploring Result Sets

Once a developer has found one or more interesting entities she can investigate them in more detail. SPOTTER supports this through two mechanisms:

Preview pane Developers can obtain more information about a result by opening the contextual preview associated with the current selected result in a pane to the right of the search pane (Figure 2b and Figure 2d).

Navigation Navigation is supported using a simple visual language that consists of two main commands:

- *dive in*: adds the selected object to the search session and creates a new step having that object as target entity. A developer can invoke this action after selecting an element using a keyboard shortcut or by pressing with the mouse the arrow to the right of the selected element (➡). For example, after selecting the annotation spotterOrder: in Figure 2a, the developer executes the dive in action by pressing the arrow; this creates a new search step (Figure 2b). The new search step has as target entity the selected annotation and allows the developer to search through methods having that annotation.
- *expand*: creates a new search step containing a single processor for searching only through the selected result set. This processors shows the entire result set. A developer can invoke this action after selecting a result from a result set using a keyboard shortcut or the arrow next to the label of a result set. For example, after diving into the spotterOrder: annotation (Figure 2b), the developer decides she is only interested in what methods have that annotation. Hence, she expands the result set returned by the *Pragmas* processor (Figure 2b). In the next search step (Figure 2c) she can see all methods having that annotation, explore them in more details and refine her search.

Figure 2b illustrates the final step of the exploration. The developer created this step by diving in a method having the spotterOrder: annotation. In this step she can search through data related to that method, like callers or methods with the same name from other classes. To maintain orientation a breadcrumb shows previous steps.

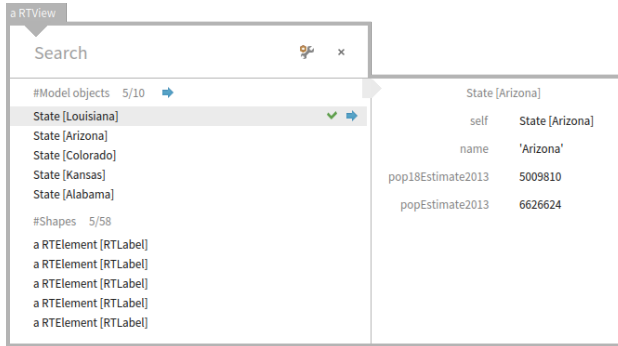
3.3.4 UIs for Feature Location and Exploration Tools

For completeness, we present here a comparison with UIs used in other feature location and exploration tools.

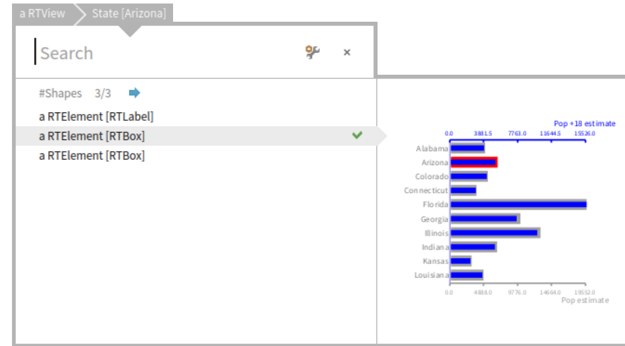
Apatite, a tool for searching and navigating through five levels of an API's hierarchy (packages, classes, methods, actions (methods containing verbs) and properties (getters and setters)) [8] relies on a similar interface for displaying the results of a search step: the first 5 results from each category are automatically presented and developers can see more results on demand. The same UI is also used by *Global Search* from IntelliJ. Nevertheless, Apatite relies on Miller columns to display a navigation session which takes considerably more scree-real estate that the UI used by SPOTTER; *Global Search* does not provide any navigation mechanism.

Ferret relies on a tree view where users need to manually expand each node to see the actual result. JQuery and SEXTANT allow users to discover available searches through a context menu and display the result in a tree or graph.

UIs for feature location focus on allowing users to reason about why a result is displayed. For example, I3, a novel user interface for feature location supporting searches through



(a) A visualization has processors for searching through model elements (*Model objects*) and graphical components (*Shapes*).



(b) Diving into a model object from a step opened on a visualization shows what shapes use that object.

Figure 3. Exploring an object modeling a Roassal visualization.

methods, highlights query terms in the code editor and uses visualizations to convey the similarity of a result with the query as well as show co-change patterns [2]. The SPOTTER UI highlights the text matching the query and uses a contextual preview to convey more information about a result.

4. Improving Information Foraging Loops

SPOTTER aims at enabling direct searches through domain-specific concepts during information foraging loops. In this section we show that SPOTTER addresses this aspect by applying it to the motivating examples discussed in Section 2.1 and highlight how the presented model supports this goal. Generic code related searches are still an integral part of an IDE (*e.g.*, *What method call this method?*, *What are the attributes of this class?*). Not taking them into account would require developers to decide between SPOTTER and a generic tool when they need to perform a search. To avoid this we also extended SPOTTER with support for generic searches (Table 2) based on previous literature, common searches from IDEs and our own developer experience. An example was presented in Figure 2 where, to determine the senders of a method having a certain annotation, the developer *(i)* started by searching for the desired annotation, *(ii)* continued by searching through methods having that annotation, and *(iii)* finished by exploring the senders of one of those methods. This enables foraging loops that combine generic with domain specific searches (Section 4.4).

4.1 Starting SPOTTER

To perform a search the developer has to first open SPOTTER on a target object. When an explicit target object is missing, SPOTTER selects the current workspace as target object. This loads global processors for searching through software entities within the current workspace, including processors for searching through classes, methods, pragmas, history, files, folders, *etc.* SPOTTER registers shortcuts and menu options for performing this action. For example, in Figure 2a the developer opened SPOTTER this way and entered the query text *'spotter'*. To allow developers to specify an explicit target object, SPOTTER further registers shortcuts and menus with other tools from the IDE. For example, a developer can open SPOTTER on any object available in the debugger or the object inspector; this way SPOTTER gets access to run-time state. The code editor makes it possible to open SPOTTER on code entities (*e.g.*, classes, annotations).

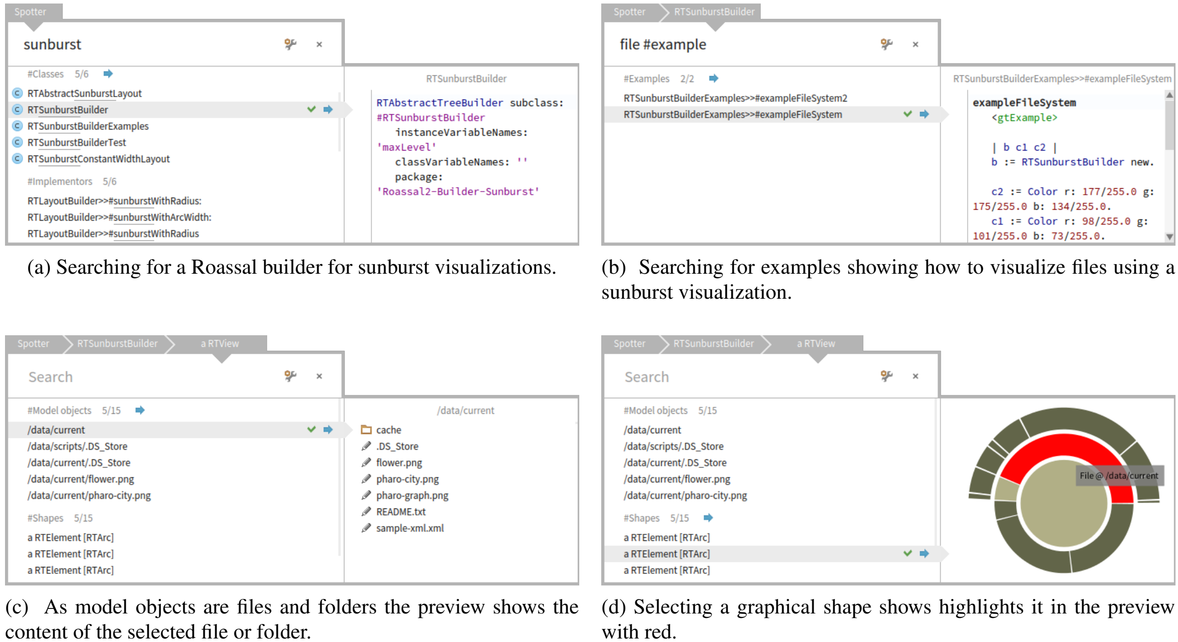
4.2 Finding Productions Within a Parser

SPOTTER supports the two developer questions related to PetitParser discussed in Section 2.1.1 through two processors. The first enables searches through the productions of a parser and has the structure described in Section 2.1.1. As PetitParser parsers are classes, this processor is associated with classes, and has an activation predicate that checks whether the target class is a PetitParser parser (*i.e.*, The class must have *PPCompositeParser* in the superclass chain). Figure 1a shows a scenario in which a developer has opened SPOTTER on a PetitParser parser for Java code and searched for productions containing the word *"hex"* in their name. The preview gives direct access to the source code of the production.

Searching through the uses of a production is achieved using a processor attached to methods and valid when the target method is a PetitParser production. Developers can use this processor by opening SPOTTER on a production or adding a production to the current search session. Figure 1b shows the latter situation where, after searching for a production, a developer uses the *dive-in* feature to add the selected production to the current search session; the preview pane shows the source code of a selected production.

4.3 Searching Within a Visualization

To improve the user experience when developing Roassal visualizations, we created, together with the developers of the Roassal framework, two dedicated processors for searching through visualization objects (*i.e.*, objects of type *RTView*). The first processor (*Model objects*, Figure 3a) enables a



(a) Searching for a Roassal builder for sunburst visualizations.

(b) Searching for examples showing how to visualize files using a sunburst visualization.

(c) As model objects are files and folders the preview shows the content of the selected file or folder.

(d) Selecting a graphical shape shows highlights it in the preview with red.

Figure 4. Finding an example of how to use a sunburst visualization to display files and folders.

search through model objects used by graphical elements; the preview pane shows the state of the model object. The second (*Shapes*, Figure 3a) targets the actual shape objects used in the visualization. As a concrete example, Figure 3 shows SPOTTER opened on a visualization for comparing two metrics about population in several US states using a horizontal double bar chart.

To determine what graphical shapes are used to render a model object a developer can dive in and add the model object to the search session. Doing this loads a processors showing the list of shapes used to render that object. This processor is attached to all objects and has an activation predicate that checks whether the previous object in a search session is a Roassal visualization; if so, the *provider* locates the shapes in that visualization that have the current object as a model. Hence, this processor would not be available when opening SPOTTER on the same object in isolation. In Figure 3b, a developer can see that there are three shapes that render the state Arizona: two *RTBox*es and a *RTLabel*.

4.4 Finding Examples

Roassal comes with a large number of examples. They are organized in classes linked with the graphical classes for which they provide examples. Hence, for each graphical class, if it has an example class, we can provide a processor that displays and searches through its examples. A developer can find and execute an example by *(i)* opening SPOTTER on the current workspace and searching for the desired Roassal class and *(ii)* diving into the class and locating the *Examples* processor (Figure 4b). The preview of an example shows the source code of that example. To get to the actual visualization a develop can then execute the example and dive into the result. As this is a visualization object, the developer can continue her investigation using the custom processors presented in Section 4.3 (Figure 4c).

4.5 Summary

Apart from the presented examples we also added SPOTTER support to other applications from the Pharo ecosystem: GUI libraries (Glamour, Spec, Morphic), Opal compiler, MongoDB bindings for Pharo, Metacello package management system, Monticello versioning system, *etc.* We further added IDE related searches for preferences, menus, help topics, shortcuts, clipboard history, and files and folders.

As we aim to support custom searches through domain concepts, providing an exhaustive list of searches is not feasible. Instead, SPOTTER enables domain-specific adaptations. This section showed that the proposed model can support searches through relevant concepts from different domains that spawn over source code and run-time objects.

By providing both domain-specific and generic searches SPOTTER offers a unified interface for embedding search support within an IDE: developers open SPOTTER on an selected entity and can immediately discover available searches. By using the search session they can further perform targeted explorations that build on previous search results.

5. Implementation

SPOTTER is part of the Pharo IDE since Pharo 4. We developed the first version of SPOTTER over three months and

Global

1) What {method, class, annotation, package, project} names match a regular expression?
2) What methods {call a method, access an attribute} whose name match a regular expression?
3) What {files, folders} are in the {current working directory, a selected file/folder}?

Packages, classes and methods

4) What packages have changes that need to be committed?
5) What are this package's {classes, extension methods, tags}?
6) What are this class' {methods, attributes, static method, static attributes, superclass methods, subclass methods}?
7) What are this class' siblings, super-classes, sub-classes?
8) What classes implement a method with the same name?
9) What attributes/fields does this method access?
10) What methods does this method call?
11) What methods call this method?
12) What methods reference this class by name?
13) What methods have this annotation?

History

14) What are the previous versions of this method, package?
15) Who edited this method before?

Run-time

16) What are the fields of this object?
17) What are the elements of this collection object?
18) What are the keys of this dictionary object?

Table 2. Generic searches supported by SPOTTER

integrated it in the alpha version of Pharo 4. We then continued to develop SPOTTER based on developer feedback (Section 7) in one-month iterations[4].

```
1  spotterForProductionsFor: aStep
2      <spotterOrder: 10>
3      <spotterTag: #PetitParser>

4      aStep listProcessor
5          title: 'Productions';
6          allCandidates: [ :aParserClass |
7              aParserClass productionMethods ];
8          candidatesLimit: 5;
9          itemName: [:aProduction | aProduction selector];
10         filter: GTFilterRegex;
11         itemFilterName: [:aProduction| aProduction selector];
12         when: [ :aClass |
13                 aClass inheritsFrom: PPCompositeParser ];
```

SPOTTER supports the creation of custom search processors through an internal DSL (*i.e.*, API). Developers can configure the predefined blocks of a processor (Section 2.1.1) using anonymous functions or provide a custom implementation for the processor. When creating processors we started by configuring the predefined blocks and switched to a custom stream-based implementation when performance became an issue. For example, lines 4–13 show the code snippet for creating a search processor for PetitParser productions. An activation predicate makes sure that this processor is only available to parsers (lines 12–13).

Lines 6 and 7 configure the provider; this component of the search processor is expressed using an anonymous function. Lines 10 and 11 configure the query engine (*e.g.*, filter productions based on their names using regular expressions). The query engine is responsible for filtering the elements returned by the provider based on the user supplied query. Query engines are modeled as classes that extend a predefined class (*i.e.*, GTFilter). Currently SPOTTER provides filters for matching elements with a user query based on: substrings matching, regular expressions, and approximate matching using thresholds; currently these filters do not support indexing of search results. The filter includes the preprocessor.

When the numbers of items that need to be filtered is large, separating the provider from the query engine can cause performance problems. To account for this, SPOTTER also supports searches based on streams that combine the provider and the query engine. For example, lines 16–24 show the processor for searching through files, which uses the API method *filter:item:* that provides stream-based searching for a filter. Currently 13 processors use this optimization.

```
14  spotterForFilesFor: aStep
15          <spotterOrder: 40>

16      aStep listProcessor
17          title: 'Files';
18          itemFilterName: [:aReference| aReference basename ];
19          filter: GTFilterFileReference item: [ :filter :context |
20              self
21                  fileReferencesBy: #files
22                  inContext: context
23                  usingFilter: filter ];
24          when: [ :aReference | aReference isDirectory ];
```

A processor is attached to an object by defining in the class of the object a method constructing the processor that has the annotation *spotterOrder* (lines 2 and 15). The annotation parameter is used to sort processors in a search step. Processors are added to tags using the annotation *spotterTag:* (*e.g.*, in line 3 the *Productions* processor is added to the tag *PetitParser*). As Pharo supports extension methods processors can also be packaged and loaded separately.

An exploration session is modeled as a linked list of step objects. The code of a processor can access the exploration session using the method parameter of its containing method (aStep, line 25). This parameter gives access to the current step object. Developers can use it to access previous steps. For example, Figure 3b contains a search processor named *Shapes* that is only available when the previous step contains a Roassal visualization. If this is the case the processor ex-

[4] The version of SPOTTER discussed in this paper is available at the following url: `http://scg.unibe.ch/download/moldablespotter/spotteronward2016.zip`

tracts all graphical shapes from that visualization that render the object loaded in the current step. In Figure 3b we can see that three graphical shapes render the current object. To check if the previous object is a Roassal visualization the processor uses an activation predicate (lines 36-38): using the current step object the activation predicate can access the previous step, if present. The provider can also access the previous objects (*i.e.*, the Roassal visualization) to extract the relevant graphical shapes.

```
25  spotterForRenderingShapesFor: aStep
26      <spotterOrder: 5>

28      aStep listProcessor
29          title: 'Shapes';
30          candidatesLimit: 5;
31          allCandidates: [ aStep previousStep origin
32              elements select: [ :each | each model = self ] ];
33          itemName: [ :each | each gtDisplayString ];
34          filter: GTFilterSubstring;
35          wantsToDisplayOnEmptyQuery: true ];
36          when: [
37              aStep hasPreviousStep and: [
38                  aStep previousStep origin isKindOf: RTView ] ]
```

6. The Cost of Custom Search Processors

SPOTTER enables developers to improve information foraging loops through domain-specific searches. Nevertheless, this approach comes with a price as custom searches need to be created by application or framework developers rather than by tool providers. This can make considerable economical sense especially for long-lived, widely-used libraries that may have a large amount of user code programmed against them. This activity can also make sense for application developers that want to address recurring problems and improve that way they navigate through their particular systems. For both views to be practical, the cost associated with creating custom searches should be small. To investigate the cost of creating a custom search we start by analyzing the current extensions from the Pharo IDE and perform a user study with software developers and PhD students.

6.1 Analyzing Existing Extensions

Based on 124 search processors currently present in the Pharo IDE, the cost of creating a processor in lines of code is $9.2 \pm 4.2 (M \pm SD)$. This includes the entire source code of the method defining a processor, as well as the source code of helper methods created together with the processor; this does not include methods called from within the processor that existed in the target class before adding the processor. As it can be seen in Figure 5, a large number of extensions require 7 or 8 lines of code. These extensions follow the same pattern as the one from lines 4–13, where the class already provides a way to get the required data and the extension just uses available methods and overrides default values from the processor. When this is not the case, the size of an extension is significantly larger.

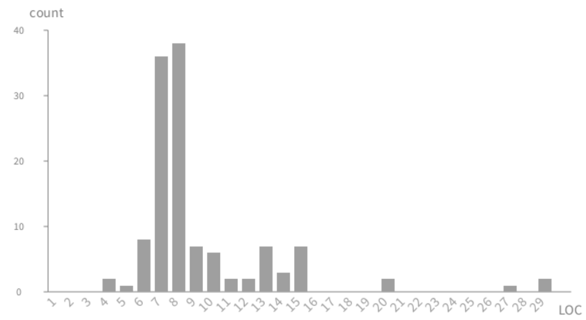


Figure 5. Size distribution in lines of code (LOC) for 124 search processors.

While lines of code do not indicate the effort needed to write those lines, they give a good indication of the small size of these extensions. This shows that even when the object model does not provides methods to access relevant data, creating a custom search processor is an affordable activity. The low cost of creating a search processor addresses the first requirement discussed in Section 2.1.

6.2 User Study Design

To investigate the creation of custom searches for SPOTTER in more details, we designed and performed a pilot user study. Our main goal was to test the extension mechanism and not the knowledge of a particular application domain or the usability of SPOTTER. Towards this goal we decided to focus on simple domain models. In doing this we assume that developers working on an application know their domain model well. Through the evaluation of this study we aimed to better understand how developers behave when creating custom searches. Hence, we structured the evaluation based on the following research questions:

RQ1 How much time does it take to create a custom extension for developers that did not extend SPOTTER before?

RQ2 Does previous experience in extending SPOTTER reduces the effort required to create a new extension?

RQ3 How do developers approach the task of creating a custom extension for SPOTTER?

The user study consisted in implementing two custom searches for SPOTTER detailed below. We selected these searches as, while covering small domain models, they give a chance for the creation of relevant search extensions.

Task 1 – Help Topics *Extend Spotter with a custom global extension for searching through all help topics from the system by their title using a textual search.* Searches through documentation and other textual data are common tasks during development. With this task we aimed to see if developers can create such searches using SPOTTER. We selected the Pharo help as a data source for the search, given that it is a familiar data source for Pharo developers. While searches through help topics are not themselves novel, as they are a common presence in IDEs, they are well understood and provide a good baseline for testing if developers

		Extension implementation	Model understanding	Task clarification	Total time	Extension size
P1	Task 1	12m30s (89%)	1m30s (11%)	0 (0%)	14m00s	6 LOC
	Task 2	7m30s (63%)	4m30s (37%)	0	12m00s	7 LOC
P2	Task 1	22m30s (94%)	0	1m30s (6%)	24m00s	6 LOC
	Task 2	5m00s (59%)	3m30s (41%)	0	8m30s	6 LOC
P3	Task 1	17m00s (83%)	3m:00s (14%)	0m30s (3%)	21m00s	11 LOC
	Task 2	6m00s (80%)	1m30s (20%)	0	7m30s	10 LOC
P4	Task 1	16m00s (86%)	1m30s (5%)	1m00s (9%)	18m30s	11 LOC
	Task 2	11m30s (70%)	4m00s (24%)	1m00s (6%)	16m30s	12 LOC
P5	Task 1	8m00s (66%)	4m00s (34%)	0	12m00s	9 LOC
	Task 2	2m30s (62%)	1m30s (38%)	0	4m00s	7 LOC
P6	Task 1	8m00s (88%)	1m00s (12%)	0	9m00s	9 LOC
	Task 2	7m00s (82%)	1m30s (18%)	0	8m30s	10 LOC

Table 3. User study results for each participant and task.

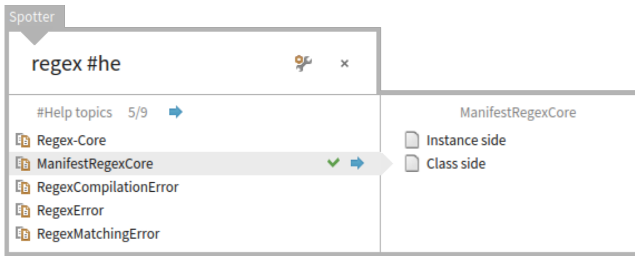


Figure 6. Searching for help topics related to regular expressions using a custom processor for searching through help topics. Each HelpTopic object has a custom preview showing its content.

understand the extension mechanism from SPOTTER. In the case of Pharo the entire help is organized as a tree of HelpTopic objects. A HelpTopic has a title, a content and a list of children topics. Participants needed to create a global search extension that extracted all available help topics, filter them using one of the available filters based on their title and open a selected help topic in the Pharo Help Browser. Our default implementation of this extension is presented in lines 39–49. A real-world search using this extension is shown in Figure 6.

```
39  spotterForHelpTopicFor: aStep
40      <spotterOrder: 200>

41      aStep listProcessor
42          title: 'Help topics';
43          allCandidates: [ SystemHelp asHelpTopic allSubtopics ];
44          candidatesLimit: 5;
45          itemName: [:helpTopic | helpTopic title ];
46          itemIcon: [:helpTopic | helpTopic topicIcon ];
47          actLogic: [:helpTopic | HelpBrowser openOn: helpTopic];
48          filter: GTFilterSubstring;
49          wantsToDisplayOnEmptyQuery: true
```

Task 2 - Morph Shortcuts *Extend Spotter with a custom textual search for searching through the shortcuts of a morph.* Morphs are the main graphical components from Pharo. A morph object uses a dispatcher object to store its shortcuts. The dispatcher object is stored in a dictionary together with other properties of a morph. A dispatcher object stores shortcuts as a set of keymap objects. Participants needed to create an extension that extracted all shortcuts from the dispatcher and supported a search based on the text of the shortcut (*e.g.*, 'ctrl+shift+s'). Unlike the previous extension, this is a specialized one that, to our knowledge, is not present in other development tools and IDEs. What we find in other IDEs are global searches through documentation that also take shortcuts into account. We selected this extension to observe how developers apply SPOTTER in a domain-specific context. Lines 50–58 illustrate our default implementation of this extension. Figure 7 shows a search involving this extension.

```
50  spotterForKeysFor: aStep
51      <spotterOrder: 15>

52      aStep listProcessor
53          title: 'Keys';
54          allCandidates: [ self kmDispatcher
55              allKeymaps asOrderedCollection ];
56          candidatesLimit: 5;
57          itemName:[:aKeymap| aKeymap shortcut printString];
58          filter: GTFilterSubstrings
```

The study had three parts. In the first part we gave participants a short introduction about how to use SPOTTER. The second part consisted in solving the two tasks and the third part in a survey for collecting impressions about the study and background data. During the introduction phase we only showed participants how to attach a custom search to a domain object. We did not go into any other details about how to extend SPOTTER. Instead we pointed participants to three places where they could get more data: *(i)* the SPOTTER documentation available in Pharo explaining SPOTTER the extension process in details; *(ii)* the class GTSpotterCandidatesListProcessor containing the main API for creating custom search processors; *(iii)* a tool for

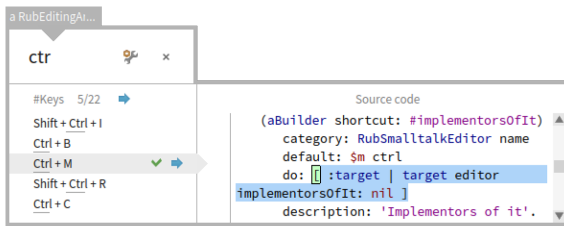


Figure 7. Processor for searching through the key bindings of a morph object. In this example the morph objects is a text editor for Pharo code, and the developer is searching for key bindings that use the *Ctrl* key. Each key binding has a custom preview that shows the code that will be executed when the key binding is pressed.

exploring all SPOTTER extensions available in the Pharo IDE.

During the second phase we captured both screen and audio recordings and asked participants to use a think-aloud strategy as they coded. For each tasks we provided participants with a description of the domain model presenting the main classes and methods. We provided these data to reduce the time needed for participants to learn the domain model. Participants first read the description of the task and of the domain and only afterwards started the coding phase. We only take the coding phase into account in our analysis. We did not impose any time limit for reading the provided material and implementing the task. A task was considered completed when the participant decided that it was good enough. The two tasks followed in immediate succession.

Six participants took part in this pilot user study. Three PhD students, having between 2 and 4 years of experience with Pharo and three software developers, having between 1 and 3 years of experience with Pharo. All knew how to use SPOTTER, however, they had no knowledge of its extension mechanism. Two developers reported interacting briefly with the domain model used in the second task in the past.

6.3 User Study Results

We first determined the correctness of the extensions. All but one participant created extensions that correctly implemented the given tasks. One participant (*P2*) did not add the functionality for opening the help topics and the shortcut objects in the required tools. In the third phase he reported that he forgot about that part of the task. We also included this participant in our analysis as the implemented parts of the extensions were correct. In analyzing the coding phase of the study we distinguished between three types of activities: *(i)* understanding and implementing a SPOTTER extension; *(ii)* understanding the domain model; *(iii)* clarifying the task. We splitted each coding session into 30 seconds intervals and assigned them to one of the aforementioned categories; we inferred the category of an interval based on what code or object the participant was looking at and the think-aloud data.

RQ1 – Effort for creating the first extension Participants took, on average, 16 minutes to implement the first task (Table 3 – *Total time*). Participants spend most of their time during this task (84%) on learning how to implement a SPOTTER extension. Participant *P5* was the exception: he decided to implement a different solution for obtaining all help topics than the one presented during the first phase. There was a large difference between participants in the completion time: participant *P2* took 24 minutes while participant *P6* took only 9 minutes. Regarding the size of an extension we obtained similar results to the discussion from Section 6.1: on average, the size of extension was 8.6 LOC (Table 3 – *Extension size*). Differences in size were due to extra features added by participants (*e.g.*, Participant *P3* added shortcuts and icons to each extension) and different ways to extract the required data. Given that this was the very first extension implemented by our participants, we consider the cost and the time to be low.

RQ2 – Effort for creating the second extension We explored our second research question by asking participants to create a second extension immediately after the first one for a more complex domain model. We noticed two changes from the first task. On the one hand, participants spend more time understanding the domain model (30% of the total time as opposed to 13% for the first task). One the other hand, the total time for solving this task improved on average by 37%; if we take only the extension implementation time into account the average individual improvement is 48%. There were nevertheless significant differences between individual participants: participant *P2* improved by 15 minutes 30 seconds while participant *P6* only by 30 seconds. In terms of lines of code, participants were consistent with the first extension. Based on this evidence we conclude that previous experience in extending SPOTTER reduces the time needed for creating new extensions.

RQ3 – Approaching the creation of extensions To understand the differences observed between participants and tasks we analyzed the strategies used by participants to implementing the given tasks. The first observation that holds for all participants is that they started to implement the required extensions by modifying an already existing extension. In the first part of the coding session (first 2-3 minutes) all participants looked through existing examples to understand the extension mechanism and selected an extension that they later modified. No participant started by reading documentation however later in the task if they got stuck, participants decided to either read documentation (*P2, P4*), browse SPOTTER classes (*P3*) or look at more examples (all participants did this to different extents).

The factor that contributed the most to high reduction in completion times between the first and the second task (participants *P2* and *P3*) was the example selected as a starting point in the first task. To exemplify, participant *P2* selected to start by modifying the global search for methods

available in Pharo. Currently this search is an optimized search using streams that combine the provider and the query engine. As discussed in Section 5, SPOTTER currently does not optimize the rapid creation of these kinds of searches. After having problems with the extension, participant *P2* chose to look at three more examples which happen to also also be searches based on streams. In the end participant *P2* moved to reading documentation and found a different way to create an extension (using the API from lines 4–13). At the opposite end participant *P6* started with an example very similar with the solution required for the first task. He adapted this example to the first task with ease.

This study revealed that the examples used by participants to learn how to extend SPOTTER had an impact on their efficiency. For this study we used all extensions currently available in the Pharo IDE as examples. These examples had no associated documentation giving insight into why they are implemented that way. These made participants chose to start from examples that were not necessarily suited for their requirements. We consider addressing these aspect by providing a collection of curated extensions, better search for examples and better documentation for extensions.

Threats to validity During the coding phase participants needed to think-aloud and were also observed by the person conducting the study; five participants knew the person conducting the study. We cannot exclude that this may have caused changes in the way they approached the tasks. The task also focused on simple domain models that many not reflect the reality of complex software applications.

7. Spotter in Practice

Section 6 explored the cost of extending SPOTTER. This gives no insight into how developer use SPOTTER in practice. To address this we collected and analyzed usage data and mailing-list discussions and performed an online survey.

7.1 Usage Data and Mailing-list Discussions

SPOTTER was integrated into the alpha version of Pharo 4 in December 2014. Six months after the initial integration we analyzed usage data recorded over a period of two months (April 2014 - May 2015) using a visual language [12]. To summarize, we noticed that developers did not discover and use the navigation features of SPOTTER to their fullest potential. For example, only half of the recorded developers used the dive-in feature at least once. Regarding search data, although we observed developers using 51 search processors, more than 74% of the time developers only used SPOTTER to search through classes and implementors of methods. One explanation for this observation is that SPOTTER exposes information that is not apparent and users need to be explicitly informed about this. We are now gathering usage data over a longer period of time to determine if developers change their behavior as they are using SPOTTER.

In the same period we also gathered feedback from discussion on several Pharo mailing-lists. Most feedback gathered from mailing-lists is related to the discoverability of features in the UI, as SPOTTER proposes a UI different from other tools in the Pharo IDE. Based on this feedback we added the possibility to select processors directly from the query (*e.g.*, entering the *"spotter #pragma"* in Figure 2c only shows the *Pragmas* processor; # filters processors by name) and selection of common processors using keyboard shortcuts.

7.2 Survey

To further explore how developers perceive and use SPOTTER, one year the release of Pharo 4, we performed an online survey during March 2016. We advertised the survey on Pharo related mailing-lists and collected 35 answers from software developers (17 responses – 48.6%), software researches (12 responses – 34.3%), students (5 responses – 14.3%) and others (1 response – 2.8%). Regarding their programming experience with object-oriented languages, 5 respondents (14.3%) reported between 1 and 3 years, 10 respondents (28.6%) between 4 and 10 years and 20 respondents (57.1%) more than 10 years. Respondents used SPOTTER until the survey during different lengths of time: 5 respondents (14.3%) less than 3 months, 20 respondents (57.1%) between 4 and 12 months and 10 respondents (28.6%) more than 12 months. When asked how often they used SPOTTER 25 respondents (71.4%) answered that during development they use SPOTTER at least several times a day; 9 respondents (25.7%) only used SPOTTER sometimes; one respondent stopped using SPOTTER. Out of the 35 respondents, 8 (22.9%) also extended SPOTTER with a custom search until the survey.

Next, respondents were asked to rate how useful they find SPOTTER as well as four individual features using a six point scale ranging from *very useful* to *very irrelevant*. The four features were: *dive-in a search result* (*FT1*), *show all results* (*FT2*), *filter search processor using #* (*FT3*) and *preview for the selected element* (*FT4*). For each individual feature, respondents could also indicate that they do not knew about that feature. Table 4 shows an overview of the results. The vast majority of respondents found SPOTTER to be at least sometimes useful: 40% very useful, 34.3% useful and 22.9% sometimes useful. Regarding the individual features, for each there were respondents that did not know about that feature. Most respondents did not know about *FT2, FT3 and FT4*. This indicates the need to further improve the user interface of SPOTTER to help users discover features. Answers for the individual features followed a similar pattern: most participants found them to be at least sometimes useful, with a few finding them irrelevant.

We also asked respondents to rate the following statements on a 5-point Likert scale:

Spotter reduced the number of tools I used for searching in the Pharo IDE. This statement received the follow-

	Very useful	Useful	Sometimes useful	Sometimes irrelevant	Irrelevant	Very irrelevant	Unknown feature
FT1	7 (20%)	7 (20%)	13 (37.2%)	2 (5.7%)	1 (2.8%)	1 (2.8%)	4 (11.4%)
FT2	9 (25.7%)	9 (25.7%)	5 (14.3%)	3 (8.6%)	0 (0%)	1 (2.8%)	8 (22.9%)
FT3	8 (22.9%)	8 (22.9%)	5 (14.3%)	2 (5.7%)	0 (0%)	3 (8.6%)	9 (25.7%)
FT4	7 (20%)	6 (17.4%)	9 (25.7%)	0 (0%)	1 (2.8%)	1 (2.8%)	11 (31.4%)
SPOTTER	14 (40%)	12 (34.3%)	8 (22.9%)	0 (0%)	0 (0%)	1 (2.8%)	—

Table 4. Survey results related to how respondents perceive SPOTTER and its features.

ing responses: 2 (5.7%) strongly agree, 13 (37.2%) agree, 7 (20%) neutral, 11 (31.4%) disagree, 2 (5.7%) strongly disagree. This indicates that some respondents use SPOTTER alongside the other search tools from the IDE.

Spotter reduced the time I need to perform searches in the Pharo IDE. This statement received the following responses: 10 (28.6%) strongly agree, 12 (34.3%) agree, 8 (22.9%) neutral, 3 (8.6%) disagree, 2 (5.7%) strongly disagree. We observe that more respondents agree than in the case of the previous statement. This suggests that respondents perform queries faster using SPOTTER than with other search tools from the Pharo IDE. This aspect still needs to be further investigated.

Threats to validity This survey is prone to both internal and external threads to validity. Respondents could chose to remain anonymous; 17 respondents (48.6%) did so by choosing to not provide an email address; this was observed mostly in respondents giving negative feedback. The survey also had a modest number of respondents.

8. Discussion

8.1 A Taxonomy of SPOTTER Searches

In Table 5 we classify all 124 search processors whose costs were analyzed in Section 6.1, based on the type of searched data. Search processors in the *Global* category correspond to global searches through code entities and files/folders mentioned in Table 2. Search processors from the category *Code entities* address developers questions from the *Packages, classes and methods* group from Table 2. *Methods (containment)* groups searches through methods contained by a class (*e.g.*, instance methods, class/static methods, methods from subclasses); *Methods (relations)* covers searches involving relations between methods (*e.g.*, callers/callees). Code critiques are warnings about code returned by the Quality Assistant tool from Pharo. The *IDE* category contains searches though data more related to the IDE like: settings, help topics, plug-ins or external projects that can be loaded into the IDE, and URLs to repositories. Search processors in the *Project* category cover searches for the Metacello package management system and the Monticello versioning system.

Domain objects groups searches through specific objects like graphical widgets, parsers, XML documents, files, *etc.* *Dynamic* in the *Other* category covers searches where the results are generated from the query string; this includes a calculator for arithmetic expressions and a processor that given an URL pointing to a Shared Workspace[5] loads the code stored in that workspace. *Extensions* offer developers the possibility of searching through all SPOTTER processors as well as domain-specific extensions for other tools.

As indicated in Section 5, developers attach extensions to an object's class. 64 extensions are attached to objects that represent code entities or to the object that models the current workspace (*e.g.*, all global processors). The remaining 60 extensions are attached to 32 different types of objects. On average an object has $1.85 \pm 1.5 (M \pm ST)$ search processors.

8.2 Spotter in Other Languages

Currently SPOTTER is developed in Pharo for the Pharo IDE and uses several querying facilities present in the Pharo IDE related to code and objects. This includes the ability to directly query code objects for relations (*e.g.*, ask for the methods of a class or for the subclasses of a class) and any object for its attributes. While this simplifies the implementation of SPOTTER there is no conceptual limitation that ties Spotter to Pharo and prevents its implementation in other IDEs for object-oriented languages. For example, to integrate SPOTTER within IntelliJ, one can start from the *Global Search* tool that already provides the possibility to execute multiple searches in parallel and extend it with the notion of search context and search session. The AST model can then be used to implement queries over code entities. The lack of extension methods in Java would require a different mechanism for associating processors with an object's class, such as aspects.

8.3 Open Questions

While initial feedback from software developers indicates that SPOTTER offers an improvement over existing search tools, empirical evidence to support this claim is missing. This aspect can be address by pursuing the following research question: *Does an IDE providing search support based on* SPOTTER *increase the efficiency of developers when performing development and maintenance tasks?* Furthermore, there already exists a wide range of exploration tools focusing on allowing developers to navigate and extract information from source code targeting various parts of the development process. Better understanding how SPOTTER

[5] http://ws.stfx.eu/

Category	Data	Count
Global	Packages	1
	Classes	1
	Annotations	1
	Methods	5
	Global variables	1
	Files/folders	2
Code entities	Packages	5
	Classes	10
	Traits	4
	Annotations	1
	Methods (containment)	9
	Methods (relations)	7
	Attributes/variables	8
	Code critiques (QA)	3
IDE	Settings	2
	Help	2
	Menus	3
	Plug-ins/Projects (Catalog)	1
	Repositories	3
Project	Configurations (Metacello)	6
	Versioning (Monticello)	7
Domain objects	Collection objects	3
	Graphical objects	5
	XML objects	2
	Examples	6
	Parser objects	3
	Bytecode	1
	Moose models [16]	6
	Files	5
Other	Code/text	3
	Extensions	2
	History	4
	Dynamic	2

Table 5. Search processors grouped based on the type of searched data. *Count* indicates the number of search processors for a data type.

compares to them and for what kind of tasks is SPOTTER better suited is also a track we are actively pursuing.

9. Conclusions

Domain concepts play an important role in program comprehension. Relying only on generic search tools during information foraging loops requires developers to focus on locating domain concepts instead of reasoning in terms of those concepts. This can be addressed if search tools enable developers to directly search through domain concepts. To support this, we proposed SPOTTER, a moldable framework that allows developers to inexpensively incorporate domain concepts in the search process as well as discover searches applicable for their own contexts. Our current prototype, developed for the Pharo IDE, shows that this approach is feasible by relying on a uniform object-oriented model and an internal DSL for expressing searches. A pilot user study shows that indeed that cost of creating custom extension is small.

SPOTTER is part of our ongoing effort into making customization of IDEs an inexpensive and practical process. We previously showed that through inexpensive customizations developers can improve that way they debug [3] and visualize objects [4]. With SPOTTER we have showed that the search process in an IDE can also be improved if the search framework puts customization into the foreground.

Acknowledgment

We gratefully acknowledge the financial support of the Swiss National Science Foundation for the project “Agile Software Analysis” (SNSF project No. 200020-162352, Jan 1, 2016 - Dec. 30, 2018). Juraj Kubelka is supported by a Ph.D. scholarship from CONICYT, Chile. CONICYT-PCHA/Doctorado Nacional/2013-63130188.

References

[1] V. P. Araya, A. Bergel, D. Cassou, S. Ducasse, and J. Laval. Agile visualization with Roassal. In *Deep Into Pharo*, pages 209–239. Square Bracket Associates, Sept. 2013.

[2] F. Beck, B. Dit, J. Velasco-Madden, D. Weiskopf, and D. Poshyvanyk. Rethinking user interfaces for feature location. In *Proceedings of the 2015 IEEE 23rd International Conference on Program Comprehension*, ICPC ’15, pages 151–162, Piscataway, NJ, USA, 2015. IEEE Press.

[3] A. Chiş, M. Denker, T. Gîrba, and O. Nierstrasz. Practical domain-specific debuggers using the Moldable Debugger framework. *Computer Languages, Systems & Structures*, 44, Part A:89–113, 2015. Special issue on the 6th and 7th International Conference on Software Language Engineering (SLE 2013 and SLE 2014).

[4] A. Chiş, T. Gîrba, O. Nierstrasz, and A. Syrel. The Moldable Inspector. In *Proceedings of the 2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! 2015, pages 44–60, New York, NY, USA, 2015. ACM.

[5] B. de Alwis and G. C. Murphy. Answering conceptual queries with Ferret. In *Proceedings of the 30th International Conference on Software Engineering (ICSE)*, pages 21–30, New York, NY, USA, 2008. ACM.

[6] B. Dit, M. Revelle, M. Gethers, and D. Poshyvanyk. Feature location in source code: a taxonomy and survey. *Journal of Software: Evolution and Process*, pages n/a–n/a, 2012.

[7] M. Eichberg and T. Schäfer. Xirc: Cross-artifact information retrieval [gpce]. In *Companion to the 19th Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA ’04, pages 43–44, New York, NY, USA, 2004. ACM.

[8] D. S. Eisenberg, J. Stylos, and B. A. Myers. Apatite: A new interface for exploring APIs. In *Proceedings of the SIGCHI*

Conference on Human Factors in Computing Systems, CHI '10, pages 1331–1334, New York, NY, USA, 2010. ACM.

[9] T. Fritz, D. C. Shepherd, K. Kevic, W. Snipes, and C. Bräunlich. Developers' code context models for change tasks. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 7–18, New York, NY, USA, 2014. ACM.

[10] D. Janzen and K. de Volder. Navigating and querying code without getting lost. In *AOSD'03: Proceedings of the 2nd International Conference on Aspect-oriented Software Development*, pages 178–187, New York, NY, USA, 2003. ACM.

[11] A. Ko, B. Myers, M. Coblenz, and H. Aung. An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks. *Software Engineering, IEEE Transactions on*, 32(12):971 –987, Dec. 2006.

[12] J. Kubelka, A. Bergel, A. Chiş, T. Gîrba, S. Reichhart, R. Robbes, and A. Syrel. On understanding how developers use the Spotter search tool. In *Proceedings of 3rd IEEE Working Conference on Software Visualization - New Ideas and Emerging Results*, VISSOFT-NIER'15, pages 145–149. IEEE, Sept. 2015.

[13] D. C. Littman, J. Pinto, S. Letovsky, and E. Soloway. Mental models and software maintenance. *J. Syst. Softw.*, 7(4):341–355, Dec. 1987.

[14] G. C. Murphy, M. Kersten, M. P. Robillard, and D. Čubranić. The emergent structure of development tasks. In *Proceedings of the 19th European Conference on Object-Oriented Programming*, ECOOP'05, pages 33–48, Berlin, Heidelberg, 2005. Springer-Verlag.

[15] E. Murphy-Hill, R. Jiresal, and G. C. Murphy. Improving software developers' fluency by recommending development environment commands. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, pages 42:1–42:11, New York, NY, USA, 2012. ACM.

[16] O. Nierstrasz, S. Ducasse, and T. Gîrba. The story of Moose: an agile reengineering environment. In *Proceedings of the European Software Engineering Conference (ESEC/FSE'05)*, pages 1–10, New York, NY, USA, Sept. 2005. ACM Press. Invited paper.

[17] M. Petrenko, V. Rajlich, and R. Vanciu. Partial domain comprehension in software evolution and maintenance. In *The 16th IEEE Int'l Conf. on Program Comprehension*, pages 13–22. IEEE, June 2008.

[18] P. Pirolli and S. Card. The sensemaking process and leverage points for analyst technology as identified through cognitive task analysis. In *Proceedings of International Conference on Intelligence Analysis*, pages 2–4, 2005.

[19] V. Rajlich and N. Wilde. The role of concepts in program comprehension. In *Proceedings of the 10th International Workshop on Program Comprehension*, IWPC '02, pages 271–, Washington, DC, USA, 2002. IEEE Computer Society.

[20] L. Renggli, S. Ducasse, T. Gîrba, and O. Nierstrasz. Practical dynamic grammars for dynamic languages. In *4th Workshop on Dynamic Languages and Applications (DYLA 2010)*, pages 1–4, Malaga, Spain, June 2010.

[21] M. P. Robillard, W. Coelho, and G. C. Murphy. How effective developers investigate source code: An exploratory study. *IEEE Trans. Softw. Eng.*, 30(12):889–903, Dec. 2004.

[22] S. Robitaille, R. Schauer, and R. Keller. Bridging program comprehension tools by design navigation. In *Software Maintenance, 2000. Proceedings. International Conference on*, pages 22–32, 2000.

[23] T. Schafer, M. Eichberg, M. Haupt, and M. Mezini. The SEXTANT software exploration tool. *IEEE Trans. Softw. Eng.*, 32(9):753–768, Sept. 2006.

[24] D. Shepherd, K. Damevski, B. Ropski, and T. Fritz. Sando: An extensible local code search framework. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, pages 15:1–15:2, New York, NY, USA, 2012. ACM.

[25] J. Sillito, G. C. Murphy, and K. De Volder. Asking and answering questions during a programming change task. *IEEE Trans. Softw. Eng.*, 34:434–451, July 2008.

[26] B. Spasojević, M. Lungu, and O. Nierstrasz. Overthrowing the tyranny of alphabetical ordering in documentation systems. In *2014 IEEE International Conference on Software Maintenance and Evolution (ERA Track)*, pages 511–515, Sept. 2014.

[27] J. Starke, C. Luce, and J. Sillito. Searching and skimming: An exploratory study. In *Software Maintenance, 2009. ICSM 2009. IEEE International Conference on*, pages 157–166, Sept. 2009.