Building Ecosystem-Aware Tools Using the Ecosystem Monitoring Framework

Boris Spasojević
University of Bern, Switzerland
spasojev@inf.unibe.ch

Abstract

Integrating ecosystem data into developer tools can be very beneficial but is usually complicated. By automating the routine parts of this task we can reduce the amount of work needed to develop these tools. We have developed a framework that allows developers to quickly develop new tools that use ecosystem data. This framework automates the execution of user-defined analyses on ecosystem projects, allowing the developer to focus only on what ecosystem data is needed for her tool and how to present it.

1 Introduction

Leveraging ecosystem data in the development process through ecosystem-aware developer tools can benefit software quality and speed of development in many ways. A large body of work has been done in integrating ecosystem data *i.e.*, data from other related projects, into the development process. Examples of such work deal with code completion [2], method and argument recommendations [1], library scoring systems [5], code snippet recommenders [13] and others.

Papers published in this field rarely take into account software evolution. They focus heavily on the usability of their respective tools and provide no support once the data it is based on is out of date. We argue that tool developers are in need of tool support to gather ecosystem data and keep it up to date.

We have implemented one such tool for the Pharo Smalltalk ecosystem and named it "Ecosystem Monitoring Framework" or EMF for short. The main goal of this framework is to enable developers to write tools that leverage ecosystem data (ecosystem-aware tools) without requiring them to spend time on the infrastructure for running their analyses on the ecosystem projects and keeping that data fresh. As shown in Figure 1, EMF is implemented as a client-server application. The server side is in charge of gathering, storing and providing the ecosystem data used by the clients — the ecosystem-aware tools. The server side periodically loads all the source code of all the projects, executes the user-defined analysis on all loaded source code and stores the results in a database to be provided to clients on demand. We use off the shelf modules for the database (MongoDB) and data provider (REST server) and implement the loading of the source code and running the analyses in bash and Pharo Smalltalk. Section 2 gives a short description of the Pharo ecosystem we used and Section 3 gives a description of how to implement a simple tool using EMF.

Using this framework, we augmented 3 existing Pharo tools with ecosystem data: a type inference engine [9], a system browser [11, 10] and a type guessing system [12]. We also developed a system that stores object-producing

Proceedings of the Seminar Series on Advanced Techniques and Tools for Software Evolution SATToSE2016 (sattose.org), Bergen, Norway, 11-13 July 2016, published at http://ceur-ws.org

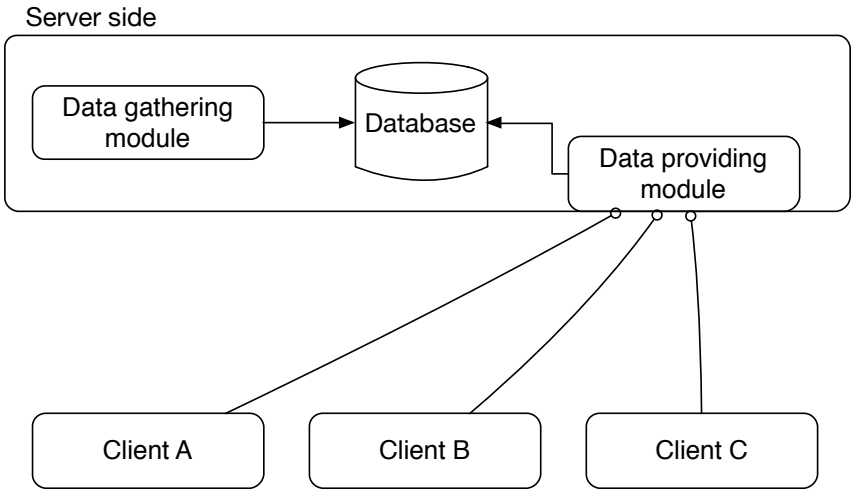

Figure 1: The architecture of EMF.

snippets, enabling another level of augmentation for tools that require objects on demand. All of these tools are described in Section 4.

Finally, we discuss the limitations of the framework as well as interesting directions for future work and improvements in Section 5.

2 The Pharo Ecosystem

We call a group of software systems united by common and mutual dependencies a Software Ecosystem. This is inline with previous definitions of software ecosystems, most notably the one by Lungu *et al.*: "A software ecosystem is a collection of software projects that are developed and evolve together in the same environment." [4]

In this section we present a short description of the Pharo ecosystem as defined in the configuration browser. The configuration browser is a tool for automated loading of Pharo projects similar to Maven[1] for java. It reads its list of project from a human maintained meta repository[2] containing scripts that automatically load projects and their dependencies. Many of these dependencies are also defined in the same meta repository, supporting the claim that these projects co-evolve, and that changes in one project affect others throughout the ecosystem. In Figure 2 we can see the dependencies (represented as edges in the graph) between projects (represented as the nodes) from the configuration browser. This graph does not show the dependency to the classes from the base image which all the projects have. It is clear from the dense network of edges in the graph that these projects are very interdependent.

To discuss the size of the ecosystem, we present Figure 3. In this figure we can see the number of projects in the ecosystem on the first of each month in the interval between 01.01.2014 and 01.08.2016. We can see that there is linear growth of the number of projects until mid 2015 after which the number stabilises around 160 projects. This illustrates that this ecosystem evolves not just in the content of individual projects, but also in its scope, supporting further the need for providing the ecosystem-aware tools with fresh ecosystem data. It also illustrates the advantage of using a meta repository as the source of the ecosystem projects rather than using a fixed set of projects.

3 Implementing an Ecosystem-aware Tool Using EMF

To better understand the process of developing ecosystem-aware tools we discuss how to implement a simple tool we call "Class name clash prevention tool".

Pharo Smalltalk does not have a concept of namespace. This means that different projects could define a class with the same name, which could cause problems if such project needed to co-exist in the same image. We call

[1] `https://maven.apache.org/`
[2] `http://smalltalkhub.com/mc/Pharo/MetaRepoForPharo30/main/`

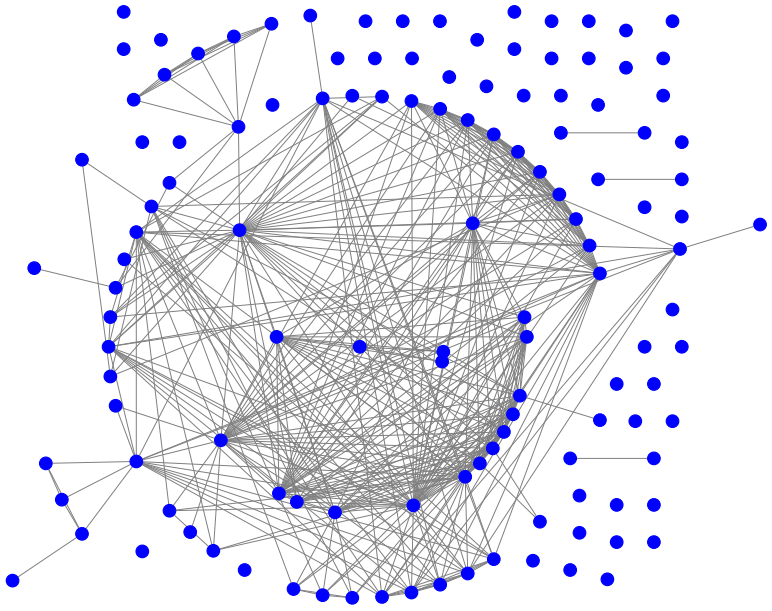

Figure 2: Dependencies between projects in the Pharo Configuration browser.

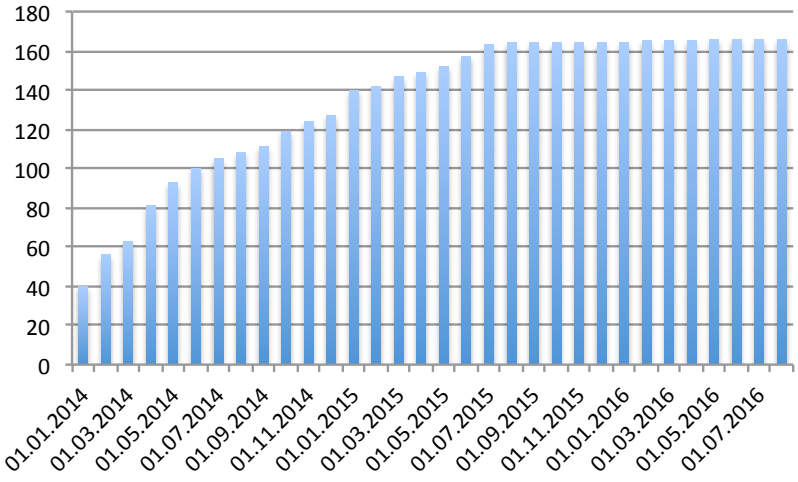

Figure 3: The growth of the Pharo ecosystem between 01.01.2014. and 01.08.2016.

```
1  ClassClashDataGatherer>>emfAnalysis
2      | toBeStoredToDb |
3      toBeStoredToDb := OrderedCollection new.
4      EMFAnalysisCore projectClasses
5          do: [ :pc |
6          | classToProject |
7          classToProject := Dictionary new
8              at: 'className' put: pc name;
9              at: 'project' put: EMFAnalysisCore projectName;
10             yourself.
11         toBeStoredToDb add: classToProject . ].
12     EMFAnalysisCore
13         save: toBeStoredToDbCollection
14         toDB: 'classClash'
15         inCollection: 'classClash' , EMFAnalysisCore executionID
```

Listing 1: Implementation of the class name clash prevention tool back end.

this a class name clash. Class name clash prevention tool[3] is an ecosystem-aware tool that informs a developer if a newly created class shares a name with another class in the ecosystem, thus preventing later class name clashes.

All ecosystem-aware tools that leverage EMF are developed in 3 steps: developing the back end, registering the back end with EMF and developing the front end. At no point is the developer concerned about the scope or structure of the ecosystem, as this is defined in advance as part of EMF. The following subsections describe each of the steps in general and the case of the Class Clash prevention tool.

3.1 Step 1: Developing the Back End

The role of the back end part of a ecosystem-aware tool is to gather relevant data from the ecosystem and store it for later access by a front end. In the case of the class name clash prevention tool relevant data consists of class names from all ecosystem projects mapped to the name of the project that defines it.

EMF offers an API to obtain a list of projects classes and the project name, so creating such a mapping is trivial. Since the default database in EMF is MongoDB we store this data as JSON documents represented in Pharo Smalltalk as Dictionary objects. EMF provides an API for storing the data to MongoDB. All the functionality of the class name clash prevention tool back end is wrapped in the `emfAnalysis` method of the `ClassClashDataGatherer` class of which source code is shown in Listing 1. This method iterates over all classes of the project (lines 4–5) and creates a new entry for the database (lines 7–11) which encodes the relation between the class name and the project name. Finally, the entries are stored to the database at the end (line 12).

3.2 Step 2: Registering the Back End with EMF

EMF uses an XML configuration file to describe the steps needed to perform an analysis on a project. The configuration file for the class name clash prevention tool is shown in Listing 2. This configuration contains only one target (lines 2–16) as it is used only for the class name clash prevention tool. The XML schema allows multiple targets to be specified, one for each analysis. This target has a "pre" step (lines 4–10), specifying the Gofer[4] script that loads the source code of the back end of the tool described in Subsection 3.1 and an "analysis" step (lines 12–15) specifying how to run the loaded back end code by invoking the `emfAnalysis` method on an instance of `ClassClashDataGatherer`. Since no post processing is needed for this tool, the "post" step is absent from the configuration.

Once this configuration file has been provided to the EMF, all the specified steps from the configuration file, and thus all the analyses, will be run once a week to gather fresh data from the ecosystem.

[3] `http://smalltalkhub.com/#!/~spasojev/ClassClash`
`http://smalltalkhub.com/#!/~spasojev/ClassClashFrontEnd`

[4] `http://pharobooks.gforge.inria.fr/PharoByExampleTwo-Eng/latest/Gofer.pdf`

```
<config>
  <target>
    <pre>
      <st>
        Gofer new
          url: 'http://smalltalkhub.com/mc/spasojev/ClassClash/main/';
          package: 'ConfigurationOfClassClashBackEnd'; load.
        (Smalltalk at: #ConfigurationOfClassClashBackEnd)
          loadDevelopment.
      </st>
    </pre>
    <analysis>
      <st>
        ClassClashDataGatherer new emfAnalysis.
      </st>
    </analysis>
  </target>
</config>
```

Listing 2: EMF configuration file for the class name clash prevention tool.

3.3 Step 3: Developing the Front End

Once the EMF data gatherer runs, and the data is gathered by the class name clash prevention tool back end, it is available for serving by the data providing module of EMF. At this point, the front end of the class name clash prevention tool can function. We implement the front end as a plugin for Nautilus[5] which is the default system browser for Pharo. The plugin reacts to a new class being created. Once that event is detected, the front end sends the name of the new class to the data providing module which returns from the database a list of all projects that contain a class with that name. This list is then presented to the developer.

4 Ecosystem-aware tools

In this section we describe the ecosystem-aware tools we developed using EMF. We developed these tools in order to demonstrate the versatility of EMF and to verify that it does satisfy the requirements of a unified framework for ecosystem-aware tools.

4.1 Ecosystem-aware type inference

Dynamically typed languages lack information about the types of variables in the source code. Developers care about this information as it supports program comprehension. Basic type inference techniques are helpful, but may yield many false positives or negatives.

In ecosystem-aware type inference we track how many times messages are sent to instances of available types throughout the source code for all available projects from the ecosystem. This means that the back end of the ecosystem-aware type inference builds a weighted mapping from types to selectors, where the weight is the number of times a message with that selector was sent to an instance of that class.

We use this information in the front end to sort the potential types of a variable the developer cares about based on their likelihood of being the actual type in the context. The likelihood is computed based on how many times the messages sent to this variable have been observed to be sent to each potential type throughout the ecosystem. Using EMF, we implemented a prototype and used it to evaluate the approach. We show that, for our implementation, measuring the frequency of association between a message and a type throughout the ecosystem source code is helpful in identifying correct types [9].

4.2 Frequently used methods

Software developers are often unsure of the exact name of the method they need to use to invoke the desired behavior in a given context. This results in a process of searching for the correct method name in documentation, which can be lengthy and distracting to the developer.

[5] `http://smalltalkhub.com/#!/~Pharo/Nautilus`

We can decrease the method search time by enhancing the documentation of a class with the most frequently used methods. Usage frequency for methods is gathered by analyzing other projects from the same ecosystem [10].

Using EMF, we implemented a proof of concept of the approach. We use the same data gathered for the ecosystem-aware type inference, but a different front end. Since methods are commonly searched for using a system browser called Nautilus, we developed a plugin for it which adds a section with the most commonly used methods for the currently observed class [11].

4.3 Ecosystem-aware type guessing

A common practice when writing Smalltalk source code is to name method arguments in a way that hints at their expected type (i.e., `aString`, `anInteger`, `aDictionary`). This practice makes code more readable and some tools (such as the auto complete feature in the Pharo Smalltalk code editor) improve the developer experience by "guessing" the type of the method argument based on these hints.

Using EMF we gather argument names throughout the ecosystem and generate a weekly report containing information about commonly used ones. This report can be used by the Type-guessing tool developer to include heuristics for better type guessing. Used these reports we developed heuristics that improved Pharo type-guesser by almost 40% [12].

4.4 Object repository

Unlike the previously described ecosystem-aware tools, which are augmentations of existing tools with ecosystem data, the Object Repository is just a back end for multiple potential front ends. The main idea behind the Object Repository is to enable front end tools to have "Objects on demand". This means that the Object Repository mines, from the ecosystem, code snippets that, when executed, produce an instance of some class. These snippets are then stored and mapped to the class they can instantiate. A front end needs only to provide a class name, and the Object Repository will provide all available snippets that instantiate that class. These snippets have many potential use cases in software documentation, testing, program comprehension *etc.*

Our proof of concept implementation of the Object Repository relies on brute force execution of code segments obtained through converting AST nodes of methods to source code. We show that applying the proposed approach to the Pharo ecosystem, as defined in EMF, results in an Object Repository that can instantiate almost 80% of the available classes in these projects [8].

5 Discussion and open questions

In order to provide fresh data to the front end tools, EMF has to periodically re-run all the analyses on new versions of source code. In our implementation we chose a one week interval, but for some tools and some quickly evolving ecosystems this might not be enough. The main challenge here is that if we wish to have completely fresh data we need to re-run the analyses on every commit to every project in the ecosystem. This is especially problematic if the projects are hosted by a third party (*e.g.*, GitHub, SmalltalkHub) which are not willing to give us notifications and full access at every commit, requiring us to poll these repositories for changes. One way this issue could be solved is if the code hosting providers offered a cloud solution for running analyses on the source code. Much like other "infrastructure as a service" solutions this should offer tool developers access to computing machines with preloaded source code of required projects, allowing the developer access to fresh data and providing a source of monetization for the hosting service.

If we manage to obtain every commit in real time there is still the issue that for every commit we need to re-analyze the entire ecosystem. Alternatively, we could express the analyses in such a way as to operate on single commits (*i.e.*, "diffs" in the source code), single class or single project. This would make them much less elegant and understandable. Further we could define a model of the ecosystem which is easier to update in real time and have the analyses be expressed in terms of that model, relying on source code only when absolutely necessary. In time, the model would have to be updated as new user needs are identified.

Another open question is how to define the scope of an ecosystem. In our implementation we relied on a manually maintained meta-repository to define our ecosystem. It would be interesting to try to find a way to express certain constraints that would define the ecosystem of interest *e.g.*, all projects using a particular library or framework. This ecosystem definition would allow to automatically extend or shrink the ecosystem scope as the individual projects evolve *e.g.*, adopt or abandon the library we are interested in.

The main challenge for adoption is that EMF is, in its current form, limited to Pharo Smalltalk. In order to port it to more popular languages such as Java we would need a concise and expressive way to analyse the

source code of Java projects. Smalltalk has the advantage that, due to its high reflectivity [7] and being partly self implementing, analyses can be written in Smalltalk itself, operating on the source code of the project in question. This is mainly not true for other languages, and we would need to use additional tools (*e.g.*, Moose [6] or Rascal [3]) to analyse the source code.

In the future, it might also be worthwhile considering porting EMF to cross language ecosystems *e.g.*, the JVM ecosystem — the ecosystem of all languages that compile to Java byte code. This raises a whole new set of questions and challenges regarding interlanguage analysis, mapping concepts from one language to another, *etc.* Many of these can be addressed by using a language agnostic meta-model (*e.g.*, Famix) but using any model reduces the amount of available information so finding the right level of abstraction is imperative.

Throughout this paper we discussed ecosystem-aware tools relying only on source code of the projects in the ecosystem. Software development today produces a wide range of different artifacts that supplement the source code and are used as additional data sources for ecosystem-aware tools. Including this additional data into our ecosystem definition and into EMF would open new possibilities for developing different kinds of ecosystem-aware tools, but also raise a series of challenges on how to obtain, analyze or keep that data fresh.

Finally, we still need a comprehensive study on user needs when it comes to ecosystem-aware tools. The tools we developed were based on our own intuition and, even though our analyses show that ecosystem data improves these tools, future work would be best served by finding exact user needs, and ensuring that EMF can support tools that tackle those problems.

With all this said, we can still conclude that EMF is an easier way of developing ecosystem-aware tools. It frees the developer from the routine parts (scoping the ecosystem, loading the projects, executing the analysis on each project, storing the resulting data and providing it to client tools) and allows her to focus on what makes her new tool unique and useful. Also, the tools we developed using it support its applicability to a wide range of problems.

Acknowledgments

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).

References

[1] M. Acharya, T. Xie, J. Pei, and J. Xu. Mining API patterns as partial orders from source code: From usage scenarios to specifications. In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC-FSE '07, pages 25–34, New York, NY, USA, 2007. ACM.

[2] M. Bruch, M. Monperrus, and M. Mezini. Learning from examples to improve code completion systems. In *Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC/FSE '09, pages 213–222, New York, NY, USA, 2009. ACM.

[3] M. Hills, P. Klint, and J. J. Vinju. Scripting a refactoring with rascal and eclipse. In *Proceedings of the Fifth Workshop on Refactoring Tools*, WRT '12, pages 40–49, New York, NY, USA, 2012. ACM.

[4] M. Lungu, M. Lanza, T. Gîrba, and R. Robbes. The Small Project Observatory: Visualizing software ecosystems. *Science of Computer Programming, Elsevier*, 75(4):264–275, Apr. 2010.

[5] Y. M. Mileva, V. Dallmeier, and A. Zeller. Mining api popularity. In *Proceedings of the 5th International Academic and Industrial Conference on Testing - Practice and Research Techniques*, TAIC PART'10, pages 173–180, Berlin, Heidelberg, 2010. Springer-Verlag.

[6] 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.

[7] F. Rivard. Smalltalk: a reflective language. In *Proceedings of REFLECTION '96*, pages 21–38, Apr. 1996.

[8] B. Spasojević, M. Ghafari, and O. Nierstrasz. The object repository, pulling objects out of the ecosystem. In *Proceedings of the 11th Edition of the International Workshop on Smalltalk Technologies*, IWST'16, pages 7:1–7:10, New York, NY, USA, 2016. ACM.

[9] B. Spasojević, M. Lungu, and O. Nierstrasz. Mining the ecosystem to improve type inference for dynamically typed languages. In *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! '14, pages 133–142, New York, NY, USA, 2014. ACM.

[10] 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.

[11] B. Spasojević, M. Lungu, and O. Nierstrasz. Towards faster method search through static ecosystem analysis. In *Proceedings of the 2014 European Conference on Software Architecture Workshops*, ECSAW '14, pages 11:1–11:6, New York, NY, USA, Aug. 2014. ACM.

[12] B. Spasojević, M. Lungu, and O. Nierstrasz. A case study on type hints in method argument names in Pharo Smalltalk projects. In *Proceedings of the 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 1, pages 283–292, Mar. 2016.

[13] S. Thummalapenta and T. Xie. Parseweb: a programmer assistant for reusing open source code on the web. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, ASE '07, pages 204–213, New York, NY, USA, 2007. ACM.