These notes have been compiled for a very short and very interdisciplinary and really informal brown bag meeting at the University of Koblenz-Landau (Koblenz campus).

BIG CODE SCIENCE

Ralf Lämmel
Software (Language) Engineer
University of Koblenz-Landau & Facebook
Some wild assumptions

Let’s say

- a developer writes 10-30 lines of code per day;
- each line of code costs 10 US$.
Increasingly bigger code

Source: http://www.visualcapitalist.com/millions-lines-of-code/

© 2018 Ralf Lämmel and SoftLang Team
What’s ‘Big Code Science’?

- Code Science is Data Science for code.
- Big Code is like Big Data except that data is code.
- (In fact, we also care about code-related artifacts.)

Big Code Science is the systematic (scientific) approach to accessing, analyzing, and understanding big data where the data here is code or data related to software development.
214 matches of searching “GitHub” on DBLP (18/03/19)

2018


- Gede Artha Azriadi Prana, Christoph Treude, Ferdian Thung, Thushari Atapattu, David Lo: *Categorizing the Content of GitHub README Files*. CoRR abs/1802.06997 (2018)


2017


Big Code Science at SoftLang

http://www.softlang.org/

© 2018 Ralf Lämmel and SoftLang Team
An early research question:
How many APIs are and how much of them is used in OSS?

Source: Ralf Lämmel, Ekaterina Pek, Jürgen Starek:
Large-scale, AST-based API-usage analysis of open-source Java projects.

© 2018 Ralf Lämmel and SoftLang Team
SAC 2011
We also need to clarify how to measure usage of API methods. The APIs or API methods used in a project provide insight into the API-related complexity of the project. In fact, such footprinting rules for obscure parts of the API. Coverage information is also helpful in improving API usability [13, 12].

There is a trend of increasing API footprint with project size. Figure 2 shows the numbers of known APIs (y-axis) that are used in the projects; reference projects are highlighted. The APIs or API methods used in a project provide insight into the API-related complexity of the project. In fact, such footprinting rules for obscure parts of the API. Coverage information is also helpful in improving API usability [13, 12].

Figure 3 shows the numbers of distinct API methods used in projects (without distinguishing APIs). Numbers of distinct API methods used in the projects increase slowly. Unbuilt projects, built projects, and reference projects are distinguished. The set of unbuilt projects exercises a higher maximum of used APIs than the set of built projects—potentially because of a correlation between the complexity of projects in the wild. The set of unbuilt projects gives a sense of the API footprint in axis). Unbuilt, built, and reference projects are distinguished. The used in the projects ordered by NCLOC-based project size (x-axis). Unbuilt, built, and reference projects are distinguished. The used in the projects ordered by NCLOC-based project size (x-axis). Unbuilt, built, and reference projects are distinguished.

An important form of API-usage analysis concerns API coverage. That is, how to precisely distinguish distinct methods so that counting uses is well defined. Particularly, in the case of instance methods, the situation is complicated due to inheritance, overloading, and polymorphism. As a starting point, we may distinguish methods by possible receiver type—no matter whether the method is overridden or inherited at a given subtype. Then, a method call is counted towards the polymorphism-based argument: the runtime receiver type may be the inherited implementation may be used, if not inherited). Such inclusion could also be made more precise by a global program analysis.

Methods on sub- and supertypes of static receiver types were not included. For simplification, we also considered overloaded methods. That is, how to precisely distinguish distinct methods so that counting uses is well defined. Particularly, in the case of instance methods, the situation is complicated due to inheritance, overloading, and polymorphism. As a starting point, we may distinguish methods by possible receiver type—no matter whether the method is overridden or inherited at a given subtype. Then, a method call is counted towards the polymorphism-based argument: the runtime receiver type may be the inherited implementation may be used, if not inherited). Such inclusion could also be made more precise by a global program analysis.
As it is the case with other forms of API-usage analysis, API
correlation on a log-log scale

There is a trend of increasing API footprint with project size.

We also need to clarify how to measure usage of API meth-
ods as basically one method.

In this context, we need to define what constitutes usage of
APIs or API methods used in a project provide insight into
the number of APIs used in a project as a proxy for the difficulty of
software asbestos

API migration as means to prioritize efforts, and to leave out map-
project of [3]. That is, coverage information is helpful in
aforementioned positions of API types in method calls, but count-
not as the hosting scope (in the case of a static call), or as the con-
the number of APIs used in a project as a proxy for the difficulty of
dependency of a project. In [16], we mention such API dependence
like data serves as a proxy for the API dependence or platform de-
numbers of known APIs used in the projects; reference projects are highlighted.
Numbers of known APIs used in the reference projects are plotted on top of built projects which
are really exercised in one program scope or only separately.

Figure 2 shows the number of known APIs (y-axis) that are
4.2 API coverage by the corpus

Project size (in MC)
Let us consider another specific API. We pick SAX—a push-based XML parsing API. The push-based characteristics imply that client code typically extends ‘handler’ classes or implements handler interfaces with handler methods such as startElement and endElement—to which XML-parsing events are pushed.

As a result, one should be prepared to find relatively low API coverage—expectation that a good API should be covered more or less by regular client code. We know that JDOM is a ‘true library’ for XML processing. We get, we pick JDOM—a DOM-like (i.e., tree-based, in-memory) API for XML processing. We know that JDOM is a ‘true library’ for XML processing. We get, we pick JDOM—a DOM-like (i.e., tree-based, in-memory)

We measured cumulative coverage for the methods of the SAX API for those SAX-using projects. The table with maxima and quartiles gives a good indication of the relatively low usage of the API for those JDOM-using projects. The table with maxima and quartiles gives a good indication of the relatively low usage of the API for those JDOM-using projects.

<table>
<thead>
<tr>
<th>Method</th>
<th>Min</th>
<th>1st Q</th>
<th>Median</th>
<th>Mean</th>
<th>3rd Q</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>startElement</td>
<td>0.3268</td>
<td>0.9804</td>
<td>2.22</td>
<td>3.023</td>
<td>4.369</td>
<td>11.44</td>
</tr>
<tr>
<td>endElement</td>
<td>0.3268</td>
<td>0.9804</td>
<td>2.22</td>
<td>3.023</td>
<td>4.369</td>
<td>11.44</td>
</tr>
</tbody>
</table>

Most of an API is never used.
The deeper research question:
How much OO is there in OSS?

<table>
<thead>
<tr>
<th>API</th>
<th># Projects</th>
<th># Methods</th>
<th># Distinct methods</th>
<th># Derived types</th>
<th># API types</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>impl.</td>
<td>ext.</td>
<td>any</td>
<td>impl.</td>
<td>over.</td>
</tr>
<tr>
<td>Swing</td>
<td>173</td>
<td>381</td>
<td>391</td>
<td>2512</td>
<td>11150</td>
</tr>
<tr>
<td>AWT</td>
<td>194</td>
<td>75</td>
<td>225</td>
<td>4201</td>
<td>756</td>
</tr>
<tr>
<td>Java Collections</td>
<td>120</td>
<td>0</td>
<td>120</td>
<td>986</td>
<td>0</td>
</tr>
<tr>
<td>SAX</td>
<td>28</td>
<td>21</td>
<td>42</td>
<td>428</td>
<td>90</td>
</tr>
<tr>
<td>JUnit</td>
<td>3</td>
<td>38</td>
<td>40</td>
<td>4</td>
<td>344</td>
</tr>
<tr>
<td>Core XML</td>
<td>11</td>
<td>5</td>
<td>14</td>
<td>89</td>
<td>13</td>
</tr>
<tr>
<td>SWT</td>
<td>5</td>
<td>8</td>
<td>10</td>
<td>37</td>
<td>86</td>
</tr>
<tr>
<td>log4j</td>
<td>1</td>
<td>8</td>
<td>8</td>
<td>25</td>
<td>87</td>
</tr>
<tr>
<td>Reflection</td>
<td>7</td>
<td>0</td>
<td>7</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>JMF</td>
<td>4</td>
<td>2</td>
<td>6</td>
<td>8</td>
<td>6</td>
</tr>
</tbody>
</table>
### 4.3 Framework-like API usage

In reality, many APIs allow for both—class library-like and framework-like usage. This is because many APIs are designed with both ends in mind. Some APIs are designed to be used as frameworks, while others are designed to be used as class libraries. The choice of which end to use depends on the specific needs of the project.

#### Table 4.

<table>
<thead>
<tr>
<th>API</th>
<th># Projects</th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>impl.</td>
<td>ext.</td>
<td>any</td>
<td>impl.</td>
<td>over.</td>
</tr>
<tr>
<td>Swing</td>
<td>173</td>
<td>381</td>
<td>391</td>
<td>2512</td>
<td>11150</td>
</tr>
<tr>
<td>AWT</td>
<td>194</td>
<td>75</td>
<td>225</td>
<td>4201</td>
<td>756</td>
</tr>
<tr>
<td>Java Collections</td>
<td>120</td>
<td>0</td>
<td>120</td>
<td>986</td>
<td>0</td>
</tr>
<tr>
<td>SAX</td>
<td>28</td>
<td>21</td>
<td>42</td>
<td>428</td>
<td>90</td>
</tr>
<tr>
<td>JUnit</td>
<td>3</td>
<td>38</td>
<td>40</td>
<td>4</td>
<td>344</td>
</tr>
<tr>
<td>Core XML</td>
<td>11</td>
<td>5</td>
<td>14</td>
<td>89</td>
<td>13</td>
</tr>
<tr>
<td>SWT</td>
<td>5</td>
<td>8</td>
<td>10</td>
<td>37</td>
<td>86</td>
</tr>
<tr>
<td>log4j</td>
<td>1</td>
<td>8</td>
<td>8</td>
<td>25</td>
<td>87</td>
</tr>
<tr>
<td>Reflection</td>
<td>7</td>
<td>0</td>
<td>7</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>JMF</td>
<td>4</td>
<td>2</td>
<td>6</td>
<td>8</td>
<td>6</td>
</tr>
</tbody>
</table>

Few APIs (frameworks) are exercised in an OO manner.
There is again a way of defining framework-like usage in a cumulative way—very similar to coverage analysis. That is, for a given implementation of an API interface, we count the number of distinct methods that are called, then such framework-like usage would be missed by i)—iii).

What are the parts of an API that account for framework-like usage: just a hand full of GUI, XML and collection types account for almost all the framework-like usage in the corpus. For instance, DOM-like APIs like JDOM or XOM are other APIs that are subject to framework-like usage more in—form. Closer inspection revealed that the JDOM API implements, for example, a bridge from in-memory XML trees to SAX events, and hence, it pushes itself as opposed to regular SAX-based functionality that is pushed. This is an unexpected but correct use of an API that is designed as an in-memory XML API. In contrast, there are other APIs that are subject to framework-like usage more in—form. Closer inspection revealed that the JDOM API implements, for example, a bridge from in-memory XML trees to SAX events, and hence, it pushes itself as opposed to regular SAX-based functionality that is pushed. This is an unexpected but correct use of an API that is designed as an in-memory XML API.

Table 4. Top 10 of the APIs with framework-like usage (sorted by the sum of numbers of API-interface implementations and overrides (i.e., client classes that subclass API classes and override API types within client code. In particular, we may measure i) the number of derived types that are implemented or overridden throughout the corpus: we show both absolute numbers of implementations/overrides and the number of derived types in client code, and the number of API types ever exercised in an OO manner.

The selected group may fail to be representative for the whole population. If we define the whole population to consist of all projects of interest'. However, we are not exactly sure how to show the types that are commonly involved in framework-like usage: just a hand full of GUI, XML and collection types account for almost all the framework-like usage in the corpus. For instance, DOM-like APIs like JDOM or XOM are other APIs that are subject to framework-like usage more in—form. Closer inspection revealed that the JDOM API implements, for example, a bridge from in-memory XML trees to SAX events, and hence, it pushes itself as opposed to regular SAX-based functionality that is pushed. This is an unexpected but correct use of an API that is designed as an in-memory XML API. In contrast, there are other APIs that are subject to framework-like usage more in—form. Closer inspection revealed that the JDOM API implements, for example, a bridge from in-memory XML trees to SAX events, and hence, it pushes itself as opposed to regular SAX-based functionality that is pushed. This is an unexpected but correct use of an API that is designed as an in-memory XML API.
A more recent research question:
How to profile developers in terms of API experience?

Source: Hakan Aksu, Ralf Lämmel, Wojciech Kwasnik:
Visualization of API Experience.
Softwaretechnik-Trends 36(2) (2016)
Figure 1: Domain & API references (DR and AR) for libGDX

be easily observed in this manner. For instance, the favorite domains (with the most references) of developer 2 are Meta, GUI, IO, and Media. The developers can be easily compared by means of the radii. For instance, developer 1 is the most experienced developer in the Archiving domain even though it makes only a small contribution to all the experience atoms of this developer.

4 Conclusion

The developed visualization approach summarizes several aspects of per-developer API usage in a concise and systematic manner. The overall idea of interpreting API usage (in terms of both references to API elements and referenced API elements) is not new [3]; the original insight of the present work is that ratios of APIs are compared for each individual developer and across all developers while adding the abstraction level of domains on top of APIs.

In future work, we plan to develop a comprehensive visualization tool that allows one to explore per-developer API usage along different dimensions based on the selection of a collection of developers, a window on the commit timeline, a collection of APIs or domains, and yet other parameters.

References


The different colors correspond to different APIs in the “GUI” domain.

The different squares with the same color correspond to different developers.
The different developers are profiled in terms of different API-related metrics.
A related and recent research question:
How to categorize APIs?

We use the data mining technique of ‘hierarchical clustering’ to compare APIs based on extracted class and method names.

Source: Johannes Härtel, Hakan Aksu, and Ralf Lämmel:
Classification of APIs by Hierarchical Clustering.
ICPC 2018
Yet another research question: How to detect and consolidate ‘clone and own’?

These are about 40 different systems that have been derived from each other. The layout conveys similarity.

Source: Thomas Schmorleiz, Ralf Lämmel: Similarity management of ‘cloned and owned’ variants.
Yet another research question:
How is a language used in the wild?

We were able to determine that most privacy policies in the world are ‘clones’ with just a few clone groups defining common use cases.

Source: Ralf Lämmel, Ekaterina Pek:
Understanding privacy policies - A study in empirical analysis of language usage.
Empirical Software Engineering 2013
A current research question:
What are the patterns of technology usage across projects?

These are patterns related to the EMF technology. For instance, EJc2 models that metamodel and derived Java code are not in sync in a certain way.
Is it all about *machine learning*?
Big Code Science is about much more than *machine learning*!

**Technical aspects**

- Syntax and semantics of languages
- Technological spaces (e.g., Java vs. Ruby)
- Software development platforms (e.g., GitHub)
Big Code Science is about much more than *machine learning*!

**Scientific and methodological aspects**

- Development of hypotheses
- Reproducibility of studies
- Realization of benefits
- Parameters
Goals of mining

* Productivity goals
* Quality-assurance goals
* Management goals
* Exploration goals

Fig. 1. Goals for Mining Software Repositories

- Productivity goals
  - Speed up developers
  - Identify opportunities
  - Identify change request impact
  - Spread knowledge (document)
  - Understand language-feature adoption
  - Understand tool usage
  - Understand software evolution
  - Understand code reviews

- Quality-assurance goals
  - Detect code clones (origin analysis)
  - Detect bugs
  - Facilitate model checking

- Exploration goals
  - Understand code reviews
Fig. 1. Goals for Mining Software Repositories

- bugs
- optimize testing efforts
- triage bug reports
- predict defects
- understand code reviews
- understanding processes
- understanding contribution models
- project popularity
- human factors
- estimate change effort
- detect developer feelings
- summarize customer opinions
These three research areas uncover artifacts which could potentially become more prominent. As example, related to green mining, artifacts about CPU, I/O, and memory traces are particularly interesting. Advanced technologies, which necessarily require energy-aware applications, such as the Google Glass, might unravel even further artifacts. Considering mobile software engineering, uncoverers that, for example, data gathered through Web IDEs could become relevant for further experiments. To better understand human behavior within the software development process, a variety of data sources can be mined. Data sources to better understand developers capture either data about the developer itself, as example through psycho-physiological measurements in particular situation while coding, or capture data about social interactions of developers. Devices, such as eye trackers, electrodermal activity, electrocardiograms, or electroencephalograms, allow detailed insights about individual behaviors during a programming task. Communication threads in emails or messages were part of early experiments in the MSR community. However, in recent years the range of these artifacts has increased. Recent experiments mine Twitter feeds, so it is conceivable that social networks, such as Facebook, will eventually become a mining artifact for software engineering as well.

Artifacts subjected to mining

These three research areas uncover artifacts which could potentially become more prominent. As example, related to green mining, artifacts about CPU, I/O, and memory traces are particularly interesting. Advanced technologies, which necessarily require energy-aware applications, such as the Google Glass, might unravel even further artifacts. Considering mobile software engineering, uncovers that, for example, data gathered through Web IDEs could become relevant for further experiments. To better understand human behavior within the software development process, a variety of data sources can be mined. Data sources to better understand developers capture either data about the developer itself, as example through psycho-physiological measurements in particular situation while coding, or capture data about social interactions of developers. Devices, such as eye trackers, electrodermal activity, electrocardiodiagrams, or electroencephalograms, allow detailed insights about individual behaviors during a programming task. Communication threads in emails or messages were part of early experiments in the MSR community. However, in recent years the range of these artifacts has increased. Recent experiments mine Twitter feeds, so it is conceivable that social networks, such as Facebook, will eventually become a mining artifact for software engineering as well.
These three research areas uncover artifacts which could potentially become more prominent. As examples, related to green mining, artifacts about CPU, I/O, and memory traces are particularly interesting. Advanced technologies, which necessarily require energy-aware applications, such as the Google Glass, might unravel even further artifacts. Considering mobile software engineering, uncovers that, for example, data gathered through Web IDEs could become relevant for further experiments. To better understand human behavior within the software development process, a variety of data sources can be mined. Data sources to better understand developers capture either data about the developer itself, as examples through psycho-physiological measurements in particular situations while coding, or capture data about social interactions of developers. Devices, such as eye trackers, electrodermal activity, electrocardiograms, or electroencephalograms, allow detailed insights about individual behaviors during a programming task. Communication threads in emails or messages were part of early experiments in the MSR community. However, in recent years the range of these artifacts has increased. Recent experiments mine Twitter feeds, so it is conceivable that social networks, such as Facebook, will eventually become a mining artifact for software engineering as well.
These three research areas uncover artifacts which could potentially become more prominent. As example, related to green mining, artifacts about CPU, I/O, and memory traces are particularly interesting. Advanced technologies, which necessarily require energy-aware applications, such as the Google Glass, might unravel even further artifacts. Considering mobile software engineering, uncovers that, for example, data gathered through Web IDEs could become relevant for further experiments. To better understand human behavior within the software development process, a variety of data sources can be mined. Data sources to better understand developers capture either data about the developer itself, as example through psycho-physiological measurements in particular situation while coding, or capture data about social interactions of developers. Devices, such as eye trackers, electrodermal activity, electrocardiograms, or electroencephalograms, allow detailed insights about individual behaviors during a programming task. Communication threads in emails or messages were part of early experiments in the MSR community. However, in recent years the range of these artifacts has increased. Recent experiments mine Twitter feeds, so it is conceivable that social networks, such as Facebook, will eventually become a mining artifact for software engineering as well.
Thanks!

http://www.softlang.org/