Pangea: A Workbench for Statically Analyzing Multi-Language Software Corpora

Andrea Caracciolo, Andrei Chis, Boris Spasojević, Mircea Lungu
Software Composition Group
University of Bern, Switzerland
{caracciolo, andrei, spasojev, lungu}@iam.unibe.ch

***Abstract*—Software corpora facilitate reproducibility of analyses, however, static analysis for an entire corpus still requires considerable effort, often duplicated unnecessarily by multiple users. Moreover, most corpora are designed for single languages increasing the effort for cross-language analysis. To address these aspects we propose Pangea, an infrastructure allowing fast development of static analyses on multi-language corpora. Pangea uses language-independent meta-models stored as object model snapshots that can be directly loaded into memory and queryed without any parsing overhead. To reduce the effort of performing static analyses, Pangea provides out-of-the box support for: creating and refining analyses in a dedicated environment, deploying an analysis on an entire corpus, using a runner that supports parallel execution, and exporting results in various formats. In this tool demonstration we introduce Pangea and provide several usage scenarios that illustrate how it reduces the cost of analysis.**

I. Introduction

One of the most time-consuming activities when performing static analysis is setting up a complete and properly configured executable workbench that fits the user's purposes. Consider the case of a researcher that wants to study the way Java software systems are written. Even for testing the simplest of hypotheses, she would habitually have to go through a large number of steps to set up the experimental infrastructure for a *big software data analysis* [1]:

1) Obtain a large number of representative Java systems. Software corpora are curated collections of open source software systems that facilitate reproducibility of analyses and allow the comparison of measurements.
2) Find an appropriate fact extractor that can analyze Java source code.
3) Use the fact extractor to export intermediate models, a step that requires parsing, and thus can take a significant amount of time for large systems.
4) Convert the intermediate models into a data format that supports complex and efficient queries.
5) Start developing and refining the static analysis that will verify the hypothesis.
6) Finally, deploy the analysis, collect and analyse the results.

Should she require replicating this experiment on a different programming language, she would have to redo all the previous steps once again.

Furthermore, the first four steps, need to be repeated by every other practitioner wanting to run a similar analysis.This need not be so. Once a appropriate corpus is identified, and a suitable environment for analyzing that corpus is built, there should be no need to repeat the investment in building the same environment again. This is particularly true since software corpora are by definition frozen in time.

We present Pangea, an infrastructure that eases multi-language static analysis of software corpora by providing a repository of language-independent object model snapshots. Pangea further supports automating common operations such as setting up the analysis environment, designing the analysis, extracting intermediate models, and executing a given analysis in parallel on all the projects contained in one or more corpora.

To enable analyses which span cross-language corpora, Pangea's repository is designed to host models which conform to language-independent meta-models. This choice imposes different kind of restrictions on the expressivity of definable analyses (See subsection VI-B). On the other hand, analyses can be written once and deployed on any corpus written in any language which is modeled in the repository. Researchers and practitioners can quickly test out ideas, perform comparative statistical analyses across programming languages, verify hypotheses, share analyses for reproducibility by third parties, and maybe provide a grain of objectivity in language discussions.

The remaining of this paper is structured as follows: section II shows a general overview of our approach. section III and section IV describe how Pangea can be set up and used in practice. section V discusses the current extent of the data repository. Sections VI and VII wrap up the paper by discussing the advantages and limitations of our approach and comparing it to the related work.

II. Pangea in a Nutshell

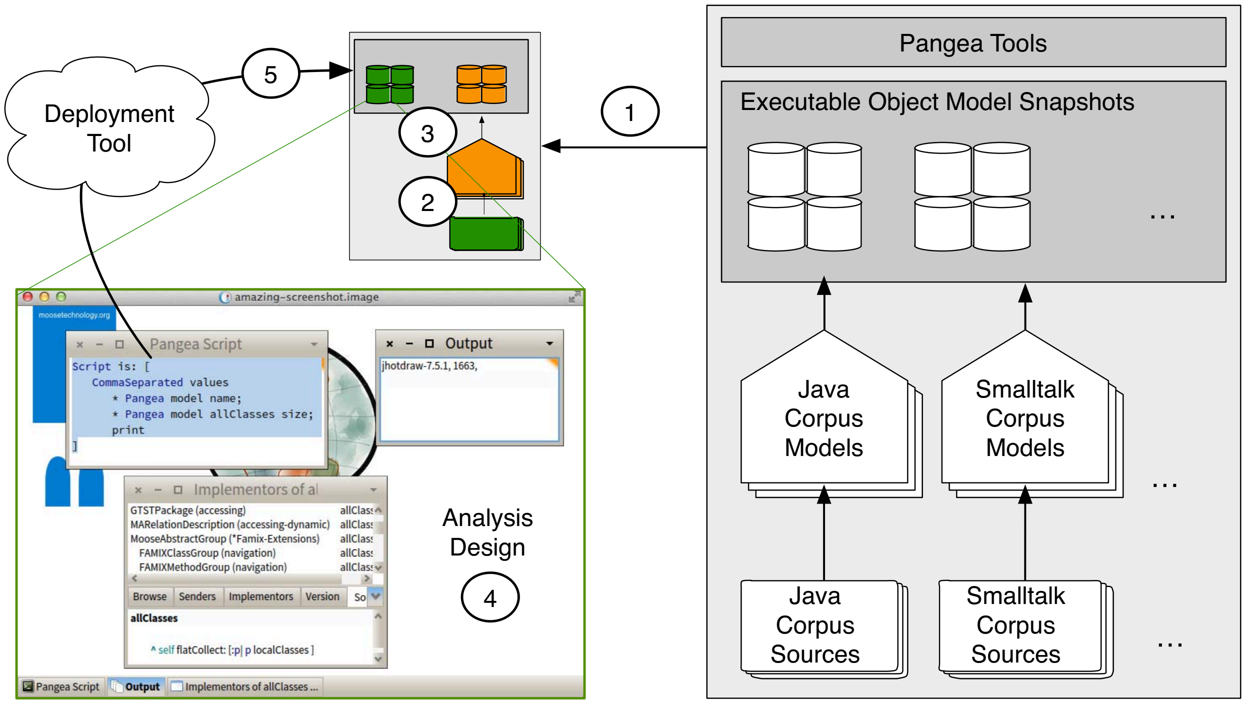
Pangea is a curated distribution of data and tools for static analysis. Next we detail its architecture, the structure of its data repository and the user workflow.

A. Pangea Architecture

Pangea stores cross-language software corpora in a centralized repository (See Figure 1).

All data is stored in centralized repository and the user can download on demand just those parts of the data that are of interest for his current needs. Analysis is then run locally.

978-1-4799-6148-1/14 $31.00 © 2014 IEEE
DOI 10.1109/SCAM.2014.39



```
Script is: [
   CommaSeparated values
      * Pangea model name;
      * Pangea model allClasses size;
      print
]
```

```
jhotdraw-7.5.1, 1663,
```

```
GTSTPackage (accessing)                     allClass
MARelationDescription (accessing-dynamic)   allClass
MooseAbstractGroup (*Famix-Extensions)      allClass
  FAMIXClassGroup (navigation)              allClass
  FAMIXMethodGroup (navigation)             allClass
```

```
allClasses

    ^ self flatCollect: [:p| p localClasses ]
```

Fig. 1. The workflow and data architecture of Pangea

B. Pangea Data

Figure 1 illustrates the three types of information that Pangea stores for each corpus:

- *Source Code.* This is the raw source code of the systems in the corpus.
- *Language-independent Models.* Pangea stores the intermediate results of various fact extractors as FAMIX-compliant models using MSE as interchange format. FAMIX is a well-known object-oriented meta-model and MSE is the default interchange format between a series of software analysis tools[2]. Models based on FAMIX are directed graphs with packages, classes, methods, and attributes as nodes. Each entity features structural properties (i.e. modifiers, signature) and metrics (i.e. LOC, NOM, cyclomatic complexity). Associations between these entities, e.g., class inheritance, method calls, attribute accesses are represented as directed edges.
- *Object Model Snapshots (OMSs).* For each system of the corpus we provide a set of object model snapshots that can be downloaded and used to efficiently execute user-defined analyses. The snapshots are modified Moose [3] images which contain a full object-oriented FAMIX interactive and visual analysis tools. The FAMIX model can be navigated and browsed to perform various source code analysis tasks that, based on an AST only, are usually complex to implement.

C. Pangea Workflow

Figure 1 sketches the main steps of the workflow:

1) Downloading the Pangea tools and the Data. By downloading OMSs, the user can avoid parsing the projects contained in a corpus and transform the extracted information into a queryable relational model (Steps 2 and 3). Figure 1 illustrates this by higlighting a hypothetical situation in which green is downloaded data and orange is generated data.
2) Extracting FAMIX models from source code. [Optional[1]]
3) Building the OMSs. [Optional]
4) Analysis Design. The user writes his scripts interactively inside a OMS. As long as her analysis can be done based on the FAMIX meta-model, she can use a series of powerful tools provided by the Moose environment.
5) Deployment. Once the analysis is designed, it can be deployed on one or more systems or corpora.

III. Setting up Pangea

A. Setting up the analysis environment

The user can download software corpora using one of the following commands:

```
pangea get src -c CorpusName
pangea get models -c CorpusName
```

The first command will download the source files associated with all the projects contained in the specified corpus. The second command is used to retrieve the same information in a more compact and executable format. By downloading OMSs, the user obtains serialized FAMIX models that can be directly expanded in memory. OMSs can also be generated locally from the sources of a given corpus using the following sequence of commands.

```
pangea make mse -c CorpusName
pangea make models -c CorpusName
```

FAMIX models can be queried using a purpose-built Smalltalk internal DSL based on Moose.

B. Evaluating an analysis algorithm

To run an analysis, the user needs to define an analysis script that will be executed against all downloaded OMSs. The script (See examples in section IV) must be designed to perform the following activities: query a FAMIX object model; perform some analysis; write the results in a file. Once defined, the analysis script needs to be saved into a file and passed as an argument to the following command:

```
pangea run script -c CorpusName script-name.st
```

This script executes the specified analysis script on every project contained in the specified software corpus. The output is printed to standard output and can be redirected to a file. The execution can be parallelized by using this other command:

```
pangea run parallel -c CorpusName script-name.st
```

The user can test his script on a single arbitrary project using this command:

```
pangea experiment -c CorpusName
```

This command will open an OMS (with Moose analysis environment) in graphical mode allowing the user to interactively build and refine his analysis routine.

IV. Analysis examples

A. Average Class Hierarchy Height

Suppose that a researcher wants to answer the following quantitative research question:

- *What is the difference between the average height of class hierarchies in Java and Smalltalk projects?*

After he has downloaded the analysis infrastructure, he writes the script in Listing VI-B which computes the required data for answering this question by outputting comma separated values of average hierarchy depth for each of the systems in a corpus[2].

```
1  Script is: [
2
3    |hierarchies totalDepth |
4    hierarchies := 0.
5    totalDepth := 0.
6
7    Pangea model allClassHierarchyRoots do:
8    [:each|
9        totalDepth := totalDepth +
10         each subclassHierarchyGroup size.
11       hierarchies := hierarchies + 1
12   ].
13
14   CommaSeparated values
15     * Pangea model name;
16     * (totalDepth / hierarchies);
17     print.
18 ].
```

Listing 1. A pangea script that outputs the average height of class hierarchies in a system

Lines 3-5 are used to initialize two counters that will be used to calculate the result.

In lines 7-12 we iterate over all the classes that are at the top of a hierarchy (*i.e.*, classes that do not have a superclass) and count the number of subclasses associated to them. The average depth of the hierarchies contained in the analyzed system is finally saved as two comma separated values (lines 14-17): the name of the system (line 14) and the numeric result of the analysis (line 15).

This analysis script has been run on the following software corpora: “QualitasCorpus-20120401r”; “Squeaksource-100”. The total execution required 1:23 minutes (73s for the first corpus and 10s for the second). By parallelizing the execution, the total time required to complete the process can be reduced to 30s.

The results presented in Figure 2 tell us that the average height of class hierarchies in Smalltalk systems is significantly larger than in Java systems.

B. Method use of Thread class

Another example question that can easily be answered using pangea is

> *What are the most commonly used methods of the* java.lang.Thread *class?*

[2]Pangea scripts use a Smalltalk-based DSL. For a reader that is not familiar with it, the Smalltalk syntax can be notoriously illustrated on the back of a postcard (http://mir.lu/st-card)

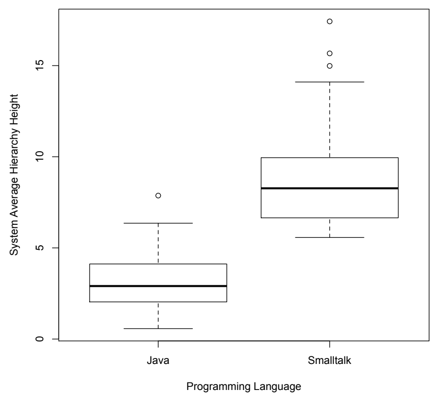


Fig. 2. Box plot showing the results obtained by collecting the average class hierarchy height of Java and Smalltalk systems.

This question cannot be answered using simple text analysis tools such as *grep*. This is because we need a lot more context about the source code *i.e.*, variable types. We need to extract the method name from all call sites where a method is invoked on an instance of class *java.lang.Thread*, and count the number of occurrences of each unique method. This is done by the script in Listing 3.

```
1  Script is: [
2
3     | invocations methods threadMethods
        threadMethodNames |
4
5     invocations := Pangea model allInvocations.
6
7     methods := invocations collect: [ :e |
8        e candidates first
9     ].
10
11    threadMethods := methods select: [ :e |
12    e parentType mooseName = #java::lang::Thread
13    ].
14
15    threadMethodNames := threadMethods
16       collect: [ :e |
17          (e mooseName subStrings: '.') second
18       ].
19
20    threadMethodNames
21       do: [ :e | Output print: e; cr ]
22 ]
```

Listing 2. A script that outputs the method name of all invocations to instances of class *java.lang.Thread*

The script begins with the script declaration (line 1) and is scoped between square brackets (line 1 and line 22). The first part of the script (line 3) declares local variables used in the script. Since method invocations are directly modeled in the FAMIX meta model, we easily get all the invocations from the model in line 5.

In lines 7 to 9 we gather all the invoked methods. Since FAMIX is language agnostic, and must support dynamically typed languages, all invocations provide a set of candidate methods. Since this script is meant to be used on a statically typed language (Java), it is safe to assume only one candidate per invocation, so we obtain the first and only candidate[3]

In lines 11 to 13 we select only the methods of the class *java.lang.Thread*. The method *parentType* invoked on a method object returns the type of the defining class (the parent), whose name we can obtain using the *mooseName* method. The name is, in the case of Java, the fully qualified name whose elements are separated with '*::*'. This is the result of the language agnostic nature of FAMIX.

Method names returned by the *mooseName* method contain both class name and method name separated by a period. In lines 15 to 18 we extract only the method name, and in lines 20 and 21 we print all the method names to standard output.

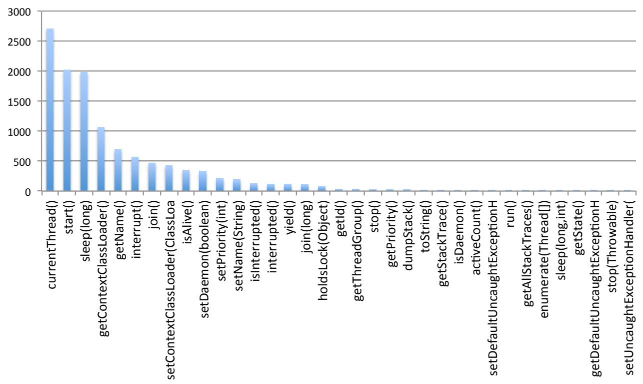


Fig. 3. Call site distribution per method of class *java.lang.Thread*. The horizontal axis shows method names and the vertical axis shows the number of times the method was used.

The output of the script is a list of method names for all invocations on instances of class *java.lang.Thread*. This list can easily be processed to produce the exact number of times each method appears in the list. Plotting this final result produces the graph shown in Figure 3, and we can clearly see that the most commonly used methods are *currentThread*, *start* and *sleep*.

With the help of Pangea, the authors have already used the analyis example presented here to build an improved documentation browser for Javadoc [4].

C. Longest Method Names

Previous empirical studies have shown that there is a correlation between the names of identifiers and program comprehension.

In the Java corpus we find method names such as: *whenCallEnsureThatContextOverloadedShouldThrowIllegalThread-StateExceptionUsingSuppliedMessage* and *getPointcutParserSupport-ingSpecifiedPrimitivesAndUsingSpecifiedClassLoaderForResolution*. It would be interesting to know how frequent are such long method names? Or a more general question is:

> *What is the distribution of method name lengths in open source systems?*

[3]The only candidate is the method of the statically declared type, ignoring subclasses and dynamic dispatch.

Let us choose as a measure of method name length the number of words obtained when *uncamelcasing* the method name.

```
Script is: [
  | interestingClasses sizes |
  interestingClasses := (Pangea model
    allModelClasses reject: [ :e|e
    isAnonymousClass ]).

  sizes := Bag new.
  interestingClasses allMethods do:
  [ :met|
    sizes add: met name numberOfCamelCaseWords
  ].

  CommaSeparated values
    * (Pangea model name);
    * ((1 to: 20) collect: [:e | sizes
    occurrencesOf: e]);
  print.
]
```

Listing 3. A pangea script that outputs the method name of all invocations done on instances of class *java.lang.Thread*

The script in Listing 3 computes the histogram of method name lengths as measured in *number of words in the method name* for a system. When deployed in parallel on both the Java and the Smalltalk corpus and the results aggregated we obtain the results in Figure 4. It is interesting to see that most method names in OO systems tend to be multi-word with less than 5 words. We also observe a trend in the Smalltalk code towards longer method names.

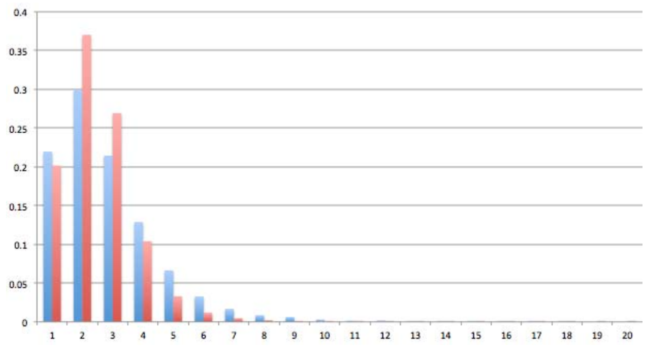


Fig. 4. The distribution (in percentages) of method name lengths (as measured in number of individual words) between Java (red) and Smalltalk (blue).

V. The data repository

The data repository currently contains OMSs for two languages — Java and Smalltalk. We plan to extend it further with models of systems written in other object-oriented languages. We invite the community to contribute with FAMIX models for corpora written in different languages as well as to propose and contribute different types of object oriented language-independent models for the existing systems.

A complete list of currently supported Pangea's software corpora is available on our website[4].

1) Java: Qualitas Corpus [5] is one of the most used software corpora available today. It is a curated collection of software systems consisting of more than 100 popular open source Java projects. It comes in two versions: a "recent" release, containing the last stable releases of 111 projects, and an "evolution" release, with all versions of 14 systems (486 versions in total).

Pangea's data repository contains the OMSs for all systems contained in Qualitas Corpus, releases: 20120401r and 20120401e.

All projects contained in Qualitas Corpus have been parsed using VerveineJ 192[5]. VerveineJ is an open source parser based on JDT[6] that generates FAMIX-based models out of Java files.

2) Smalltalk: To the best of our knowledge, there is no curated corpus of Smalltalk systems. Therefore, as a first step we have built models for a collection of Smalltalk systems selected according to following criteria: all the projects from SqueakSource[7] that have more than 100 classes and which pass a manual filtering step eliminating duplicated projects and repositories that do not represent single projects (*e.g.,* collections of utilities or exercises). This results in a total of 28 systems. The Corpus has been named Squeaksource-100 and can be found on the project website.

3) Other Languages: The workbench can be extended with other collections of software systems written in an object-oriented programming language as long as exporters from source code to FAMIX are available and as long as corpora are established.

These collections of software systems can be parsed and included in Pangea.Data using one of the following tools: inFamix (supports Java, C and C++) [8], CAnalyzer (supports Smalltalk and C) [9], VerveineJ (supports Java) [10], PMCS (supports C#) [11].

VI. Discussion

Pangea streamlines the way researchers and practitioners define and run analyses on a large collection of software systems, and as the data repository becomes richer with more systems, in more languages, it will allow multi-language empirical studies; this is not easy to do at the moment.

Pangea lets the user focus only on the analysis algorithm and ignore the details related to setting up the infrastructure needed to run the analysis. This reduces the amount of time needed to define the analysis and minimizes the number of lines of code required for its specification. Having a smaller code base for the analysis together with a publicly available

[4]http://scg.unibe.ch/research/pangea#corpora
[5]https://gforge.inria.fr/projects/verveinej/
[6]http://www.eclipse.org/jdt/
[7]http://www.squeaksource.com/
[8]http://www.intooitus.com/products/infamix
[9]http://www.squeaksource.com/CAnalyzer/
[10]https://gforge.inria.fr/projects/verveinej/
[11]https://bitbucket.org/erikdoe/pmcs/

and easily installable workbench simplifies the process of sharing analysis algorithms and makes experiments easier to reproduce.

A. Downloading vs. Generating OMSs

The data in Pangea can be large. For QualitasCorpus only the source code is 3.3G while intermediate models, and OMSs together with tools are 17.6G. This is why users can choose one of two strategies:

1) download the source code and generate the other artifacts locally.
2) skip the generation part and download directly the OMSs

Our benchmarks show that since parsing is often very time-consuming, downloading the models is preferred given an adequate internet connection.

The following table show a quantified comparison between the two mentioned strategies for "QualitasCorpus-20120401r". We analyze varying internet connection speeds and use a MacBook Pro with 2.5GHz, 8GB of RAM for model generation.

Strategy	1 Mbps	5 Mbps	10 Mbps
#1	29:34	5:55	2:57
#2	10:14	4:14	3:29

Once the setup has been complete, the cost of performing the actual analysis is low. For the previous examples and similar queries the order of magnitude of the required time is minutes.

B. Limitations

One limitation of Pangea is FAMIX — the meta-model adopted to represent software systems. As a common denominator between multiple OO programming languages it necessarily lacks specific information for individual languages (*e.g.,* detailed information such as the AST of a method's body is not captured by the meta-model). This is a disadvantage for users interested in performing low-level analysis at the level of individual statements. The limitation can be overcome by accessing the source code files distributed along the models by following the pointer associated to each code entity. This operation can be done in the context of a Moose analysis by two simple method invocations:

```
aCodeEntity  sourceAnchor completeText
```

Another limitation is that users must become familiar with the Smalltalk programming language.

VII. Related Work

Software corpora are becoming increasingly popular. The Qualitas Corpus has been used for more than 30 studies over the past 5 years.

Daniel Rodriguez et al. [6] have recently published a survey of all the existing software engineering repositories publicly available. Some of the repositories they mention — The Sourcerer project dataset [7], software-artifact infrastructure repository [8] — could be taken into consideration to extend Pangea.Data.

PROMISE [9] is a collection of intermediate results that have been obtained from previous analyses. Its goal is to provide raw materials for empirical studies. The provided data is user contributed, often uncorrelated and usually focused on very specific aspects and properties of software systems.

VIII. Conclusion and Future Work

In this paper we introduced Pangea, a workbench for setting up and running multi-language empirical studies. The main goal of Pangea is to reduce the cost of cross-language analyses by sharing a set of model files generated from well established software corpora. At the moment of writing, our data repository contains two versions of the Qualitas Corpus and a collection of Smalltalk projects. All the models are based on FAMIX, a language-independent meta-model that enables running analyses on software systems written in different object-oriented languages. Pangea also offers a convenient toolkit that was designed to automatically set up a pre-configured analysis environment, and run user-defined analyses.

We plan to enrich our data repository with more models from diverse OO languages: both FAMIX and beyond and explore the possibility of exposing our workbench as a web service.

Acknowledgements We gratefully acknowledge the financial support of the Swiss National Science Foundation for the project "Agile Software Assessment" (SNSF project No. 200020-144126/1, Jan 1, 2013 - Dec. 30, 2015).

References

[1] O. Nierstrasz and M. Lungu, "Agile software assessment," in *Proceedings of International Conference on Program Comprehension (ICPC 2012)*, 2012, pp. 3–10. [Online]. Available: http://scg.unibe.ch/archive/papers/Nier12bASA.pdf

[2] S. Demeyer, S. Tichelaar, and S. Ducasse, "FAMIX 2.1 — The FAMOOS Information Exchange Model," University of Bern, Tech. Rep., 2001.

[3] 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)*. New York, NY, USA: ACM Press, Sep. 2005, pp. 1–10, invited paper. [Online]. Available: http://scg.unibe.ch/archive/papers/Nier05cStoryOfMoose.pdf

[4] B. Spasojevic, M. Lungu, and O. Nierstrasz, "Overthrowing the tyranny of alphabetical ordering in documentation systems," in *Proceedings of International Conference on Software Maintenance and Evolution (ICSME 2014)*, 2014.

[5] E. Tempero, C. Anslow, J. Dietrich, T. Han, J. Li, M. Lumpe, H. Melton, and J. Noble, "Qualitas corpus: A curated collection of java code for empirical studies," in *2010 Asia Pacific Software Engineering Conference (APSEC2010)*, Dec. 2010, pp. 336–345.

[6] D. Rodriguez, I. Herraiz, and R. Harrison, "On software engineering repositories and their open problems," in *Realizing Artificial Intelligence Synergies in Software Engineering (RAISE), 2012 First International Workshop on*, june 2012, pp. 52 –56.

[7] S. Bajracharya, J. Ossher, and C. Lopes, "Sourcerer: An internet-scale software repository," in *Search-Driven Development-Users, Infrastructure, Tools and Evaluation, 2009. SUITE '09. ICSE Workshop on*, may 2009, pp. 1 –4.

[8] G. R. Group *et al.*, "Software-artifact infrastructure repository (SIR)," 2009.

[9] T. Menzies, B. Caglayan, E. Kocaguneli, J. Krall, F. Peters, and B. Turhan. (2012, June) The promise repository of empirical software engineering data. [Online]. Available: http://promisedata.googlecode.com