Multi-dimensional Exploration of API Usage

Coen De Roover
Software Languages Lab
Vrije Universiteit Brussel
Belgium

Ralf Lämmel
Software Languages Team
University of Koblenz-Landau
Germany

Ekaterina Pek
ADAPT Lab
University of Koblenz-Landau
Germany

***Abstract*—This paper is concerned with understanding API usage in a systematic, explorative manner for the benefit of both API developers and API users. There exist complementary, less explorative methods, e.g., based on code search, code completion, or API documentation. In contrast, our approach is highly interactive and can be seen as an extension of what IDEs readily provide today. Exploration is based on multiple dimensions: i) the hierarchically organized scopes of projects and APIs; ii) metrics of API usage (e.g., number of project classes extending API classes); iii) metadata for APIs; iv) project- versus API-centric views. We also provide the QUAATLAS corpus of Java projects which enhances the existing QUALITAS corpus to enable API-usage analysis. We implemented the exploration approach in an open-source, IDE-like, Web-enabled tool EXAPUS.**

***Index Terms*—API usage, code exploration, metadata, program comprehension, reverse engineering, QUAATLAS, EXAPUS, QUALITAS**

I. INTRODUCTION

The use (and the design) of APIs is an integral part of OO software development. Projects are littered with usage of easily a dozen APIs; perhaps every third line of code references some API [1]. Accordingly, understanding APIs or their usage must be an important objective. Much of the existing work, as discussed in detail in §III, focuses on some form of documentation or discovery of API-usage scenarios perhaps by code completion or code search [2], [3], [4], [5].

In our work on API migration [6], [7], we have always missed an *exploration*-based approach to understanding API usage in a systematic manner. In this paper, we describe a form of exploration which is interactive in nature while leveraging query-based program understanding underneath [8], [9].

Contributions: We identify *exploration insights* as they are expected by API users and developers with regard to their overall intention to understand API usage. For instance, one insight may relate to the coupled use of APIs in a project or packages thereof; another insight may relate to the use of particular API facets (e.g., 'non-trivial API usage') across projects in a corpus. Such insights rely on multiple dimensions of exploration, e.g., hierarchical organization of scopes and project- versus API-centric perspectives.

We set up QUAATLAS (for QUALITAS API Atlas)—a Java-based *corpus for API-usage analysis* that builds on top of the existing QUALITAS corpus while revising it substantially such that fact extraction can be applied with the level of precision required for API-usage analysis, while also adding metadata that supports exploration and records knowledge about APIs.

We provide conceptual support for said exploration insights by means of an *abstract model of API-usage views*, which we implemented in EXAPUS (for Explore API usage)—an IDE-like, Web-enabled tool so that we also provide *tool support for exploration* that can be used by others for exploration experiments. The paper's website[1] provides access to QUAATLAS, EXAPUS, and screencasts demonstrating their usage.

Road-map: §II motivates multi-dimensional exploration of API usage by means of an 'exploration story'. §III discusses related work and further motivates our research. §IV describes basic concepts regarding APIs and API usage. §V describes the development of the QUAATLAS corpus that can be used for experimenting with API-usage exploration in the Java context. §VI presents an inventory of abstract insights expected from exploration. §VII describes an abstract model of views for exploration. §VIII describes the EXAPUS tool which supports the described exploration approach. §IX concludes the paper.

II. AN EXPLORATION STORY

Joanna Programmer is a new hire in software development at the fictional *Acme Corporation*. The company's main product is *JHotDraw* and *Joanna* was hired to respond to pending renovation plans.

JHotDraw has been reverse-engineered in the past for the sake of incorporating crosscutting concerns such as logging, enabling refactoring (e.g., for design patterns), or generally understanding its architecture at various levels. Such existing research does not directly apply to *Joanna*'s assignment. She is asked to renovate *JHotDraw* to use JSON instead of XML; to replace native GUI programming by HTML5 compliance. Further, an Android SDK-based version is needed as well. *Joanna* is not particularly familiar yet with *JHotDraw*, but she quickly realizes that much of the challenge lies in the API usage of *JHotDraw*. This is when *Joanna* encounters EXAPUS.

Fig. 1 summarizes API usage in *JHotDraw* as analyzed with EXAPUS. The tree view shows all APIs as they are known to EXAPUS and exercised by *JHotDraw*. The heavier the border, the more usage. Rectangles proxy for APIs that are packages. Triangles proxy for APIs with a package subtree.

Let us focus on the requirement for replacing XML by JSON. In Fig. 1, two XML APIs show up: *DOM* and *SAX*. *Joanna* begins with an exploration of *DOM* usage. Fig. 3 summarizes *DOM* usage in *JHotDraw* as analyzed with EXAPUS.

[1] http://softlang.uni-koblenz.de/explore-API-usage

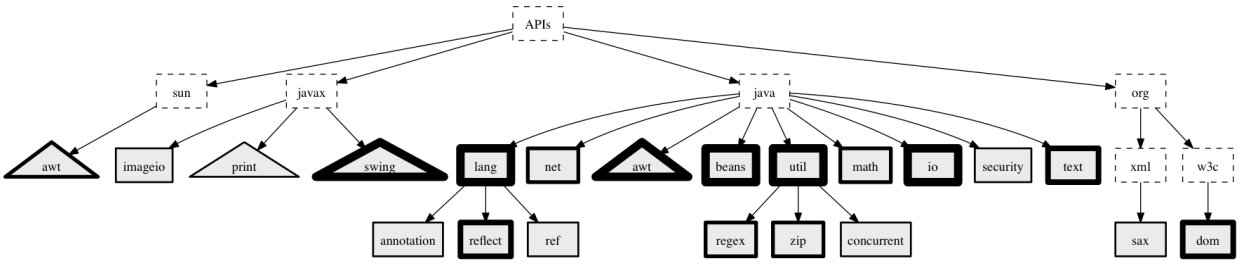

Fig. 1. API usage in *JHotDraw* with scaling applied to numbers of API references.

Pattern	API Tags	Element	Name	Line	#ref	#elem
jhotdraw					94	19
org					94	19
jhotdraw					94	19
xml					94	19
JavaxDOMInput					62	11
JavaxDOMOutput					24	10
css					8	5
CSSRule					7	5
StyleManager					1	1
applyStylesTo(Element)					1	1
METHOD_PARAMETER	DOM	INTERFACE	org.w3c.dom.Element	42	1	1

The view only shows packages and types with API references to *DOM*. Out of the 13 top-level packages of *JHotDraw*, only 1 of them, the *xml* package and its subpackage *css* reference *DOM*. There is a total of 4 class types that contain references. The combined reference count is 94 where 19 unique API elements are referenced, which is a relatively small number of used API elements in the view of hundreds of API elements declared by the *DOM* API.

```
public void applyStylesTo(Element elem) {
   for (CSSRule rule : rules) {
      if (rule.matches(elem)) {
         rule.apply(elem);
      }
   }
}
```

Fig. 2. The slice of *JHotDraw* with *DOM* usage.

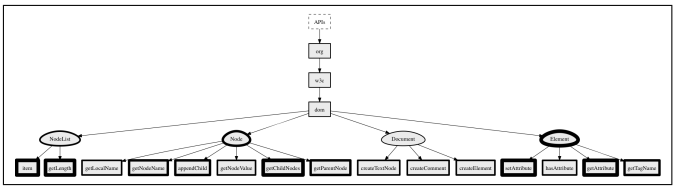

Fig. 3. Minuscule view for *DOM* usage in *JHotDraw*: with leaves for methods, eggs for types, and the remaining nodes for packages.

Encouragingly, *DOM*'s footprint in *JHotDraw* only covers a few types and methods.

A logical option for continuing the exploration is to examine the distribution of API usage across *JHotDraw*. In this manner, *Joanna* gets a sense of the locality of API usage. The corresponding view is shown in Fig. 2 and it strikingly reveals good news in so far that *DOM* usage is limited to the *JHotDraw* package *org.jhotdraw.xml*, which she shall explore further to prepare a possible XML-to-JSON migration.

III. Related Work

We identify the following categories of related work.

1) Project Exploration: There are several conceptual styles of project comprehension. The work of Brühlmann et al. [10] exemplifies a human-intensive effort, where experts annotate project parts to capture human knowledge. They further use the emerged meta-model to analyze features and architectural flaws of the project.

Query-driven comprehension can proceed through user-defined queries that identify code of interest, as in the work of Hou and Hoover [11] Mens and Kellens [12] or De Roover et al. [9] such that a comprehensive tool suite facilitates defining and exploring query results. Alwis and Murphy [8] identify and investigate pre-defined queries for exploration of a software system, e.g., "What calls this method."

Visual summary of projects usually involves some sort of scaling, color coding, and hierarchical grouping, as discussed by Lanza and Ducasse [13]. More involved visualizations have been explored as well, such as a 3D city metaphor [14].

Our approach combines these conceptual styles. We allow the user to accumulate and refine knowledge about APIs, their facets, and domains. The exploration activities explained in the paper are intuitive; flexibility in their combination enables answering typical questions like those in [8]. Tag clouds, tables, and trees accompanied by metrics provide basic and familiar visual aid in exploration.

2) Measuring API Usage: Research on API usage often leverages usage frequency, or popularity, of APIs and their parts. For instance, Mileva et al. use popularity to identify most commonly used library versions [15] or to identify and predict API usage trends over time [16]. Holmes et al. appeal to popularity as the main indicator: for the API developer, to be able to prioritize efforts and be informed about consumption of libraries; for the API user, to be able to identify libraries of interest and be informed of ways of their usage [17]. Eisenberg et al. use font scaling w.r.t. popularity of API elements to help navigate through its structure [5], [18]. Thummalapenta and Xie use Google search to find relevant code examples for further frequency analysis of used API elements [19]. Ma et al. investigate coverage of Java Standard API to identify which parts are ignored by the API users [20]. In our work, we suggest several metrics indicating specific forms of API usage; their distribution is integrated in the table and graph views of our tool, providing sorting and scaling.

3) Understanding API Usage: Robillard and DeLine discovered in a field study on API learning obstacles that API users prefer to learn from patterns of related calls rather than illustrations of individual methods [21]. Hou and Li report similar obstacles based on an exploratory study of newsgroup discussions [22]. Generally, information about API usage may be used in helping developers. Nasehi and Maurer show that API unit tests can be used as usage examples [23]. Zhong et al. cluster API calls and mine patterns to recommend useful code snippets to API users [3]. Bruch et al. develop intelligent code completion that narrows down the possible suggestions to those API elements that are actually relevant [24]. Hou et al. [25] compare different filtering, sorting and grouping strategies to this end. The latter would, for instance, group all methods related to managing the components of an AWT container. Mandelin et al. present an approach for synthesizing a snippet to fill in a gap in the code using an API, given certain contextual information [26]. Our effort differs in that it enables navigating both projects and APIs in the familiar IDE-like manner with API usage in focus. We also identify a catalogue of exploration activities to perform.

IV. Basic Concepts

We set up the basic concepts underlying this paper: APIs, API usage, and API-usage metrics. We also augment the basic notion of API with extra concepts for API domains and API facets to raise the level of abstraction in exploration.

APIs: We use the term API to refer to the actual interface but also to the underlying implementation. We do not pay attention to any distinction between libraries and frameworks. We simply view an API as a set of types (classes, interfaces, etc.) referable by name and distributed together for use in software projects. Without loss of generality, this paper invokes *Java* for most illustrations and intuitions.

Indeed, we assume that package names, package prefixes, and types within packages can be used to describe APIs. For instance, the package prefix *javax.swing* (and possibly others) could be associated with the *Swing* API for GUI programming. It is important that we view *javax.swing* as a package *prefix* because *Swing* is indeed organized in a package tree. In contrast, the *java.util* API corresponds to all the types in the package of ditto name. There are various sub-packages of *java.util*, but they are preferably considered separate APIs. In fact, the *java.util* API deserves further breakdown because the package serves de facto distinct purposes, notably Java's collections and Java's event system. (This is not an uncommon situation.) We use the term *sub-API* to refer to declared subsets of the types in a given API.

Clearly, an API may exist in different versions, in which case it needs to be decided whether or not the versions should be treated like different APIs, as far as API-usage analysis is concerned.

API Usage: We are concerned with API usage in given software projects. API usage is evidenced from any sort of reference from projects to APIs. References are directly associated with syntactical patterns in the code of the projects, e.g., a method call in a class of a project that invokes a method of an API type, or a class declaration in a project that explicitly extends a class of an API. The resulting patterns can hence be used to classify API references and to control exploration with regard to the kinds of references to present to users.

A reasonably precise analysis of API usage requires that the underlying projects are 'resolved' in that each API reference in a project can be followed to the corresponding declaration in the API. Further, since exploration of API usage relies on the developer's view on source code of projects, we effectively need compilable source code of all projects.

API-usage Metrics: For quantifying API usage, metrics are needed that can be used in exploration views in different ways, e.g., for ordering (elements or scopes of APIs or projects) or for scaling in the visualization of API usage. For the purpose of this paper, the following metrics suffice:

#proj: Number of projects referencing APIs.
#api: Number of APIs being referenced.
#ref: Number of references from projects to APIs.
#elem: Number of API elements being referenced.
#derive: Number of project types derived from API types.
#super: Number of API types serving as supertype for derivations.
#sub: Number of project types serving as subtype for derivations.

These metrics can be applied, of course, to different selections of projects or APIs as well as specific packages, types, or methods thereof. For instance, we may be interested in #api for a specific project. Also, we may be interested in #ref for some part of an API.

Further, these metrics can be configured to count only specific patterns. It is easy to see now that the given metrics are not even orthogonal because, for example, #derive can be obtained from #ref by only counting patterns for 'extends' and 'implements' relationships.

API Domains: We assume that each API addresses some programming domain such as XML processing or GUI programming. We are not aware of any general, widely adopted attempt to associate APIs with domains, but the idea appears to merit further research. We have begun collecting programming domains (or in fact, API domains) and tagging APIs appropriately. Let us list a few API domains and associate them with well-known *Java* APIs:

GUI: GUI programming, e.g., *Swing* and *AWT*.
XML: XML processing, e.g., *DOM*, *JDOM*, and *SAX*.
Data: Data structures incl. containers, e.g., *java.util*.
IO: File- and stream-based I/O, e.g., *java.io* and *java.nio*.
Component: Component-oriented programming, e.g., *JavaBeans*.
Meta: Meta-programming incl. reflection, e.g., *java.lang.reflect*.
Basics: Basic language support, e.g., *java.lang.String*.

API domains are helpful in reporting API usage and quantifying API usage of interest in more abstract terms than the names of individual APIs, as will be illustrated in §VI.

API Facets: An API may contain dozens or hundreds of types each of which has many method members in turn. Some APIs use sub-packages to organize such API complexity, but those sub-packages are typically concerned with advanced API usage whereas the core facets of API usage are not distinguished in any operational manner. This makes it hard to understand API usage at a somewhat abstract level.

1. input : $corpus, candidateList$
2. output : $corpus$
3. for each $name$ in $candidateList$:
4. $(p_{src}, p_{bin}) = obtainProject(name)$;
5. $patches = exploratoryBuild(p_{src}, p_{bin})$;
6. $timestamp = build(p_{src}, patches)$;
7. $(java, classes, jars) = collectStats(p_{src})$;
8. $java' = filter(java)$;
9. $(jars_{built}, jars_{lib}) = detectJars(timestamp, java', jars)$;
10. $java'_{compiled} = detectJava(timestamp, java', classes, jars_{built})$;
11. $p'_{src} = (java'_{compiled}, jars_{lib})$;
12. $p'_{bin} = jars_{built}$;
13. $p' = (p'_{src}, p'_{bin})$;
14. if $validate(p') : corpus = corpus + p'$;

Fig. 4. Pseudocode describing the corpus (re)-engineering method.

Accordingly, we propose leveraging a notion of API facets in the sense of aspects or concerns supported by the API. In this paper, we assume that facets are represented as named collections of specific API types or methods. As an illustration, we name a few API facets of the typical *DOM*-like API such as *DOM* itself, *JDOM*, or *dom4j*:

Input / Output: De-/serialization for *DOM* trees.
Observation: Getter-like access and other 'read only' forms.
Addition: Addition of nodes et al. as part also of construction.
Removal: Removal of nodes et al. as a form of mutation.
Namespaces: XML namespace manipulation.
Nontrivial XML: Use of CDATA, PI, and other XML idiosyncrasies.
Nontrivial API: Usage of types and methods that are beyond normal API usage. For instance, XML APIs may provide some framework for node factories or adapters for API integration.

API facets are helpful in communicating API usage to the user at a more abstract level than the level of individual types and methods, as will be illustrated in §VI. We leverage knowledge of the APIs to identify (to name) API facets and to tag APIs appropriately. The idea of grouping API members, e.g., by their functional roles, has also been studied in related work on code completion; see §III.

V. The Quaatlas Corpus for API-usage Analysis

Our study requires a suitable corpus of mature, well-developed projects coming from different application domains. Arguably, such projects show sufficient and advanced API usage. We decided to restrict ourselves to open-source Java projects; in order to increase quality and reproducibility of our research, we decided to use an existing, established and curated, collection of Java projects—the Qualitas corpus [27], release 20101126r. As we discuss in §IV, API usage entails the ability to resolve types. However, Qualitas does not guarantee the availability of a project's library types. The collection consists of source and binary forms as they are provided by the project developers.

In the interest of similar research tasks that require a dependency-resolved corpus, we detail our method for corpus (re-)engineering. The resulting dependency-resolved Qualitas variant is available on the paper's website.

A. Method

The pseudocode depicted in Fig. 4 describes our corpus (re)-engineering method. The input is a (possibly empty) *corpus* to be extended and a list of candidate projects, $candidateList$, to be added to it. The output is the corpus populated with refined projects.

Line 4 assumes that a project can be obtained both in its source and binary forms (e.g., downloading them from the project website). During an exploratory build (line 5), the nature of the project is manually investigated by an expert. The expert investigates how the project is built, what errors occur during the build (if any), and how to patch them. At this stage, we also compare the set of built JARs with the JARs in the binary distribution form of the project. If the former set is smaller than the latter (e.g., because default targets in build scripts may be insufficient and a series of target calls or invocation of several build scripts is needed), we attempt to push the build of the project for completeness. Once the exploratory build is successful, we are able to automatically build the project (line 6), if necessary after applying patches.

After the build, we collect the full path, creation and modification times of each file in the project (line 7). For Java files we extract qualified names of contained top-level types, for class files we detect their qualified names. For JARs we explore their contents and collect information about the contained class files.

On line 8, we apply a filter, keeping only the source code that we consider to be both *system* and *core* (see §V-C). On line 9, we use the known start time of the build together with information about Java types computed on lines 7 and 8 to classify the JARs found after the build either as library JARs or as built JARs. On line 10, we use the identified built JARs and the compiled class files to identify Java types that were compiled during the build. On line 11, we refine the project's source code form p'_{src} to include only the compiled Java types together with the necessary library JARs. On line 12, the binary form p'_{bin} is refined to consist of the built JARs.

The refined project p' (line 13) is validated (line 14) by rebuilding the project in a sandbox, outside its specific setup, making sure to use only those files that have been identified by the method.[2] A successful sandbox build indicates that source and library files have been discerned correctly. In that case, we add the refined project to the corpus (line 14).

This pseudocode is, of course, an idealized description of the process. In practice, we would execute line 4 only once per project; line 5 could be repeated several times, if the build coverage is found unsatisfactory in terms of compiled types—something that becomes clear only on line 9. We treat lines 6–10 as an atomic action, call it a "corpus build," and perform it on regular basis.

B. Exploratory Builds

The Qualitas corpus contains 106 projects. We were able to build 86 projects, of which 54 required a patch to build. We limit our exploratory build efforts to approximately 2-3 hours per project.

[2]In practice, we use an Eclipse workspace with automatically generated configuration files (i.e., `.system` and `.classpath`).

We distinguish the following patch operations. The most common patch operation is *adding JARs*: almost 200 JARs were added to 30 projects to make them build (not counting libraries that Maven downloads automatically). Another patch operation is *creating directories* (20 projects) and *adding resource files* (4 projects). Other patch operations are *patching build files* (6 projects) and *setting up usage of a specific Java version* (4 projects). Almost half of the patched projects (25 out of 54) needs only one type of fixing.

We used ANT to build the majority of projects. Some projects support more than one build setup (e.g., both ANT and Maven)—in such cases we opted for ANT. In total, for building the corpus, ANT was used 69 times, Maven 2.x—11 times, Maven 1.x—3 times, a Bash script—2 times, and Makefile—once.

C. Identifying Relevant Types

We heuristically classify files (and hence, types) as *core*, *test*, and *demo* by checking whether some part of the file path starts or ends with any of the following terms. Category *test*: test, tests, testcase, testcases, testsuite, testsuites, testing, junit.[3] Category *demo*: demo, demos, tutorial, tutorials, example, examples, sample, samples. For the *test* category, we also check whether the file name ends with either "Test," "TestCase," or "TestSuite." By default, source code is considered to be *core*.

Table I lists descriptive statistics about percentage of non-*core* files per project. We observe that an average project contains 10,73 % non-*core* files. Usually those files are tests.

TABLE I
NON-*core* FILES IN PROJECTS.

%	Min.	1st Qu.	Median	Mean	3rd Qu.	Max
test	0.04	4,57	9,75	11,81	17,04	41,91
demo	0.11	0,91	1,78	5,88	6,66	30,65
total	0.04	5,36	10,73	14,14	19,09	46,2

QUALITAS comes with metadata that designates package prefixes of Java types as implying system scope (or simply *system*), which corresponds roughly to our *core*, as we measured. Our approach handles packages with mixed *core* and other types.

D. Conversion to Eclipse Projects

We automatically convert the built and possibly patched projects into Eclipse projects, while only including types that are identified as both *system* by QUALITAS and *core* by us. Successful compilation of a project in Eclipse shows that the classpath and the discerning of source and library code are correct. We successfully converted 79 projects (out of 86 built ones). There are a few cases where validation via export to Eclipse is not possible. For example, in case of the **nekohtml** project, different parts of the code are compiled with different versions of a library JAR, which cannot be easily modeled in an Eclipse project with its 'global' setting of the classpath.

In some cases, the exported code would not compile due to non-*core* dependencies, while we could make the projects compile by revising the classification of *core* code. In those few cases where a project's *core* code indeed required the non-*core* code (e.g., to run self tests via a specific command line option), we included the compiled non-*core* classes into a library JAR with the name `<project>-sys.jar`. Some systems use 3rd-party source code; we also ship those compiled classes in a library JAR with the name `<project>-nonSys.jar`.

E. API Metadata

The corpus, obtained so far, is still not a sufficient starting point for exploring API usage. Any conceivable experiment relies on basic awareness of the involved APIs. Accordingly, we systematically analyzed the package tree implied by API usage (external dependencies) in all projects of the corpus. We leveraged the concepts of API domains and facets as discussed in §IV. The results are available online.

We identified 98 APIs in terms of packages or package prefixes, and associated online resources. We assigned names to the APIs, where necessary. We also added a few sub-APIs to compensate for otherwise too diverse APIs. In this manner, we faced more than 100 APIs. We found it difficult to experiment with exploration while using the diverse API names directly. Hence, we identified 27 API domains and assigned them to APIs. (With a few exceptions, each API is associated with exactly one domain.) In order to be able to explore some specific APIs in more detail, we also explored API facets for a few APIs. Most notably, at the time of writing, we added a matured set of API facets for the *JDOM* API.

VI. EXPLORATION INSIGHTS

Overall, developers need to understand API usage, when APIs relate to or affect their development efforts such as a specific maintenance, migration, or integration task. We assume that an exploration effort can be decomposed into a series of primitive exploration activities meant to increase understanding via some attainable insights. In this section, we present a catalogue of such *expected, abstract insights*.

A. Format of Insight Descriptions

We use the following format. The *Intent* paragraph summarizes the insight. The *Stakeholder* paragraph identifies whether the insight benefits the *API developer*, the *project developer*, or both. The *API usage* paragraph quantifies API usage of interest, e.g., whether one API is considered or all APIs. The *View* paragraph describes, in abstract terms, how API-usage data is to be rendered. The *Illustration* paragraph applies the abstract insight concretely to APIs and projects of QUAATLAS. We use different forms of illustrations: tables, trees, and tag clouds. The *Intelligence* paragraph hints at the 'operational' intelligence supported by the insight.

[3]We did not apply this heuristic to **junit**, since the application area of this project is testing. Instead, **junit** types were classified manually.

Pattern	#refs	#elems	#derives	#super	#sub
informa	874	50	0	0	0
jspwiki	744	76	3	2	3
roller	733	40	0	0	0
columba	197	27	0	0	0
velocity	89	47	4	3	4
jmeter	8	4	0	0	0

Fig. 5. *JDOM*'s API Dispersion in QUAATLAS (project-centric table).

B. The *API Dispersion* Insight

Intent – Understand an API's dispersion in a corpus by comparing API usage across the projects in the corpus.

Stakeholder – API developer.

API Usage – One API.

View – The listing of projects with associated API-usage metrics for quantitative comparison and API facets for qualitative comparison.

Illustration – Fig. 5 summarizes *JDOM*'s dispersion quantitatively in QUAATLAS. 6 projects in the corpus exercise *JDOM*. The projects are ordered by the #ref metric with the other metrics not aligning. Only 2 projects (*jspwiki* and *velocity*) exercise type derivation at the boundary of API and project.

Intelligence – The insight is about the significance of API usage across corpus. In the figure, arguably, project *jspwiki* shows the most significant API usage because it references the most API elements. Project *jmeter* shows the least significant API usage. Observation of significance helps an API developer in picking hard and easy projects for compliance testing along API evolution—an easy one to get started; a hard one for a solid proof of concept. For instance, development of a wrapper-based API re-implementation for API migration relies on suitable 'test projects' just like that [6], [7].

C. The *API Distribution* Insight

Intent – Understand API distribution across project scopes.

Stakeholder – Project developer.

API Usage – One API.

View – The hierarchical breakdown of the project scopes with associated API-usage metrics for quantitative comparison and API facets for qualitative comparison.

Illustration – Remember *JHotDraw*'s slice of *DOM* usage in Fig. 2 in §II. This view was suitable for efficient exploration of project scopes that directly depend on *DOM*.

Intelligence – The insight may help a developer to decide on the feasibility of an API migration, as we discussed in §II.

D. The *API Footprint* Insight

Intent – Understand what API elements are used in a corpus or varying project scopes.

Stakeholder – Project developer and API developer.

API Usage – One API.

View – The listing of used API packages, types, and methods.

Pattern	#projs	#refs	#elems	#derives
org.jdom	6	2391	84	5
Element	5	1912	44	1
Document	5	160	6	1
Namespace	4	82	6	0
Attribute	4	70	6	0
Text	4	67	4	2
JDOMException	6	54	4	0
Content	3	21	6	0
CDATA	3	9	1	0
DocType	2	4	1	0
ProcessingInstruction	2	4	1	0
IllegalDataException	1	2	1	0
Comment	1	2	1	0
EntityRef	1	2	1	0
Verifier	1	1	1	0
DefaultJDOMFactory	1	1	1	1
org.jdom.input	6	103	10	0
org.jdom.output	5	101	24	2
org.jdom.xpath	2	50	8	0

Fig. 6. *JDOM*'s API Footprint in QUAATLAS (API-centric table).

Scope	Tags incl. facets		#proj
org.jdom	JDOM		2
Verifier	JDOM,Nontrivial API		1
DefaultJDOMFactory	JDOM,Nontrivial API		1
EXTENDS_CLASS	JDOM,Nontrivial API	AnakiaJDOMFactory	1
Document	JDOM		0

Fig. 7. 'Non-trivial API' usage for package *org.jdom* in QUAATLAS.

Illustration – Remember the tree-based representation of the API footprint for *JHotDraw* as shown in Fig. 3 in §II. In a similar manner, while using a table-based representation, Fig. 6 summarizes *JDOM* usage across QUAATLAS. All *JDOM* packages are listed. The core package is heavily used and thus the listing is further refined to show details per API type. Ordering relies on the #ref metric. Clearly, there is little usage of API elements outside the core package.

Intelligence – Overall, the footprint describes the (smaller) 'actual' API that needs to be understood as opposed to the full ('official') API. For instance, many APIs enable nontrivial, framework-like usage [1], [28], but in the absence of actual framework-like usage, the project developer may entertain a much simpler view on the API. In the context of API evolution, an API developer consults an API's footprint to minimize changes that break actual usage or to make an impact analysis for changes. In the context of wrapper-based API re-implementation for API migration, an API developer or a project developer (who develops a project-specific wrapper) uses the footprint to limit the effort [6], [7].

E. The *Sub-API Footprint* Insight

Intent – Understand usage of a sub-API in a corpus or project.

Stakeholder – API developer and, possibly, project developer.

API Usage – One API.

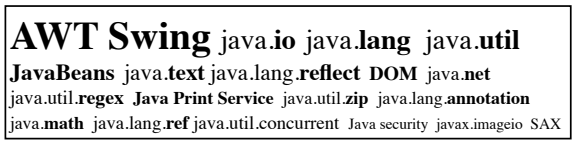

Fig. 8. The API Cocktail of *JHotDraw* (cloud of API tags).

View – A list as in the case of the API Footprint insight, except that it is narrowed down to a sub-API of interest.

Illustration – Fig. 7 illustrates 'Non-trivial API' usage for *JDOM*'s core package. The selection is concerned with a project type which extends the API type *DefaultJDOMFactory* to introduce a project-specific factory for XML elements. Basic IDE functionality could be used from here on to check where the API-derived type is used.

Intelligence – In the example, we explored non-trivial API usage, such as type derivation at the boundary of project and API—knowing that it challenges API evolution and migration [7]. More generally, developers are interested in specific sub-APIs, when they require detailed analysis for understanding. API developers (more likely than project developers) may be more aware of sub-APIs; they may, in fact, capture them, as part of the exploration. (This is what we did during this research.) Such sub-API tagging, which is supported by the Sub-API Footprint insight may ultimately improve API documentation in ways that are complementary to existing approaches [4], [5].

F. The API Cocktail Insight

Intent – Understand what APIs are used together in larger project scopes.

Stakeholder – Project developer.

API Usage – All APIs.

View – The listing of all APIs exercised in the project or a project package with API-usage metrics applied to the APIs.

Illustration – Remember the tree-based representation of the API cocktail for *JHotDraw* as shown in Fig. 1 in §II. The same cocktail of 20 APIs is shown as a tag cloud in Fig. 8. Scaling is based on the #ref metric.

Intelligence – The cocktail lists and ranks APIs that are used in the corresponding project scope. Thus, the cocktail proxies as a measurement for system complexity, required developer skills, and foreseeable design and implementation challenges. API usage is part of the software architecture, in the sense of "what makes it hard to change the software" and chances are that API usage may cause some "software or API asbestos" [29]. While a large cocktail may be acceptable and unavoidable for a complex project, the cocktail should be smaller for individual packages in the interest of a modularized, evolvable system.

G. APIs Versus Domains

We can always use API domains in place of APIs to raise the level of abstraction. Thus, any insight that compares APIs may as well be applied to API domains. APIs are concrete technologies while API domains are more abstract software concepts. Consider Fig. 9 for illustration. It shows API domains for all of *JHotDraw* and also for its *undo* package. Thus, it presents the API cocktails of Fig. 8 in a more abstract manner.

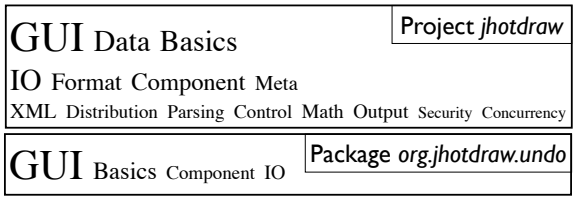

Fig. 9. Cocktail of domains for *JHotDraw*.

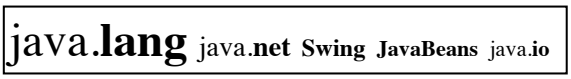

Fig. 10. API Coupling for *JHotDraw*'s interface *org.jhotdraw.app.View*.

H. The API Coupling Insight

Intent – Understand what APIs or API domains are used together in smaller project scopes.

Stakeholder – Project developer.

API Usage – All APIs.

View – See §VI-F except APIs or domains are listed for smaller project scopes.

Illustration – Fig. 10 shows API Coupling for the interface *org.jhotdraw.app.View* from the *JHotDraw*'s *app* package[4]. According to the documentation, the package "defines a framework for document-oriented applications and provides default implementations". The *View* type "paints a document on a *JComponent* within an *Application*". (*Application* is the main type from the package which "handles the lifecycle of views and provides windows to present them on screen".) The coupled use of APIs can be dissected in terms of the involved types as follows:

java.lang: trivial usage of strings.
java.net: types for the location to save the view.
JavaBeans: de-/registration of *PropertyChangeListener*s.
java.io: exception handling for reading/writing views.
Swing: usage of *JComponent* on which to paint a document; usage of *ActionMap* for actions on the GUI component.

Intelligence – Simultaneous presence of several domains or APIs in a relatively small project scope may indicate accidental complexity and poor separation of concerns. Thus, such exploration may reveal a code smell [30], [31] that is worth addressing. Alternatively, a dissection, as performed for the illustrative example, may help in understanding the design and reasonable API dependencies.

I. The API Profile Insight

Intent – Understand what API facets are used in varying project scopes.

Stakeholder – Project developer and, possibly, API developer.

[4]The lifecycle of the interface as explained by its documentation: http://www.randelshofer.ch/oop/jhotdraw/JavaDoc/org/jhotdraw/app/View.html

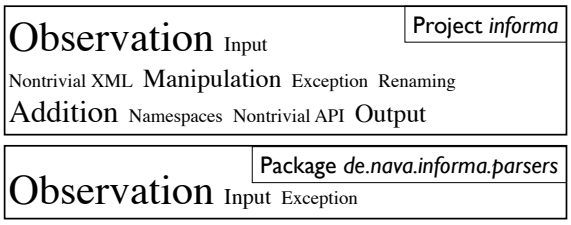

Fig. 11. *JDOM*'s API Profile in the *informa* project (cloud of facet tags).

API Usage – One API with available facets.

View – The listing of all API facets exercised in the selected project scope with API-usage metrics applied to the facets.

Illustration – Fig. 11 shows *JDOM* profiles for a project and one of its packages. The project, as a whole, exercises most facets of the API. In contrast, the selected package is more focused; it is concerned only with loading XML into memory, reading access by getters and friends, and some inevitable exception handling. There is no involvement of namespaces, non-trivial XML, or data access other than observation.

Intelligence – At the level of a complete project, the profile reveals the API facets that the project depends on. As some of the facets are more idiosyncratic than others, such exploration may, in fact, reveal "software or API asbestos" [29], as discussed in §VI-F. For instance, the *(J)DOM* facets 'Non-trivial API' and 'Non-trivial XML' and to a lesser extent also 'Namespaces' proxy for development challenges or idiosyncrasies. At the level of smaller project scopes, an API's profile may characterize an actual usage scenario, as in the case of the profile at the bottom of Fig. 11. Such a facet-based approach to understanding API-usage scenarios complements existing, more code pattern-based approaches [2], [3]. API profiles also provide feedback to API developers with regard to 'usage in the wild', thereby guiding API evolution or documentation.

VII. Exploration Views

Let us systematically conceptualize attainable views in abstract terms. In this manner, a more abstract model of exploration arises and a foundation for tool support is provided.

We approach this task essentially as a data modeling problem in that we describe the structure behind views and the underlying facts. We use Haskell for data modeling.[5]

A. Forests

We begin by modeling the (essential) facts about projects and APIs as well as API usage. To this end, we think of two forests: one for all the projects in the corpus, another for all the APIs used in the corpus.

```
-- Forests as collections of named trees
data Forest = Forest [(UqName,PackageTree)]
```

[5]Products are formed with "(...)". Lists are formed with "[...]". We use Haskell's **data** types to group alternatives (as in a sum); they are separated by '|'. Each alternative groups components (as in a product) and is labeled by a constructor name. Enums are degenerated sums where the constructor name stands alone without any components. Other types may suffice with **type** aliases on top of existing types.

Each project or API gives rise to one tree (root) in the respective forest. Such a tree breaks down recursively into package layers. If a package layer corresponds to an actual package, then it may also contain types. Types further break down into members. Thus:

```
-- Trees breaking down into packages, types, etc.
data PackageTree = PackageTree [PackageLayer]
data PackageLayer = PackageLayer UqName [PackageLayer] [Type]
data Type = Type UqName [Member] [Ref]
data Member = Member Element UqName [Type] [Ref]
data Element = Interface | Class | InstanceMethod | StaticMethod | ...

-- Different kinds of names
type RName = QName -- qualified names within forests
type QName = [UqName] -- qualified names within trees
type UqName = String -- unqualified names
```

In both forests, we associate types and members with API-usage references; see the occurrences of Ref. Depending on the forest, the references may be 'inbound' (from project to API) or 'outbound' and each reference may be classified by the (syntactic) pattern expressing it. Thus:

```
data Ref = Ref Direction Pattern Element RName
data Direction = Outbound | Inbound
data Pattern = InstanceMethodCall | ExtendsClass | ...
```

The components of a reference carry different meanings depending on the chosen direction:

	Outbound	**Inbound**
Pattern	Project pattern	Project pattern
Element	API element	Project element
RName	Name of API element	Name of project element

The project forest is obtained by walking the primary representation of projects and deriving the forest as a projection/abstraction at all levels. The API forest is obtained by a (non-trivial) transposition of the project forest to account for the project-specific jars and memory constraints on simultaneously open projects.

B. View Descriptions

We continue with the descriptions of views. These are the executable models that are interpreted on top of the forests of APIs and projects. Here is the overall structure of these descriptions:

```
type View = (
  Perspective ,          -- Project- versus API-centric
  ApiSelection ,         -- APIs and parts thereof to consider
  ProjectSelection ,     -- Projects and parts thereof to consider
  Details ,              -- Details to retain
  Metrics )              -- Metrics to be applied

data Perspective = ApiCentric | ProjectCentric
```

The API-centric perspective uses the hierarchical organization of APIs (packages, sub-packages, types, members) as the organizational principle of a view. Likewise, the project-centric perspective uses the hierarchical organization of projects as the organizational principle of a view.

Selection of projects, APIs, or parts thereof is based on names of APIs and projects as well as qualified names for the relevant scopes; we do not cover here selection based on API

domain tags and API facet tags, which would require only a routine extension:

```
type ApiSelection     = Selection
type ProjectSelection = Selection
data Selection
 = UniversalSelection              -- Select entire forest
 | Selection [(UqName, Scope)]     -- Select tree scopes
```

Scopes for selection are described as follows:

```
data Scope
 = RootScope                       -- Select entire tree
 | PrefixScope [UqName]            -- Selection by package prefix
 | PackageScope [UqName]           -- Selection by package name
 | TypeScope [UqName]              -- Selection by type name
 | MethodScope [UqName] Signature  -- Selection by method signature

type Signature = ... -- details omitted
```

We left out some forms, e.g., scopes for fields or nested types. However, all of the above forms have proven relevant in practice. For instance, we have used package scopes and prefix scopes for API and API domain tagging respectively. Likewise, we have used type scopes and method scopes for API facet tagging.

Each view description controls details for each selection:

```
type Details = (
  [ ProjectDetail ],   -- Project elements to retain
  [ ApiDetail ] )      -- API elements to retain

type ProjectDetail = (Element, Usage)
type ApiDetail = (Element, Usage)
type Usage = Bool      -- Whether to retain only elements with usage
```

(See above for type *Element*.) The selection of details is important for usability of exploration views. For instance, Fig. 3 shows only API elements that are actually used to summarize an API foot print concisely. In contrast, Fig. 7 shows all API types to better understand what API types possibly could exercise the chosen API facet.

Finally, applicable API-usage metrics are to be identified for the view. The choice of metrics serves multiple purposes. First, the final, actual view should only include metrics of interest to limit the presented information. Second, metrics can be identified for ordering/ranking entries in the actual views, as we have seen throughout §II and §VI. Third, metrics can be configured to only count certain aspects of API-usage. Last but not least, selection of metrics lowers the computational complexity of materializing views. Thus:

```
type Metrics = [(Metric, Maybe Order)]
data Metric = RefMetrics [Source] [Target]
            | ElemMetric [Source] [Target]
            | DeriveMetric [Derivation]
            | ... -- Further metrics omitted

data Order = Ascending | Descending -- Whether to order by the metric

-- What API references to count
type Source = Pattern -- Usage patterns to count
type Target = Element -- API elements to count

-- Forms of derivation to count
data Derivation = ProjectClassExtendsApiClass
                | ProjectClassImplementsApiInterface
                | ProjectInterfaceExtendsApiInterface
```

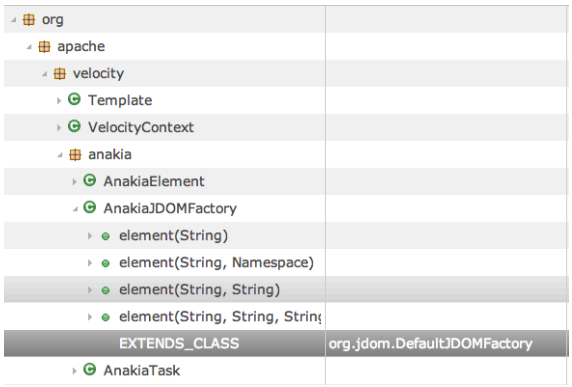

Fig. 12. The dual view for Fig. 7 (project-centric table).

For instance, the #ref metric can be configured to only count references to API types as opposed to, for example, API methods; the #derive metric can be configured to only count class-to-class extension as opposed to other forms of derivation.

To summarize, means of selection, details, and metrics provide a rich domain-specific query language to express what API usage should be included into an actual view and how to rank API usage. Thereby, a foundation is provided for interactive tool support.

C. Operations on Views

Because of the hierarchical nature of forests, established means of 'package exploration', as in an IDE are immediately feasible. That is, one can fold and unfold scopes; references to API or project elements can be resolved to their declarations and declarations can be associated with references to them. Beyond 'package exploration', views can be defined to dissect API usage explicitly.

More interestingly, exploration can switch between API-centric and project-centric perspectives. Fig. 12 shows a project-centric view which is fundamentally dual to the API-centric view of Fig. 7 in that the selected outgoing reference corresponds to the originally incoming reference selected in Fig. 7. The hierarchical exploration has been readily unfolded to expose the encompassing project scopes. Such travel between perspectives may be very insightful. In the example at hand, an API developer may have spotted the relevant API usage in the API-centric view, as part of a systematic exploration of non-trivial API usage in the corpus. In an attempt to better understand the broader context of the API reference, the developer needs to consult the project-centric view for the culprit. Such context switches are laborious when only basic IDE support for package exploration is available.

VIII. The Exapus Exploration Platform

The Exapus web server processes all Java projects in the Eclipse workspace it is pointed to. Fact extraction proceeds through a recursive descent on the ASTs produced by the Eclipse JDT. Whether an AST node of a project references an API member is determined on a case-by-case basis. In

general, identifiers are resolved to their corresponding declaration. Identifiers that resolve to a binary rather than a source member are considered an API reference. Hence, we require the workspace to have been prepared as described in §V. For each reference, EXAPUS extracts the referenced element (e.g., a method declaration), the referencing pattern (e.g., a super invocation) as well as the encompassing project scope in which the reference resides (i.e., a path towards the root of the AST).

Exploration views as of §VII are computed by selecting references from the resulting fact forest (e.g., only those to a particular sub-API) and superimposing one of either two hierarchical organizations: a project-centric hierarchy of project members and the outbound references within their scope; or an API-centric hierarchy of API members and the inbound references within their scope.

The EXAPUS web interface enables exploring the computed exploration views through trees (e.g., Fig. 1) and tables (e.g., Fig. 2). An exploration view can be refined further on the kind of the referenced elements (e.g., a particular type) and the referencing pattern (e.g., constructor invocation), as well as sorted by a particular metric. Multiple views can be shown simultaneously and navigated between. The interface owes its dynamic and IDE-like feel to the widgets of the Eclipse Rich Ajax Platform.

IX. CONCLUSION

We have described and illustrated a powerful notion of multi-dimensional exploration of API usage, which is implemented in the tool EXAPUS and can be experimented with on the grounds of the QUAATLAS corpus. The tool, the corpus, and all related metadata are available via the paper's website.

In the future, we hope to enrich API-usage exploration by flow-sensitive analyses, which is important when interprocedural API-usage scenarios or flow-based determination of API coupling are to be enabled.

Metadata (tags) for APIs, API domains, and API facets are being integrated with the knowledge base of http://101companies.org/, thereby also enabling rule-based detection of those phenomena on other corpora including the one of 101companies [32]. We are also working on *corpus engineering* so that the maintainance and improvement of corpora such as QUAATLAS is more effectively supported by the community.

The introduced notion of API-usage exploration calls for empirical research on understanding API usage where exploration is taken into account in addition to more established means of documentation, code search, and code completion.

ACKNOWLEDGMENTS

Coen De Roover is funded by the *Cha-Q* SBO project sponsored by the "Flemish agency for Innovation by Science and Technology" (IWT Vlaanderen).

REFERENCES

[1] R. Lämmel, E. Pek, and J. Starek, "Large-scale, AST-based API-usage analysis of open-source Java projects," in *SAC*, 2011, pp. 1317–1324.

[2] J. Jiang, J. Koskinen, A. Ruokonen, and T. Systä, "Constructing usage scenarios for API redocumentation," in *ICPC*, 2007, pp. 259–264.

[3] H. Zhong, T. Xie, L. Zhang, J. Pei, and H. Mei, "MAPO: Mining and recommending API usage patterns," in *ECOOP*, 2009, pp. 318–343.

[4] J. Stylos, A. Faulring, Z. Yang, and B. A. Myers, "Improving API documentation using API usage information," in *VL/HCC*, 2009, pp. 119–126.

[5] D. S. Eisenberg, J. Stylos, A. Faulring, and B. A. Myers, "Using association metrics to help users navigate API documentation," in *VL/HCC*, 2010, pp. 23–30.

[6] T. T. Bartolomei, K. Czarnecki, R. Lämmel, and T. van der Storm, "Study of an API migration for two XML APIs," in *SLE*, 2010, pp. 42–61.

[7] T. T. Bartolomei, K. Czarnecki, and R. Lämmel, "Swing to SWT and back: Patterns for API migration by wrapping," in *ICSM*, 2010, pp. 1–10.

[8] B. de Alwis and G. C. Murphy, "Answering conceptual queries with Ferret," in *ICSE*, 2008, pp. 21–30.

[9] C. De Roover, C. Noguera, A. Kellens, and V. Jonckers, "The SOUL tool suite for querying programs in symbiosis with Eclipse," in *PPPJ*, 2011, pp. 71–80.

[10] A. Brühlmann, T. Gîrba, O. Greevy, and O. Nierstrasz, "Enriching reverse engineering with annotations," in *MoDELS*, 2008, pp. 660–674.

[11] D. Hou and H. J. Hoover, "Using SCL to specify and check design intent in source code," *IEEE Trans. Software Eng.*, vol. 32, no. 6, pp. 404–423, 2006.

[12] K. Mens and A. Kellens, "IntensiVE, a toolsuite for documenting and checking structural source-code regularities," in *CSMR*, 2006, pp. 239–248.

[13] M. Lanza and S. Ducasse, "Polymetric views - A lightweight visual approach to reverse engineering," *Trans. Software Eng.*, vol. 29, no. 9, pp. 782–795, 2003.

[14] R. Wettel, M. Lanza, and R. Robbes, "Software systems as cities: A controlled experiment," in *ICSE*, 2011, pp. 551–560.

[15] Y. M. Mileva, V. Dallmeier, M. Burger, and A. Zeller, "Mining trends of library usage," in *ERCIM Workshops*, 2009, pp. 57–62.

[16] Y. M. Mileva, V. Dallmeier, and A. Zeller, "Mining API popularity," in *TAIC PART*, 2010, pp. 173–180.

[17] R. Holmes and R. J. Walker, "Informing Eclipse API production and consumption," in *OOPSLA*, 2007, pp. 70–74.

[18] D. S. Eisenberg, J. Stylos, and B. A. Myers, "Apatite: A new interface for exploring APIs," in *CHI*, 2010, pp. 1331–1334.

[19] S. Thummalapenta and T. Xie, "SpotWeb: Detecting framework hotspots and coldspots via mining open source code on the Web," in *ASE*, 2008, pp. 327–336.

[20] H. Ma, R. Amor, and E. D. Tempero, "Usage patterns of the Java standard API," in *APSEC*, 2006, pp. 342–352.

[21] M. P. Robillard and R. DeLine, "A field study of API learning obstacles," *Empir. Softw. Eng.*, vol. 16, no. 6, pp. 703–732, 2011.

[22] D. Hou and L. Li, "Obstacles in using frameworks and APIs: An exploratory study of programmers' newsgroup discussions," in *ICPC*, 2011, pp. 91–100.

[23] S. M. Nasehi and F. Maurer, "Unit tests as API usage examples," in *ICSM*. IEEE, 2010, pp. 1–10.

[24] M. Bruch, M. Monperrus, and M. Mezini, "Learning from examples to improve code completion systems," in *ESEC/SIGSOFT FSE*, 2009, pp. 213–222.

[25] D. Hou and D. M. Pletcher, "An evaluation of the strategies of sorting, filtering, and grouping api methods for code completion," in *ICSM*, 2011, pp. 233–242.

[26] D. Mandelin, L. Xu, R. Bodík, and D. Kimelman, "Jungloid mining: Helping to navigate the API jungle," in *PLDI*, 2005, pp. 48–61.

[27] 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 *APSEC*, 2010, pp. 336–345.

[28] R. Lämmel, R. Linke, E. Pek, and A. Varanovich, "A framework profile of .NET," in *WCRE*, 2011, pp. 141–150.

[29] A. Klusener, R. Lämmel, and C. Verhoef, "Architectural modifications to deployed software," *Sci. of Comput. Program.*, vol. 54, no. 2–3, pp. 143–211, 2005.

[30] E. V. Emden and L. Moonen, "Java quality assurance by detecting code smells," in *WCRE*, 2002, pp. 97–108.

[31] C. Parnin, C. Görg, and O. Nnadi, "A catalogue of lightweight visualizations to support code smell inspection," in *SOFTVIS*, 2008, pp. 77–86.

[32] J.-M. Favre, R. Lammel, M. Leinberger, T. Schmorleiz, and A. Varanovich, "Linking documentation and source code in a software chrestomathy," in *WCRE*, 2012, pp. 335–344.