Lecture series on
Software knowledge analytics

API topics (some of them ongoing research)

— API clustering — An exercise in abstraction

— Joint API usage — An exercise in causality

— API developer profiles — An exercise in hypothesis building and validation

Ralf Lämmel, Uni Koblenz, May 2022

Published
In ICPC 2018

API clustering

An exercise in abstraction

Johannes Härtel, Hakan Aksu, and Ralf Lämmel
Uni Koblenz, SoftLang Team

Source image: https://pixabay.com/illustrations/the-fourth-dimension-fractal-noise-2713919/

The Problem of Comprehending a Set of APIs

Crowd-sourced Categories

Home » Categories » XML Processing

XML Processing

Sort: **popular** | newest

1. **Xerces2 J** — **1,584** usages

xerces » xercesImpl — Apache

Xerces2 is the next generation of high performance, fully compliant XML ...

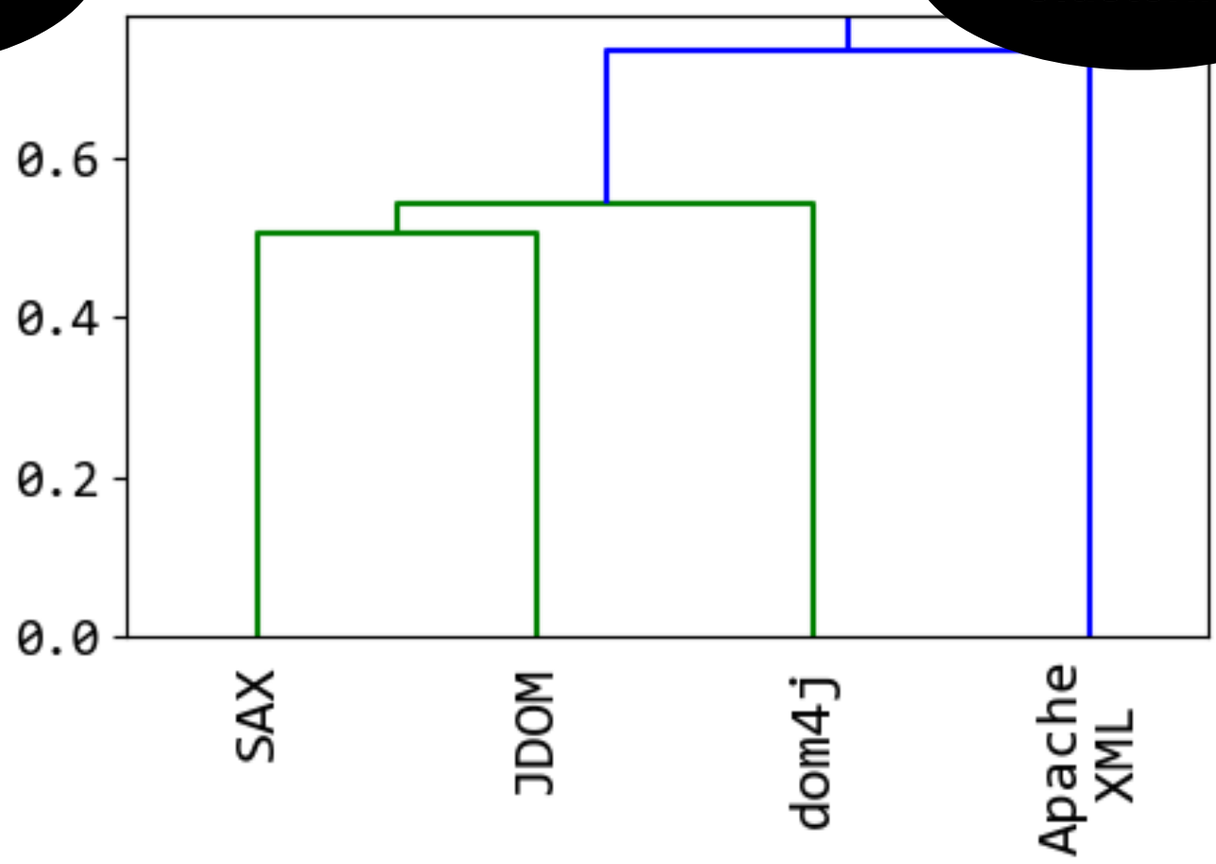
Last Release on Feb 20, 2013

2. **Dom4j** — **1,519** usages

dom4j » dom4j — BSD

dom4j: the flexible XML framework for Java

Computed Clustering



More Recent Problems

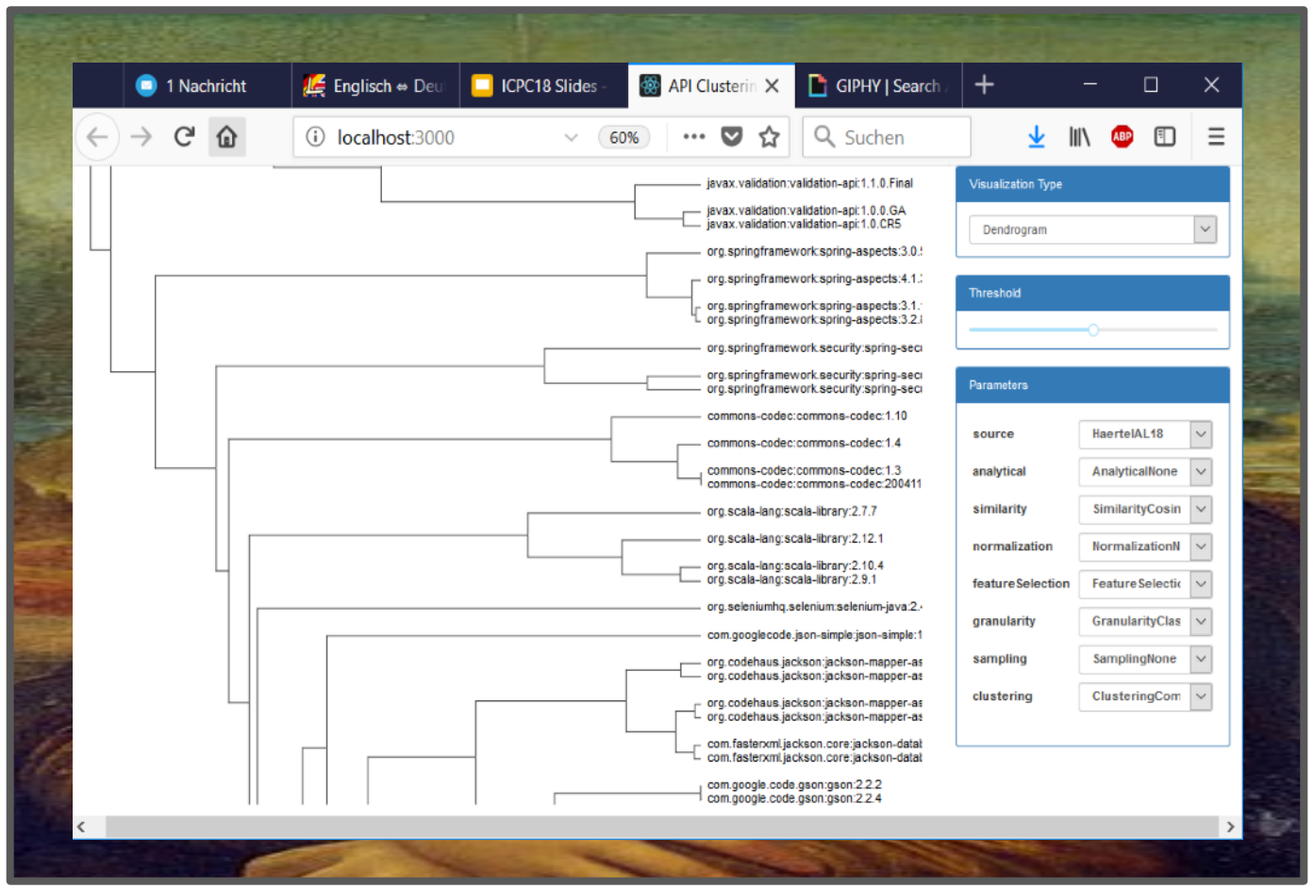
Table 2: The 10 categories of third-party libraries with more version changes.

Category	Updates	Upgrades	Downgrades
Graphical User Interface	808	607	201
Utilities	478	351	127
HTTP	60	38	22
Page Navigation	42	24	18
JSON	30	18	12
Annotations	23	23	0
Google Services	15	12	3
Date and Time	14	12	2
Device	14	9	5
HTML Parser	9	7	2

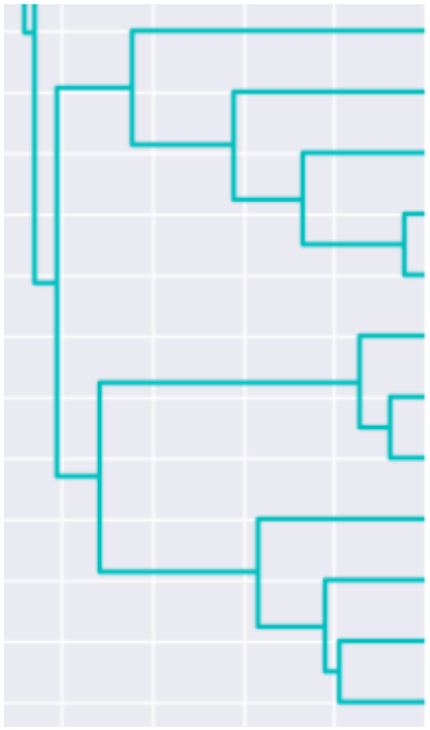
[47]

[47] *Pasquale Salza, Fabio Palomba, Dario Di Nucci, Cosmo D'Uva, Andrea De Lucia, Filomena Ferrucci. 2018. Do Developers Update Third-Party Libraries in Mobile Apps?. In ICPC. IEEE Computer Society.*

A Visualization (Demo)

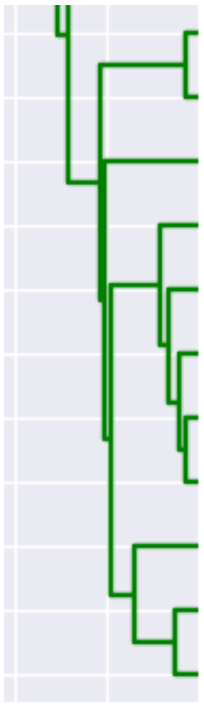


A Visualization (Demo): One API Clustering



```
[PostgreSQL Drivers] postgresql:postgresql:7.2
[PostgreSQL Drivers] postgresql:postgresql:7.4.1-jdbc3
[PostgreSQL Drivers] org.postgresql:postgresql:9.4.1212
[PostgreSQL Drivers] org.postgresql:postgresql:9.4-1201-jdbc41
[PostgreSQL Drivers] postgresql:postgresql:9.1-901-1.jdbc4
[JDBC Extensions] org.springframework:spring-jdbc:4.2.5.RELEASE
[JDBC Extensions] org.springframework:spring-jdbc:1.2.9
[JDBC Extensions] org.springframework:spring-jdbc:2.0.6
[MySQL Drivers] mysql:mysql-connector-java:3.0.10
[MySQL Drivers] mysql:mysql-connector-java:5.0.5
[MySQL Drivers] mysql:mysql-connector-java:5.1.38
[MySQL Drivers] mysql:mysql-connector-java:5.1.6
```

A Visualization (Demo): Another API Clustering



[MySQL Drivers] mysql:mysql-connector-java:5.0.5
[MySQL Drivers] mysql:mysql-connector-java:5.1.6
[JDBC Pools] commons-dbcp:commons-dbcp:1.0
[MySQL Drivers] mysql:mysql-connector-java:5.1.38
[MySQL Drivers] mysql:mysql-connector-java:6.0.6
[PostgreSQL Drivers] org.postgresql:postgresql:9.4.1212
[PostgreSQL Drivers] org.postgresql:postgresql:9.4-1201-jdbc41
[PostgreSQL Drivers] postgresql:postgresql:9.1-901-1.jdbc4
[PostgreSQL Drivers] postgresql:postgresql:7.2
[MySQL Drivers] mysql:mysql-connector-java:3.0.10
[PostgreSQL Drivers] postgresql:postgresql:7.4.1-jdbc3

Two Baselines

Two Baselines: ‘Cocktail from [36]’ Baseline

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*.

[36]

not [36]

GUI Data Basics — Project *jhotdraw*
IO Format Component Meta
XML Distribution Parsing Control Math Output Security Concurrency

GUI Basics Component IO — Package *org.jhotdraw.undo*

Fig. 8. Cocktail of domains for *JHotDraw*.

[36]

[36] *Coen De Roover, Ralf Lämmel, and Ekaterina Pek. 2013. Multi-dimensional exploration of API usage. In ICPC. IEEE Computer Society, 152–161.*

Two Baselines: ‘Curated Suite’ Baseline

Select the most used APIs by mining 2.5 Million POMs on Github.

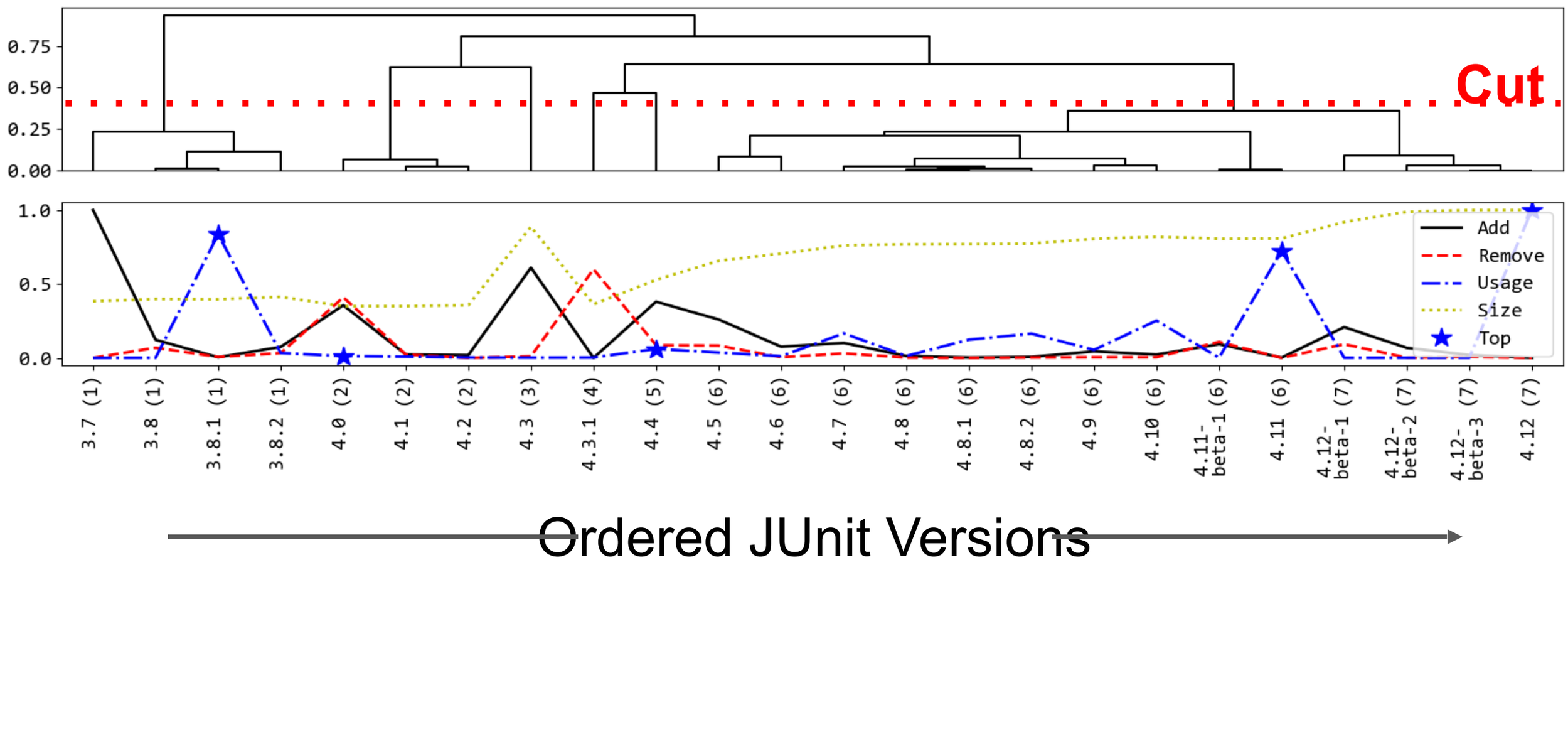
GroupId, ArtifactId & Version	#API usg.	API rank	#Vers. usg.
junit:junit:4.11	552625	0	113152
mysql:mysql-connector-java:5.1.38	183490	1	16039
org.springframework:spring-webmvc:4.2.5.RELEASE	175226	2	7520
org.slf4j:slf4j-api:1.7.5	167060	3	27329
org.springframework:spring-context:.1.6.RELEASE	164921	4	6978
log4j:log4j:1.2.17	157603	5	80550
org.springframework:spring-core:4.2.5.RELEASE	130559	6	5897
org.slf4j:slf4j-log4j12:1.7.5	122700	7	16546
org.springframework:spring-web:4.2.5.RELEASE	119705	8	5496

. . .

.
.
.

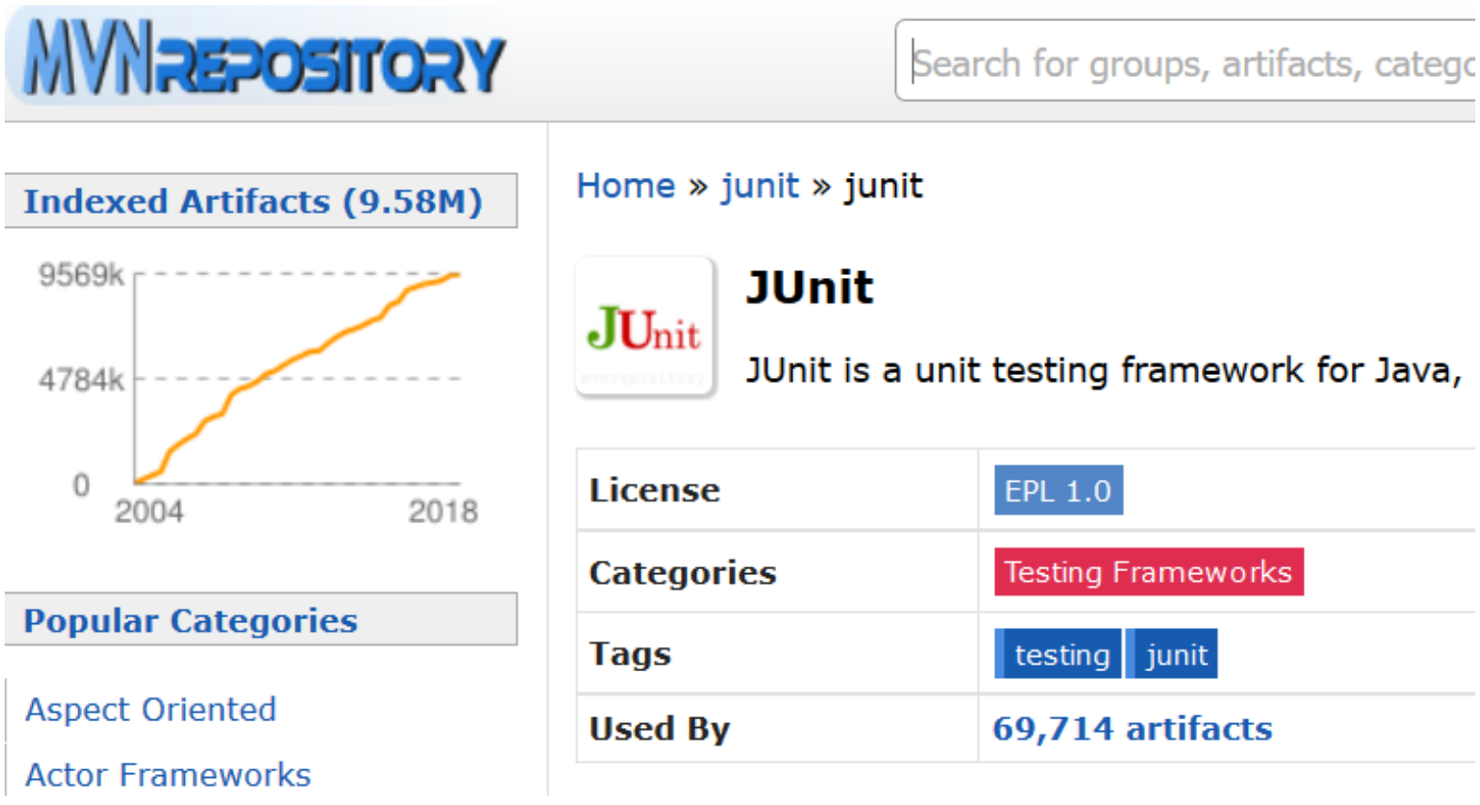
Two Baselines: ‘Curated Suite’ Baseline

Select popular and different JUnit versions by mining the change history (added/removed methods between JARs).

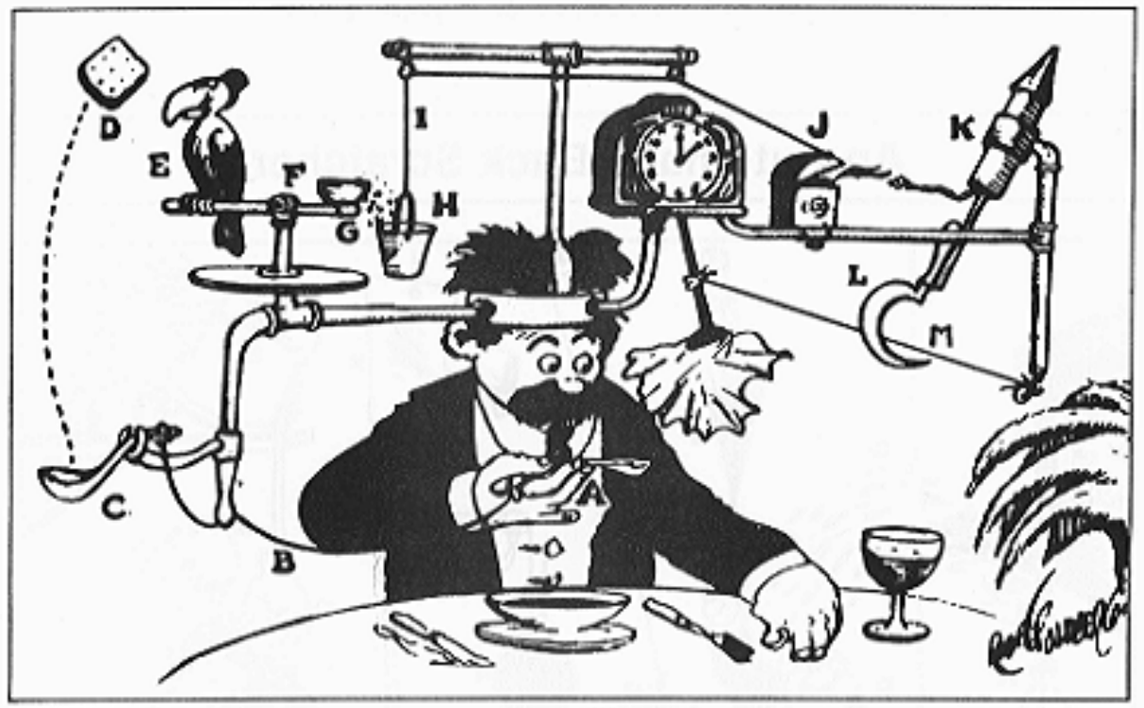


Two Baselines: ‘Curated Suite’ Baseline

Mine tags and categories from Maven Central Repository.



Clustering, Variations and Limitations



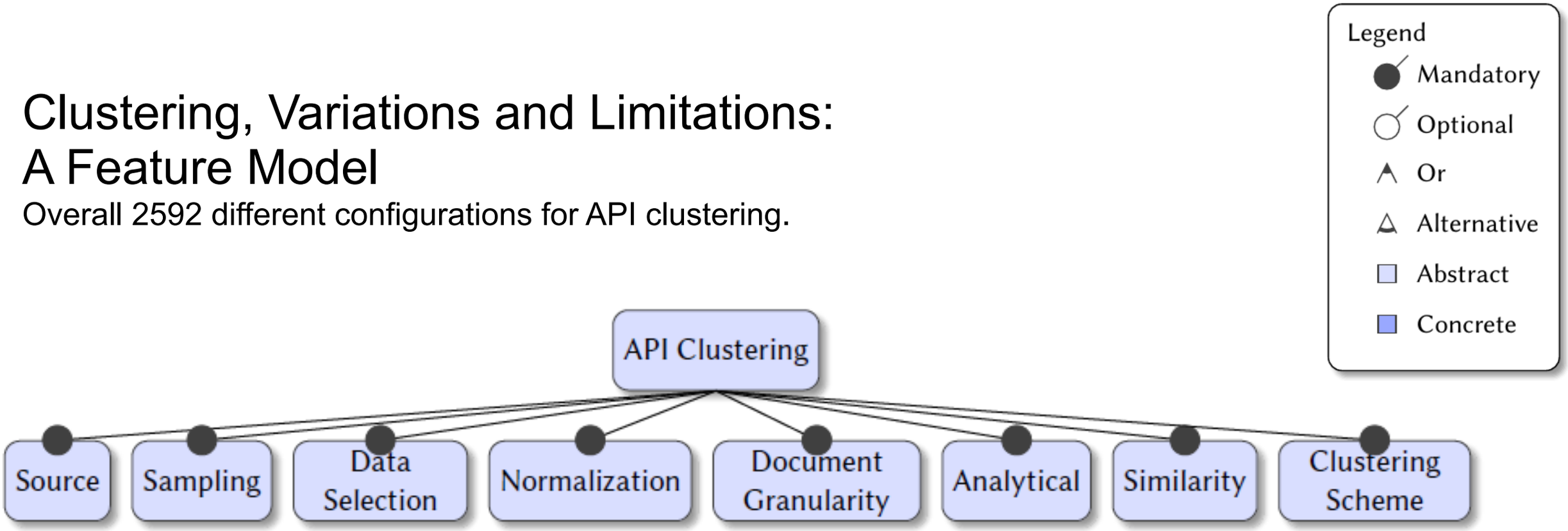
Clustering, Variations and Limitations

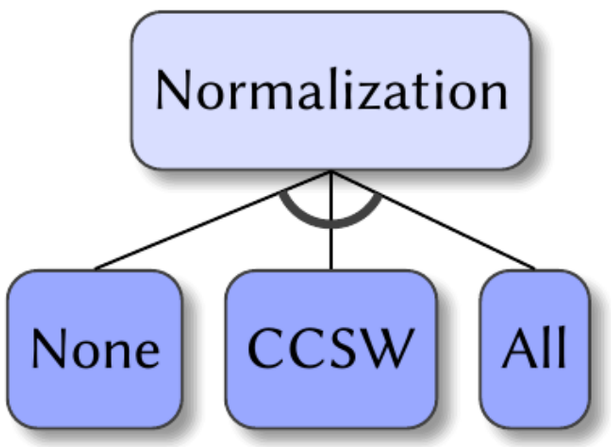
Possible reasons for an improper clustering.

A. APIs, Packages or Classes? **(Document Granularity**
B. Public/private identifiers? **(Data Selection)**
C. Stemming vs. no stemming? **(Normalization)**
D. Stop-words? **(Normalization)**
E. Camel-case splitting? **(Normalization)**
F. LDA, LSI or none? **(Analytical)**
G. Cosine vs. other similarity functions? **(Similarity)**
H. Average/Complete/Single - linkage **(Linkage Scheme)**

Clustering, Variations and Limitations: A Feature Model

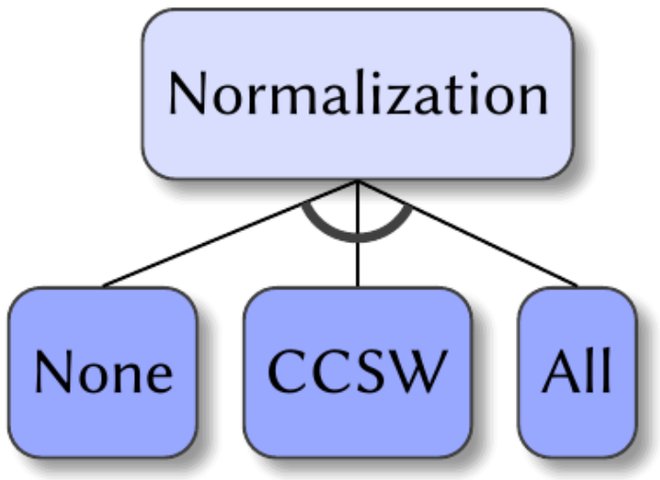
Overall 2592 different configurations for API clustering.

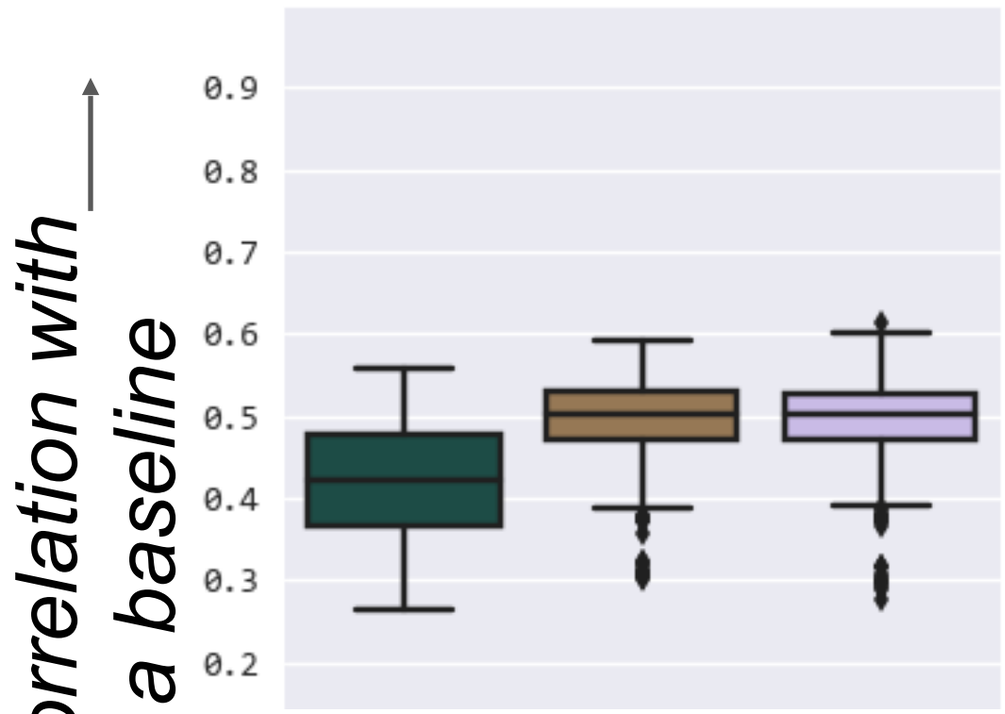




Clustering, Variations and Limitations: Systematic Correlation Checks

Depicting the correlation of all configuration sharing a particular Feature.





Conclusion

A. The interactive **Visualization** is actively used for exploring API Sets.
B. The deployed **Baselines** and the way of curation can be applied to different context.
C. More **Clustering, Variations and Limitations** by extending the feature model and using SBSE techniques (Search Based Software Engineering).

End

Classification of APIs by Hierarchical Clustering
https://github.com/softlang/apiclustering

Johannes Härtel - University of Koblenz - johanneshaertel@uni-koblenz.de

Unpublished work

Joint API usage

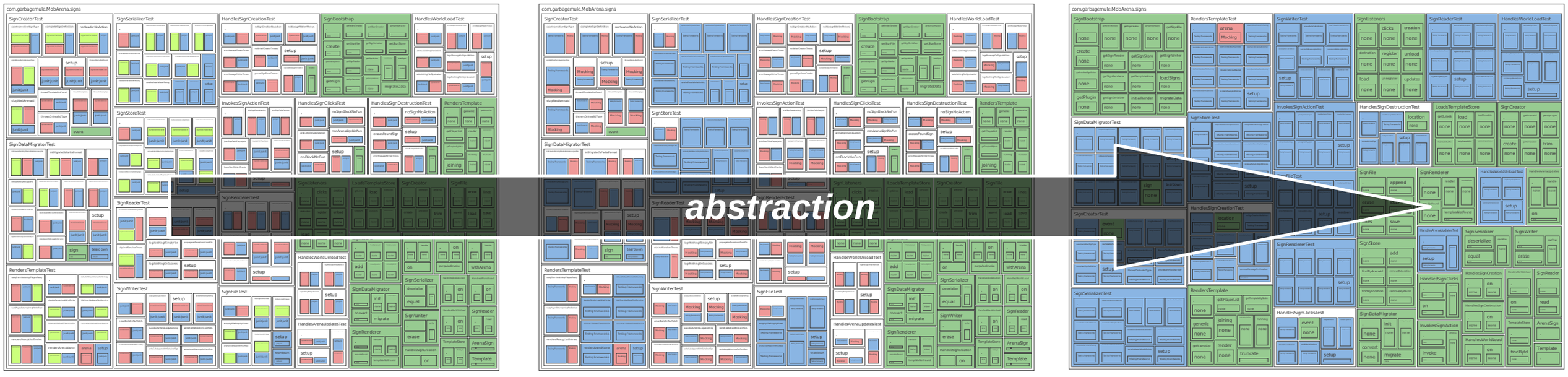
An exercise in causality

Philipp Seifer, Katharina Gorjatschev, and Ralf Lämmel
Uni Koblenz, SoftLang Team

Source image: https://pixabay.com/illustrations/the-fourth-dimension-fractal-noise-2713919/

Motivation

- Comprehension of the **purpose** of source-code scopes *(methods, classes et al.)*
- Via **abstraction** to API classifications



API Classification

- Semantic hints with respect to the purpose of source-code
- e.g., by clustering [1]
- e.g., via user defined classifications
  - *Maven Central Repository* (MCR) **categories** (e.g., Test Frameworks) and **tags** (e.g., testing, matching)

[1] Johannes Härtel, Hakan Aksu, and Ralf Lämmel. *Classification of APIs by hierarchical clustering*. In: Proc. of Conference on Program Comprehension, ICPC. ACM, 2018, pp. 233–243

Related Work

- Topic modeling for source code
- Feature location
- . . .

Challenges

Challenges

Can we aid comprehension of source-code (scopes) by utilizing API classifications?

- C1 – **META**
- C2 – **IMPL**
- C3 – **EXPL**

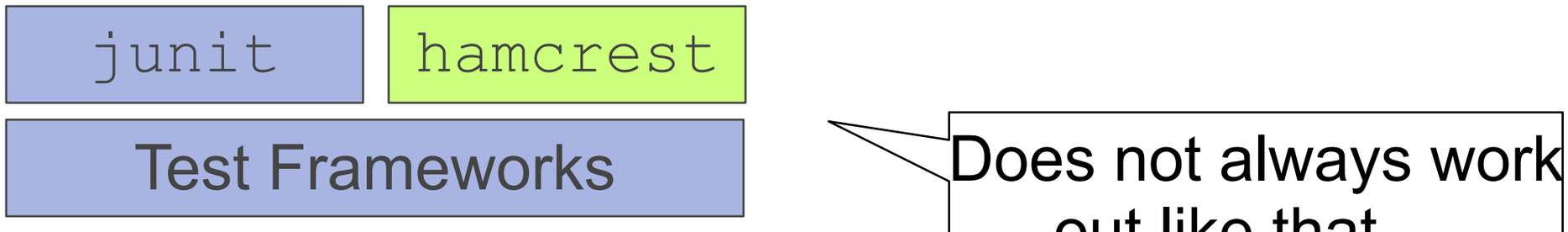
Challenge 1 – **META** (Example)

Scenario

A class uses multiple testing APIs (e.g., `junit` and `hamcrest`).

Abstraction

We visualize this scope using the *Testing Frameworks* MCR category.



Challenge 1 – **META**

How can we abstract from usage of specific APIs in source-code scopes?

- *idea* - utilization of API classifications instead of concrete APIs
- e.g., utilizing **meta**data such as MCR cateogries or MCR tags

Challenge 1 – **META**

- How can this abstraction be utilized to aid source-code comprehension?
  - e.g., source-code visualization, IDE tooling
- Is this abstraction effective in aiding source-code comprehension?
  - Validation: user studies, MSR techniques

Challenge 1 – **META** (Problem)

Scenario

A class uses testing (`junit`) and mocking (`mockito`) APIs.

Abstraction

We visualize this scope using the *Testing Frameworks* and *Mocking* MCR categories.

… can we somehow utilize other relationships?

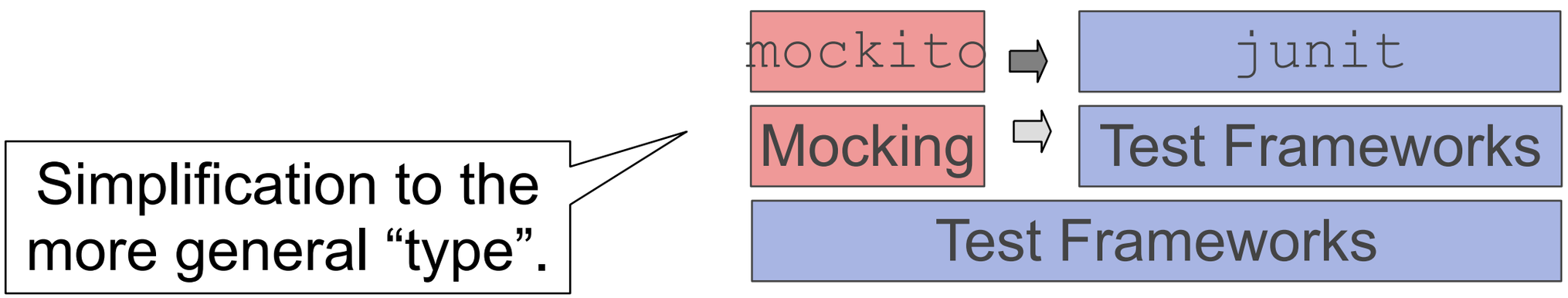
Challenge 2 – **IMPL** (Example)

Scenario

A class uses `junit` and `mockito`. We *identify* that `mockito` implies `junit`.

Abstraction

We transfer the implication to the category level (option 1/2).



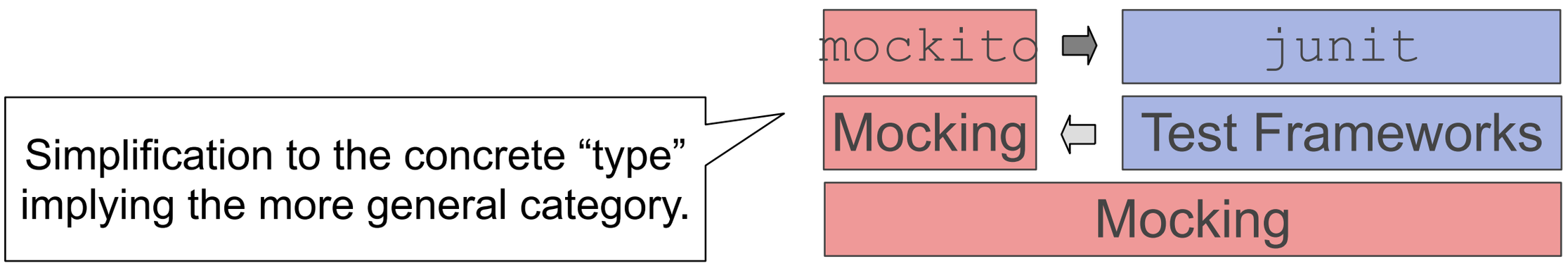
Challenge 2 – **IMPL** (Example)

Scenario

A class uses `junit` and `mockito`. We *identify* that `mockito` implies `junit`.

Abstraction

We transfer the implication to the category level (option 2/2).



Challenge 2 – **IMPL**

What are the effects of combined API usage on understanding the purpose of scopes?

- Scopes involve several APIs in different categories
- These APIs are in certain relationships
  - Straightforward relationship: API **impl**ies another, **A ⇒ B**

Challenge 2 – **IMPL**

- How can we utilize the *implication* relationship between APIs to abstract on the level of API categories?
  - How can the implication be transferred to the API category level?

Challenge 2 – **IMPL**

- What other kinds of implications exist?
  - e.g., conditional implication
  - Utilization of MSR techniques to comprehensibly identify existing relationships
- How can these implications be transferred to the category level?
  - Here, the answer may be even less straightforward!

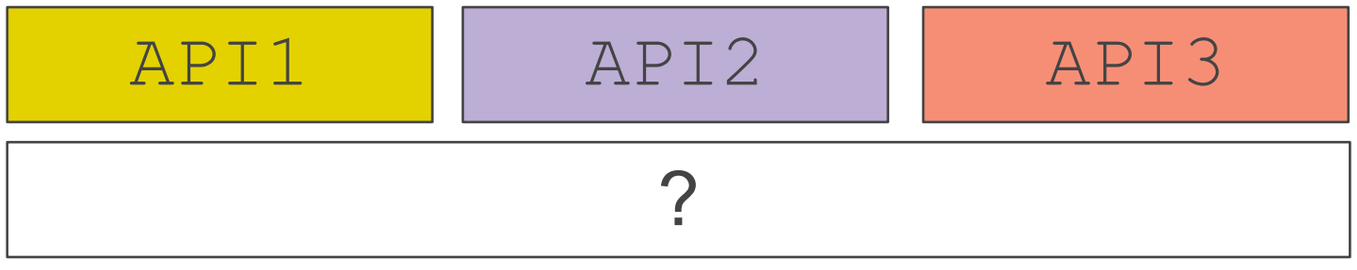
Challenge 2 – **IMPL** (Problem)

Scenario

A class uses API1, API2 and API3. There are no implications.

Abstraction

Can we still somehow abstract on the category level?



Challenge 3 – **EXPL** (Example)

Scenario

A class uses API1, API2 and API3. API1 dominates this scope.

Abstraction

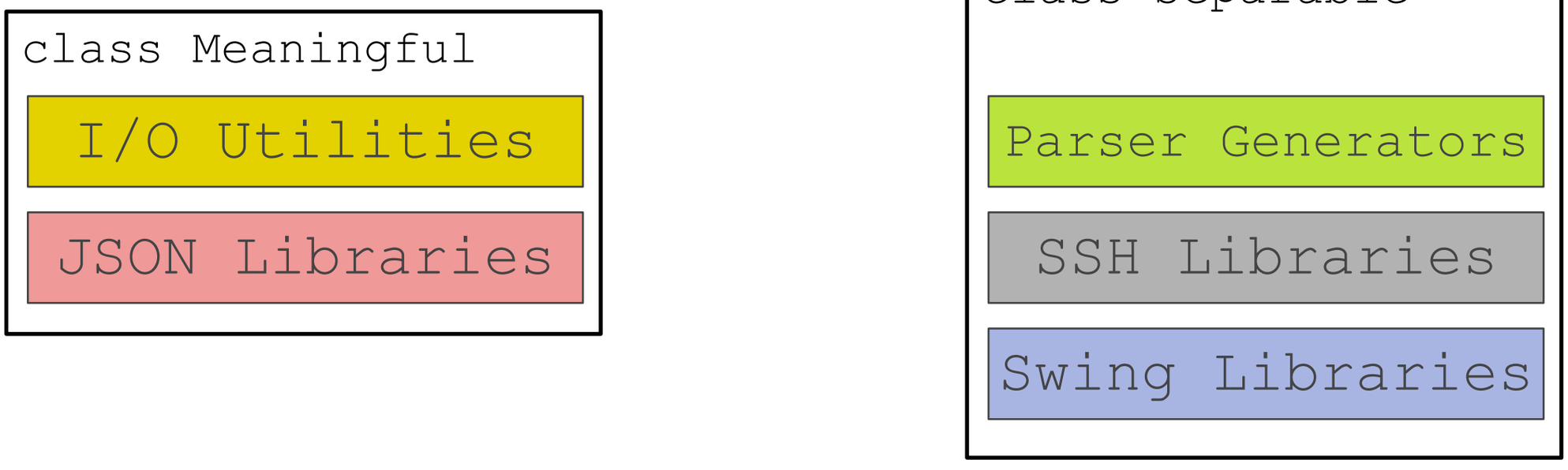
We use only the category of the dominant API.

Challenge 3 – **EXPL**

Can API categories be combined in a meaningful manner?

- Additional relationships between APIs exist
  - Several APIs contribute to purpose of scope, there may not always be implications
  - Such API usage may be classified as, e.g., separable or meaningfully combined

Combined API Usage



Challenge 3 – **EXPL**

- How can combined API use be **exp**lained in the general case?
  - Can scopes be considered dominated by one API category?
  - Can categories be combined based on weighted influences of component categories?
- Is this abstraction (still) helpful in comprehending the purpose of scopes?

Preliminary Study

Preliminary Study

- Goal: Visualization of source-code utilizing MCR categories (**C1-META**)
- Goal: Identifying implications in combined API useage and reducing complexity of visualization (**C2-IMPL**)

- Data set: **3778** Java repositories on GitHub
  - 100+ stars
  - 100+ commits
  - 2+ contributors
  - existing POM file (i.e., using Maven)

Methodology – Data Collection

- **Repositories** GitHub API with quality filters mentioned on previous slide

- **POM-based Dependencies**
  - Repositories with **2+** valid dependencies (in POM file(s))
  - Validity: Respective dependency can be resolved

Methodology – Repo processing

- Parsing and symbol resolution using [3] (requiring `jars` for all dependencies)
- Tagging of each method (transitively classes and packages) with
  - Used classes and their respective API (if it exists)
    ➜ Set of **actual API dependencies**
  - (indirectly therefore MCR metadata)

Possibly a subset of POM-based dependencies.

[3] `https://javaparser.org` (including `JavaSymbolSolver`)

Methodology – Combined API Usage

- **Goal** Identification of (candidate) implications between API **A** and API **B**
- For a given project P we have
  - T(A) Number of scopes of type *T* using API **A**
  - T(B) Number of scopes of type *T* using API **B**
  - T(A&B) Number of scopes *T* using APIs **A** and **B**
- $T(\mathbf{A} \Rightarrow \mathbf{B}) = \dfrac{T(A\&B)}{T(A)}$

Methodology – Combined API Usage

API Usages				API Usages - Percentage		
T	**T(JUnit)**	**T(Hamcrest)**	**T(Both)**	**T**	**T(J ⇒ H)**	**T(H ⇒ J)**
method	23208	5944	5550	**method**	0.24	**0.93**
class	3488	1104	1021	**class**	0.29	**0.92**
package	847	341	320	**package**	0.38	**0.94**

Candidate: Hamcrest implies JUnit

Methodology – Combined API Usage

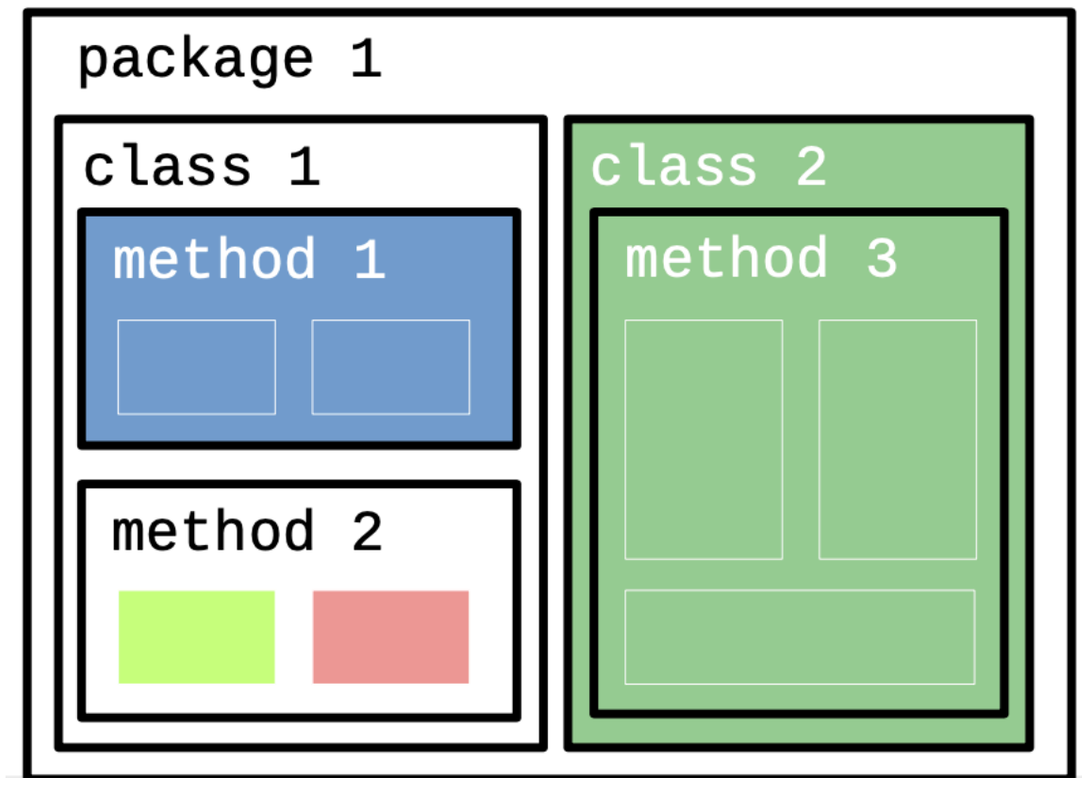
- Set of candidate API implications
- Manual verification by labeling a sample as
  - Dependent
  - Independent (**meaningfully combined** or **separable**)

- **Separable** APIs are used in the same scope T, but in fact could easily be used in two scopes T1 and T2, where each scope uses exactly one API.

- **Meaningfully Combined** APIs are used in the same scope T, which could not be split easily. Yet, there exists no dependency between the APIs, i.e., they may frequently be used independently as well.

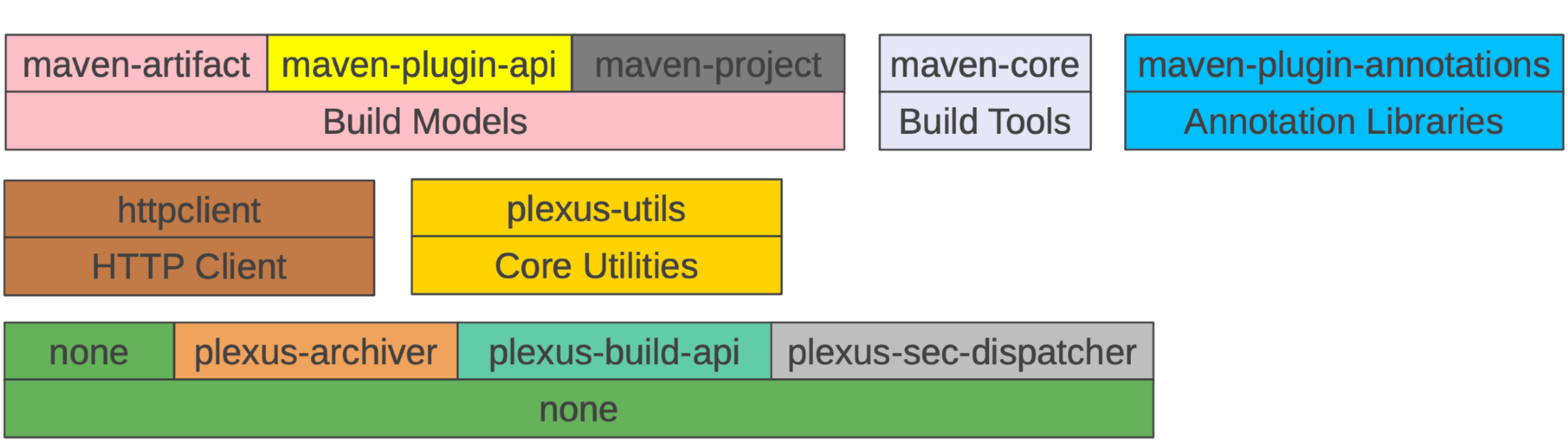
Preliminary Study – Visualization

- visualization as treemaps: packages, classes and methods
- color methods (classes, packages) based on APIs or their MCR categories



Preliminary Study – Example **META**

- Example project [5]
- APIs:



[5] https://github.com/maven-download-plugin/maven-download-plugin/


API Level → Category Level
maven-artifact
maven-plugin-api
maven-project
maven-core
maven-plugin-annotations
plexus-utils
httpclient
none

API Level → **Category Level**

maven-artifact | maven-plugin-api | maven-project
Build Models

maven-core
Build Tools

maven-plugin-annotations
Annotation Libraries

plexus-utils
Core Utilities

httpclient
HTTP Client

none
none

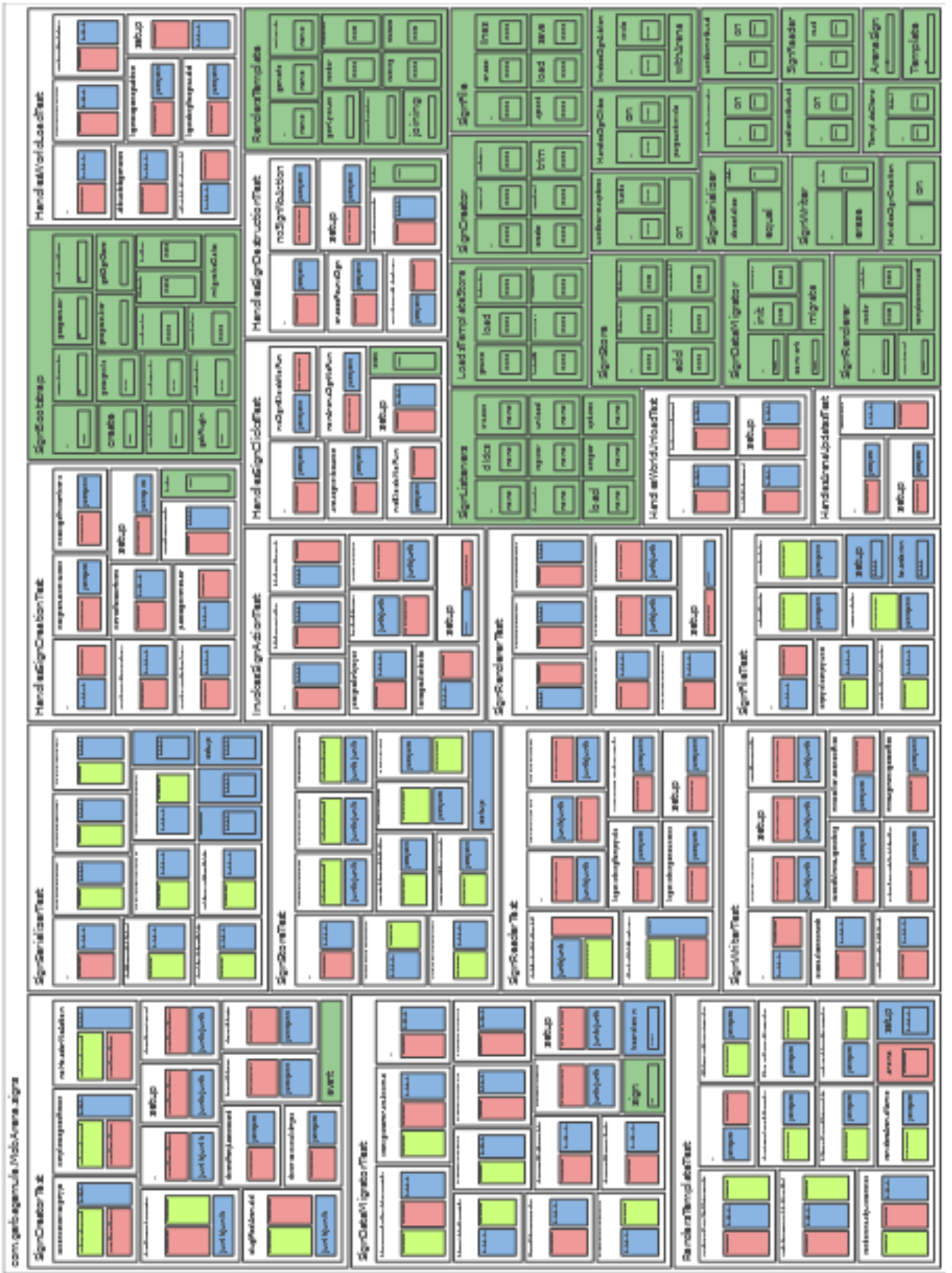
Preliminary Study – Example **IMPL**

- Example project `garbagemule/MobArena` [2]
- Used APIs 

- Identified implications 

[2] `https://github.com/garbagemule/MobArena` (package `signs`)

API Level → Category Level → Categories + Implications



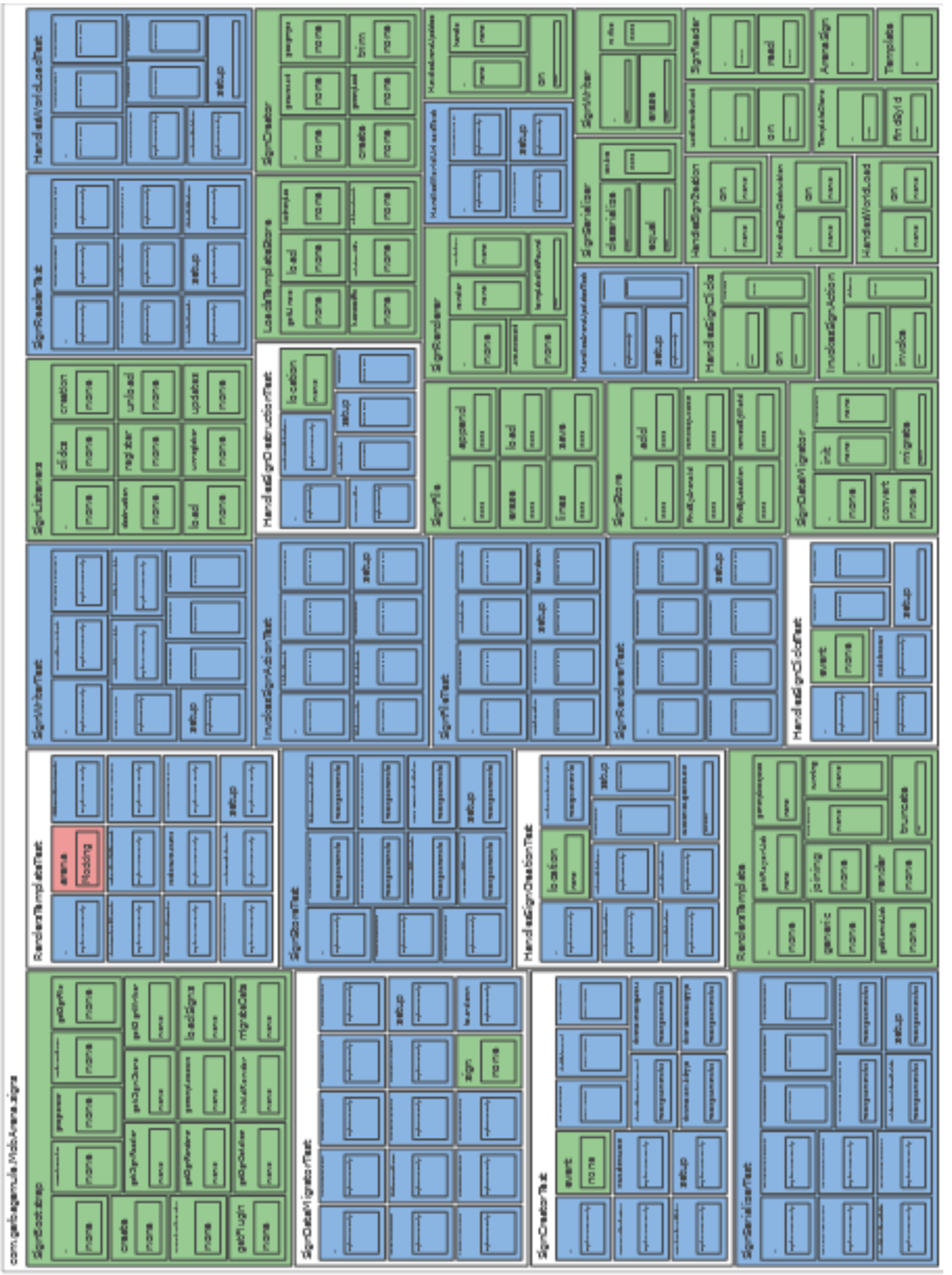
none mockito junit hamcrest

API Level → **Category Level** → Categories + Implications

none | none | mockito | junit | hamcrest

none | Mocking | Test Frameworks

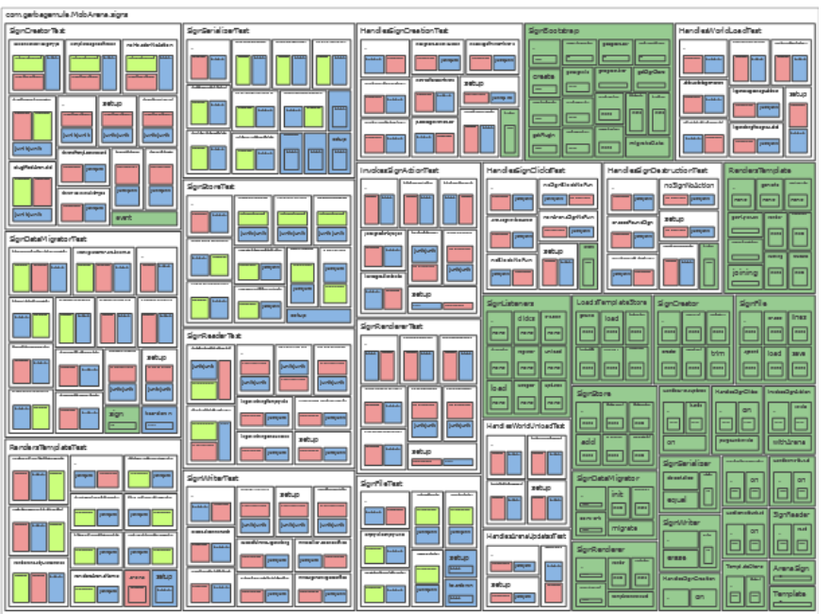
API Level → Category Level → Categories + **Implications**

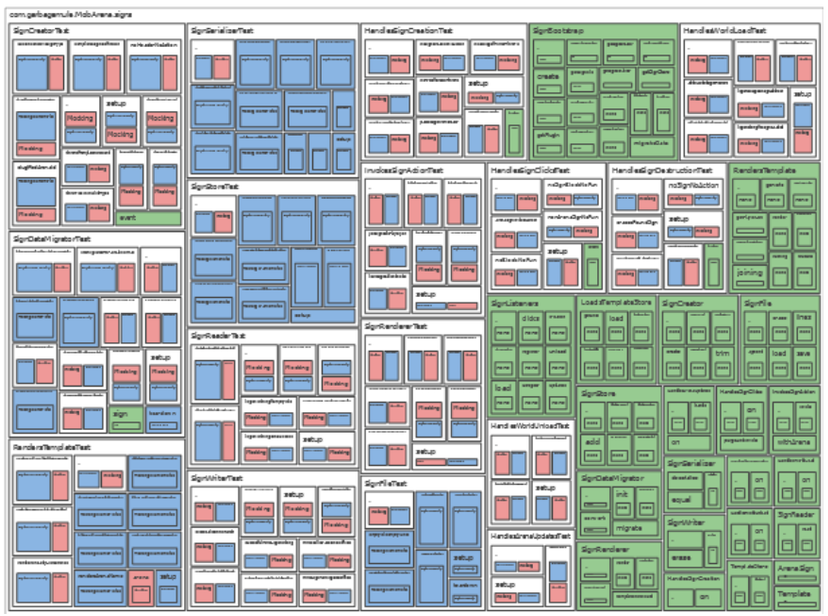


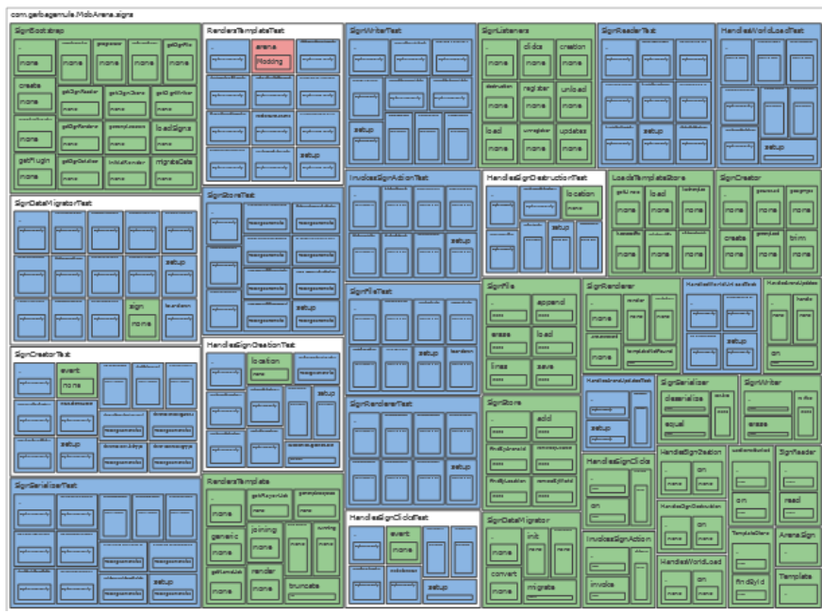
none	mockito	junit	hamcrest
none	Mocking	Test Frameworks	
none	Test Frameworks		

Summary

- Abstracting to categories simplifies visualization
- Utilizing the implication relationship further simplifies visualization







Preliminary Study – Implications

- Examples
  - org.hamcrest:**hamcrest-all** ⇒ junit:**junit**
  - org.mockito:**mockito-core** ⇒ junit:**junit**
  - org.apache.lucene:**lucene-analyzers-common** ⇒ org.apache.lucene:**lucene-core**

[2] https://github.com/garbagemule/MobArena (package signs)

Preliminary Study – Results

- We were able to find examples of the **implication** relationship between APIs
- We utilized API classifications (MCR categories) as well as implications between APIs to **visualize** source-code
- Abstraction along these categories and implication seem to **aid** visual source-code comprehension

---

Source code available @ *https://github.com/gorjatschev/applying-apis*

Future Work

- Systematic identification of implication relationships
- Classification of API usage (e.g., separable or meaningfully combined)
- Definition of dominance of an API w.r.t to other APIs in the same scope

Future work — Automatic Detection and Systematic Identification of Implications

- Hypothesis - There are various kinds of implications, e.g., conditional implications.
- Motivation - We can utilize implications (as shown in preliminary study); automatic identification of implication scenarios relevant for that; identification of additional kinds of implications relevant, as they may lead to different insights.
- Approach - MSR techniques; source-code analysis; perhaps analysis of data flow between APIs.
- Challenges - Not clear which implication relationships exist.
- Current status - Semi-automatic analysis based on co-occurrence of APIs.

Future work — Definition of dominance of an API w.r.t to other APIs in the same scope

- Hypothesis - Some APIs are more relevant for understanding the purpose of a scope in source code, than others.
- Motivation - Enables abstracted view of source-code based on only those APIs that are "relevant".
- Approach - Weighted contribution...
- Challenges - ...but can not be based only on number of API calls. Requires some measure for "impact" on the respective scope.
- Current status - We only looked at clear implications between APIs.

Future work — Empirical Validation: Comprehension of the Purpose of Source-Code Scopes

- Hypothesis - Abstraction to API categories (resp. including utilization of implications) allows for comprehension of the purpose of software components in less time.
- Motivation - (Validation)
- Approach - Empirical user study.
- Challenges - Design of plausible tasks without bias; difficult to base tasks on real-world problems.
- Current status - TBD

References

[0] Katharina Gorjatschev. *Applying API Categories to the Abstractions Using APIs*. Master Thesis, University of Koblenz-Landau. 2022

[1] Johannes Härtel, Hakan Aksu, and Ralf Lämmel. *Classification of APIs by hierarchical cluster-*
ing. In: Proc. of Conference on Program Comprehension, ICPC. ACM, 2018, pp. 233–243

[2] `https://github.com/garbagemule/MobArena` (package `signs`)

[3] `https://javaparser.org` (including `JavaSymbolSolver`)

[4] `https://github.com/gorjatschev/applying-apis`

Unpublished work

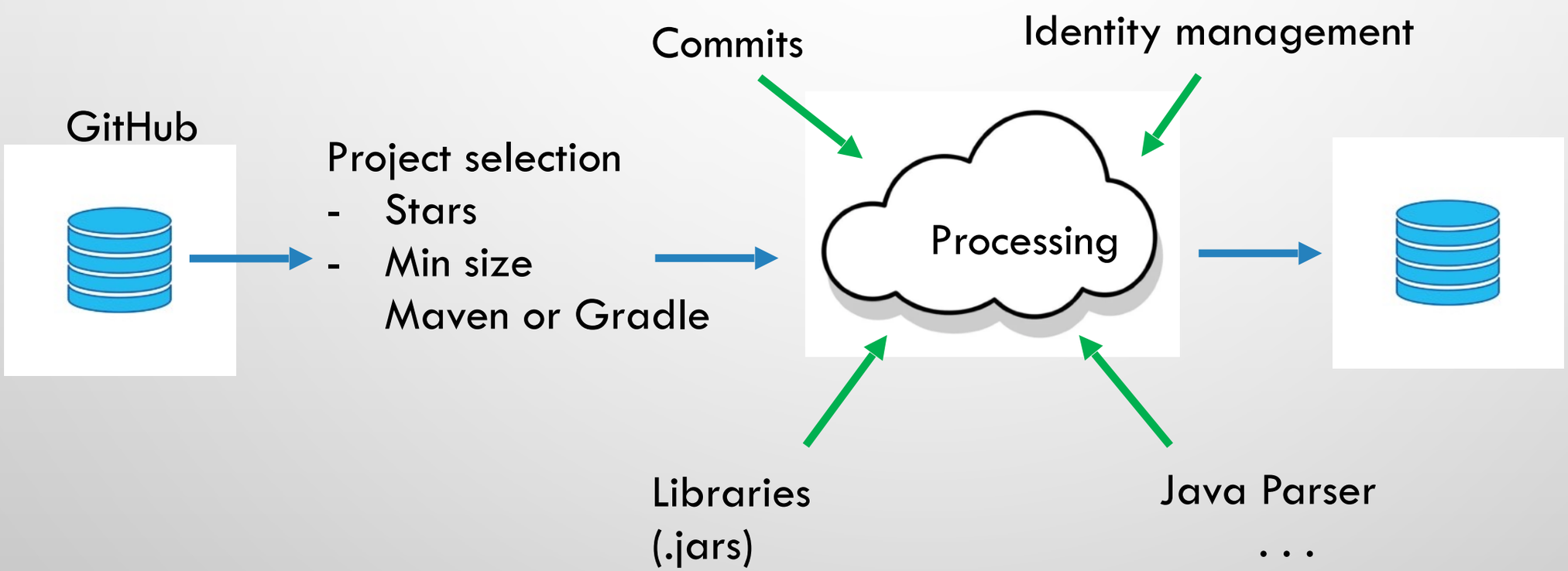
API developer profiles

An exercise in hypothesis building and validation

Hakan Aksu and Ralf Lämmel
Uni Koblenz, SoftLang Team

Source image: https://pixabay.com/illustrations/the-fourth-dimension-fractal-noise-2713919/

DATA EXTRACTION AND PROCESSING



API-USAGE DATA AS VECTORS

API usage developer 1
api1 500
api2 300
api3 150
…

API usage developer 2
api2 200
api3 80
api1 30
…

API usage developer 3
api3 3000
api2 2000
api1 500
…

Different types of vectors:
— Per commit
— For all commits
— Commits in a time window
— Some sampling

DRAFT HYPOTHESES
(NOT YET PROPERLY FALSIFIABLE)
(WANTED MORE — PLEASE PARTICIPATE!)

- **Triviality** hypothesis: The more a developer contributes to the project, the "bigger" API usage. (Several terms require clarification!)
- **Diversity** hypothesis: Developers can be differentiated based on their API usage.
- **Similarity** hypothesis: Developers may similar to other developers in terms of their relative API usage.
- **Prediction** hypothesis: API usage a developer can be predicted for the immediate future.

Metrics

- AR: API-References → Number of API references (API usage)
- DAR: Distinct AR → Number of API references (API usage)
- LOC: Lines of Code → Changed LOC
- TD: Time Difference → … between first and last commit
- CF: Changed Files → Number of changed files
- CP: Changed Packages → Number of changed packages (folders)
- DCF, DCP: Distinct → …
- Com: Commits → Number of Commits

Triviality hypothesis: The more a developer contributes to the project, the “bigger” API usage.

EVALUATION DETAILS

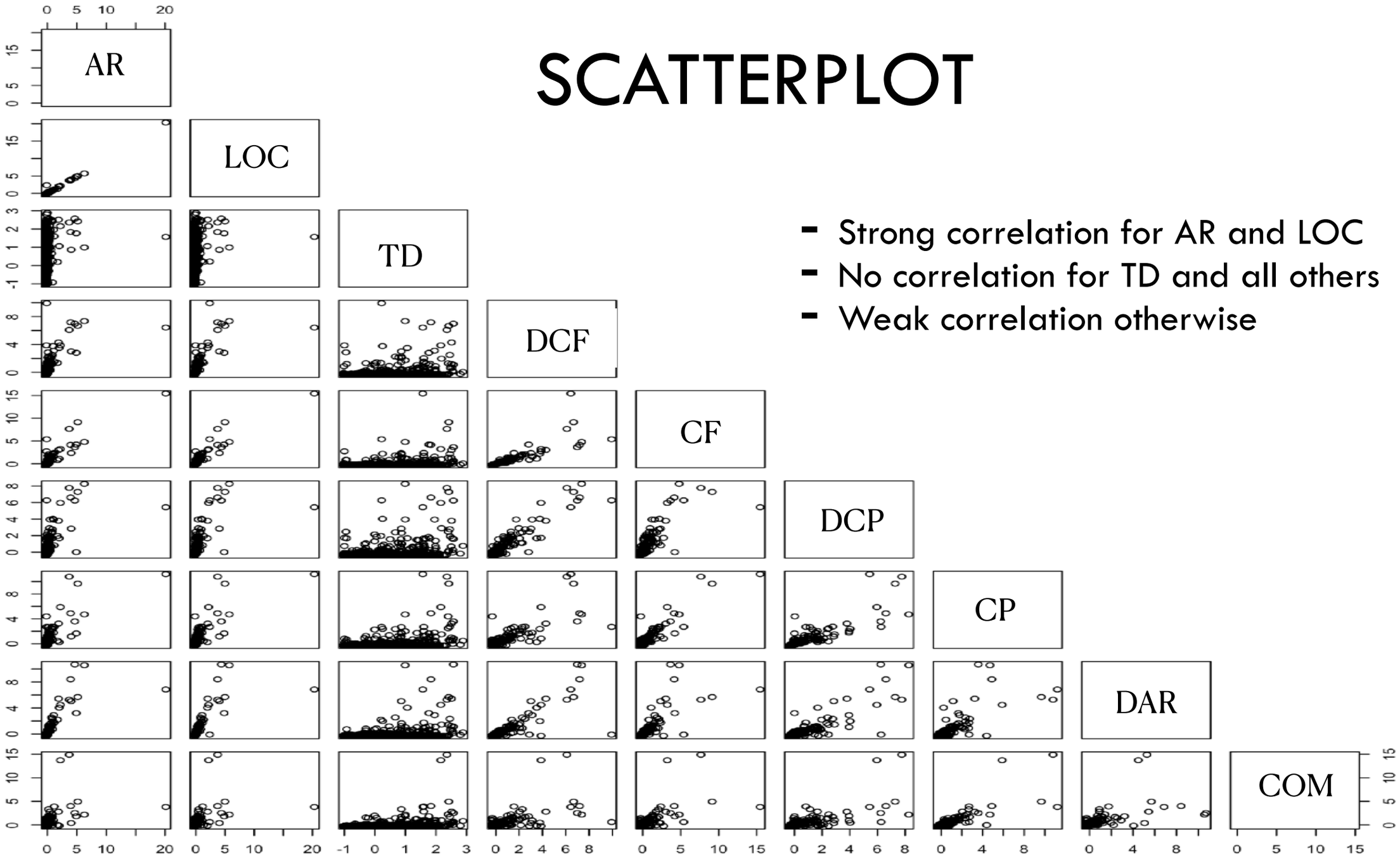
- 5 Github projects
  - Kafka, RxJava, Spring-boot, Spring-framework, Elasticsearch
- 600 developers
  - Top100 developers per project
  - Top 200 developers for Elasticsearch (given the size of the project)

Triviality hypothesis: The more a developer contributes to the project, the “bigger” API usage.

Analyses

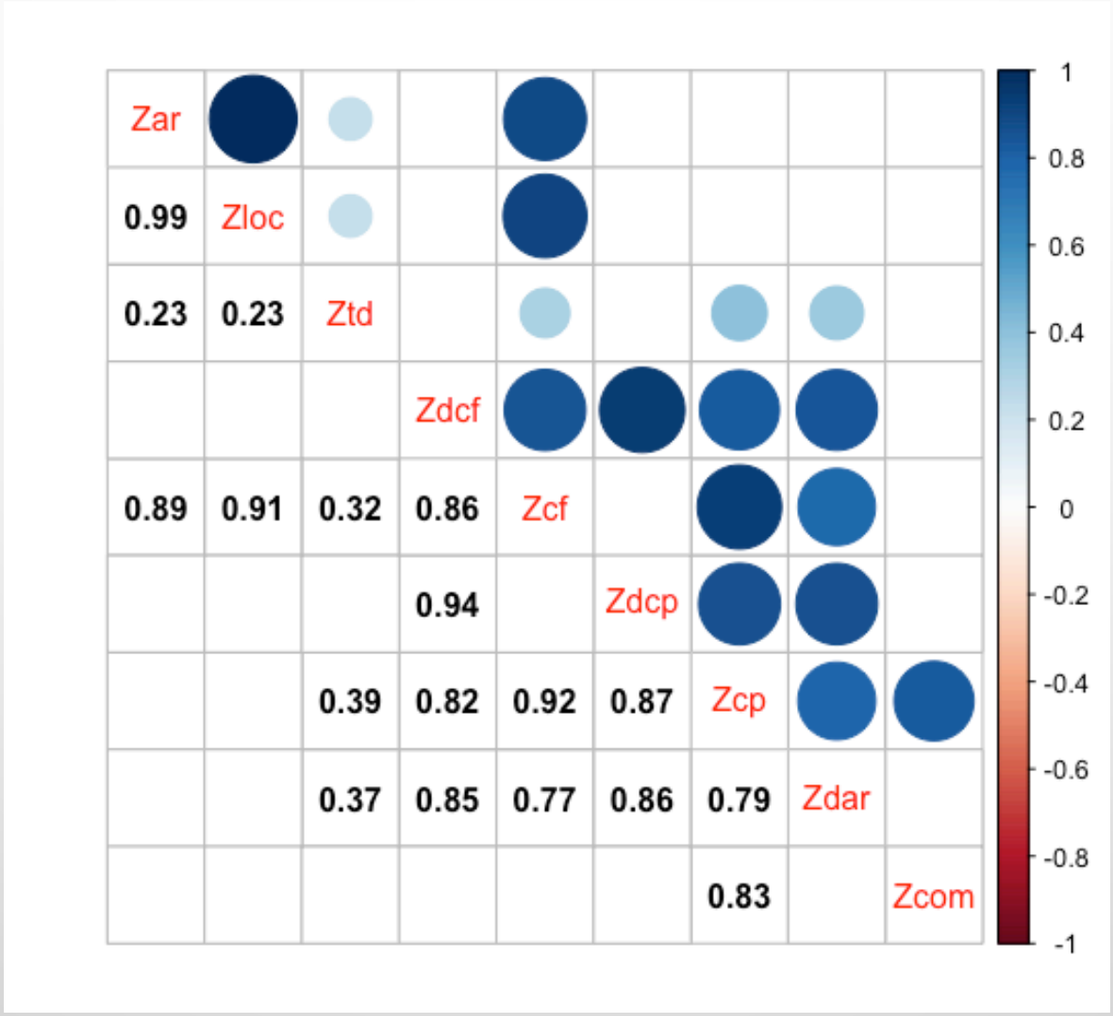
- Scatter plot
- Pearson correlation coefficients
- Multi-linear regression
- What else makes sense here?

SCATTERPLOT



- Strong correlation for AR and LOC
- No correlation for TD and all others
- Weak correlation otherwise

PEARSON CORRELATION COEFFICIENT



MULTI-LINEAR REGRESSION FOR AR

Call: lm(formula = Zar ~ Zloc + Ztd + Zdcf + Zcf + Zdcp + Zcp + Zcom, data = data_csv)

Coefficients:

	Estimate	Std. Error	t value	Pr(>|t|)	
(Intercept)	-3.703e-16	3.684e-03	0.000	1.00000	
Zloc	1.093e+00	1.163e-02	94.000	< 2e-16	***
Ztd	8.127e-04	4.089e-03	0.199	0.84253	
Zdcf	-1.164e-01	1.601e-02	-7.268	1.15e-12	***
Zcf	-1.416e-01	2.326e-02	-6.088	2.06e-09	***
Zdcp	9.842e-02	1.527e-02	6.446	2.39e-10	***
Zcp	5.667e-02	1.718e-02	3.298	0.00103	**
Zcom	6.276e-03	7.451e-03	0.842	0.39993	

Signif. codes: 0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Shouldn’t we also do this for DAR?

Also — and I am not saying this — is there any hypothesis that needs these coefficients?

Similarity hypothesis: Developers may similar to other developers in terms of their relative API usage.

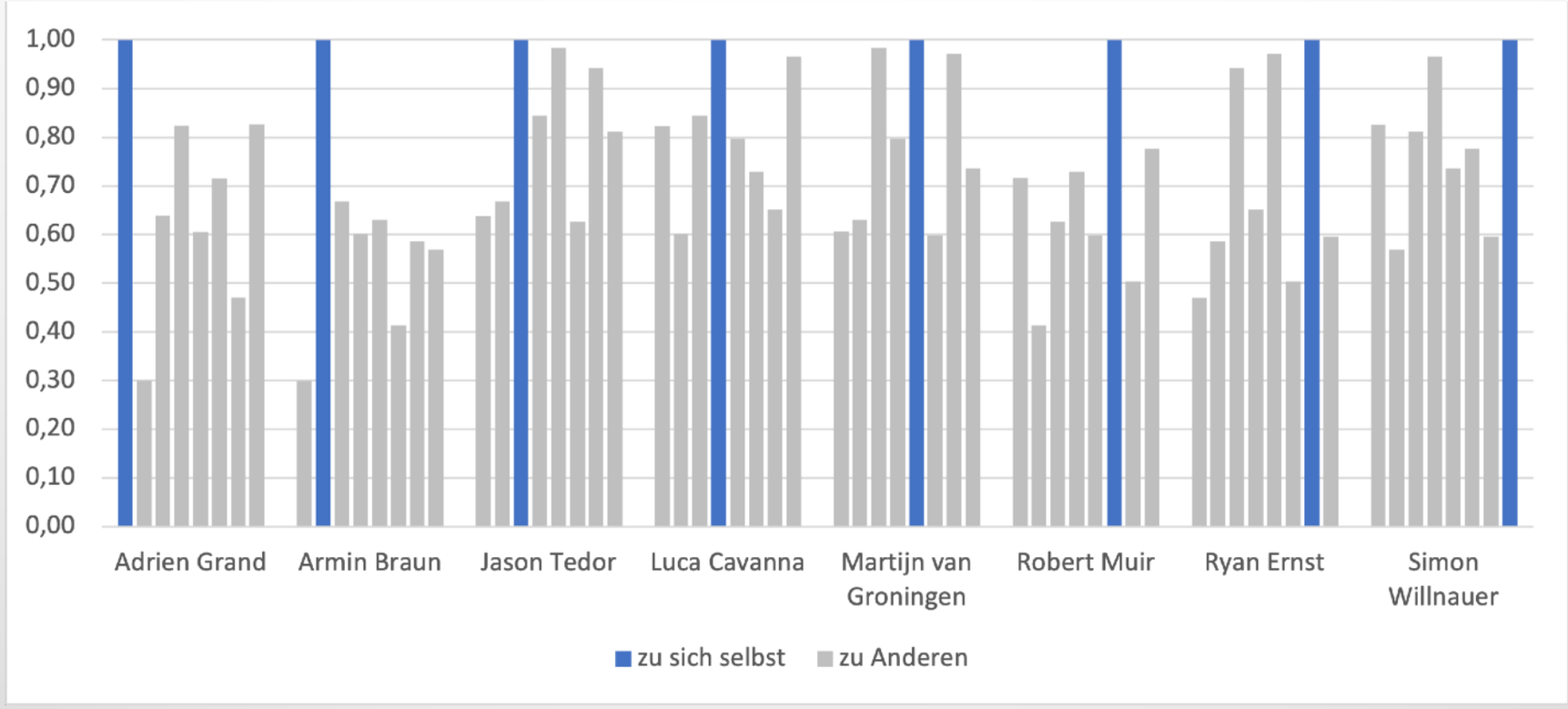
Evaluation

- Top 8 developers from „Elasticsearch“
- Build API vectors over the complete time for each developer.
- Compare the developers for similarity.

COSINE

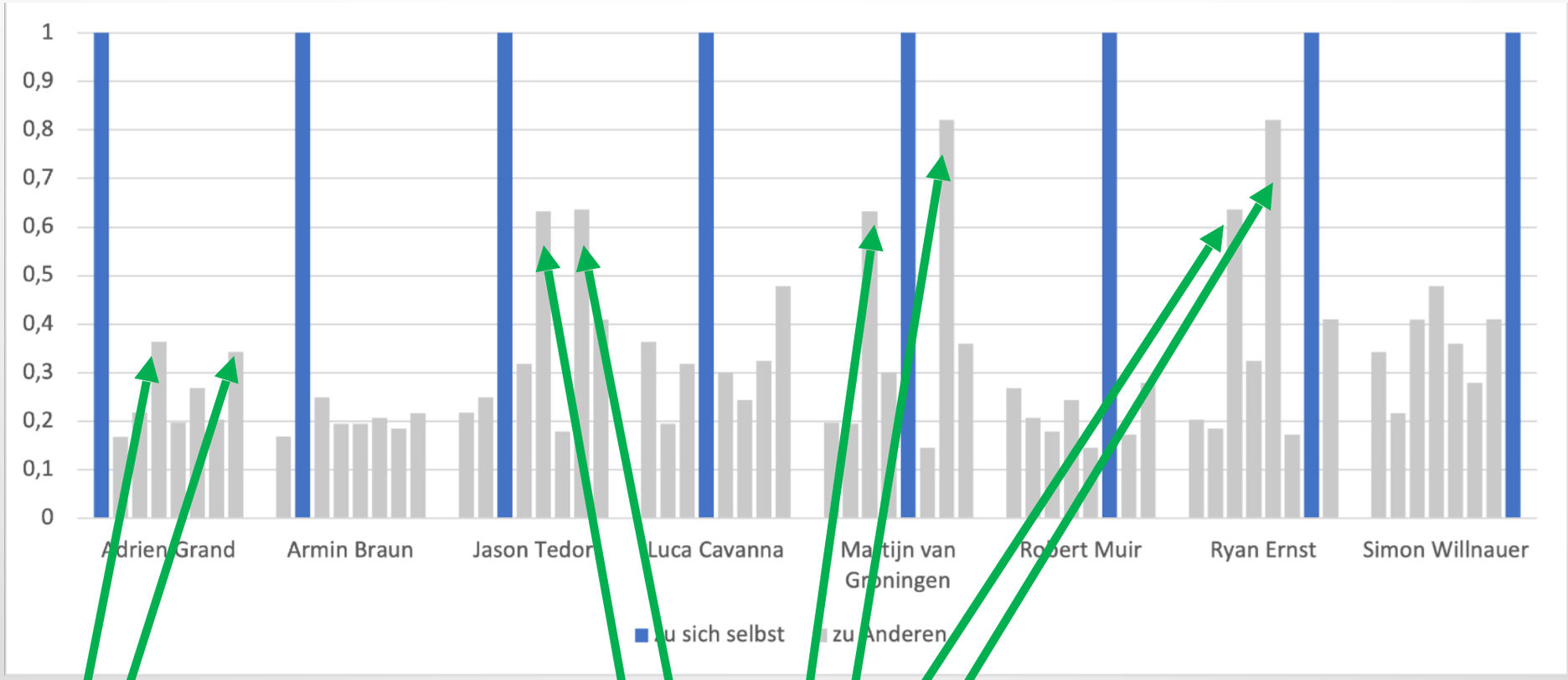
	Adrien Grand	Armin Braun	Jason Tedor	Luca Cavanna	Martijn van Groningen	Robert Muir	Ryan Ernst	Simon Willnauer
Adrien Grand	1,00	0,30	0,64	0,82	0,61	0,72	0,47	0,83
Armin Braun	0,30	1,00	0,67	0,60	0,63	0,41	0,59	0,57
Jason Tedor	0,64	0,67	1,00	0,85	0,98	0,63	0,94	0,81
Luca Cavanna	0,82	0,60	0,85	1,00	0,80	0,73	0,65	0,97
Martijn van Groningen	0,61	0,63	0,98	0,80	1,00	0,60	0,97	0,74
Robert Muir	0,72	0,41	0,63	0,73	0,60	1,00	0,50	0,78
Ryan Ernst	0,47	0,59	0,94	0,65	0,97	0,50	1,00	0,60
Simon Willnauer	0,83	0,57	0,81	0,97	0,74	0,78	0,60	1,00

COSINE



$$\cos(\theta) = \frac{\mathbf{a} \cdot \mathbf{b}}{\|\mathbf{a}\| \, \|\mathbf{b}\|} = \frac{\sum_{i=1}^{n} a_i \cdot b_i}{\sqrt{\sum_{i=1}^{n} (a_i)^2} \cdot \sqrt{\sum_{i=1}^{n} (b_i)^2}}$$

JACCUARD



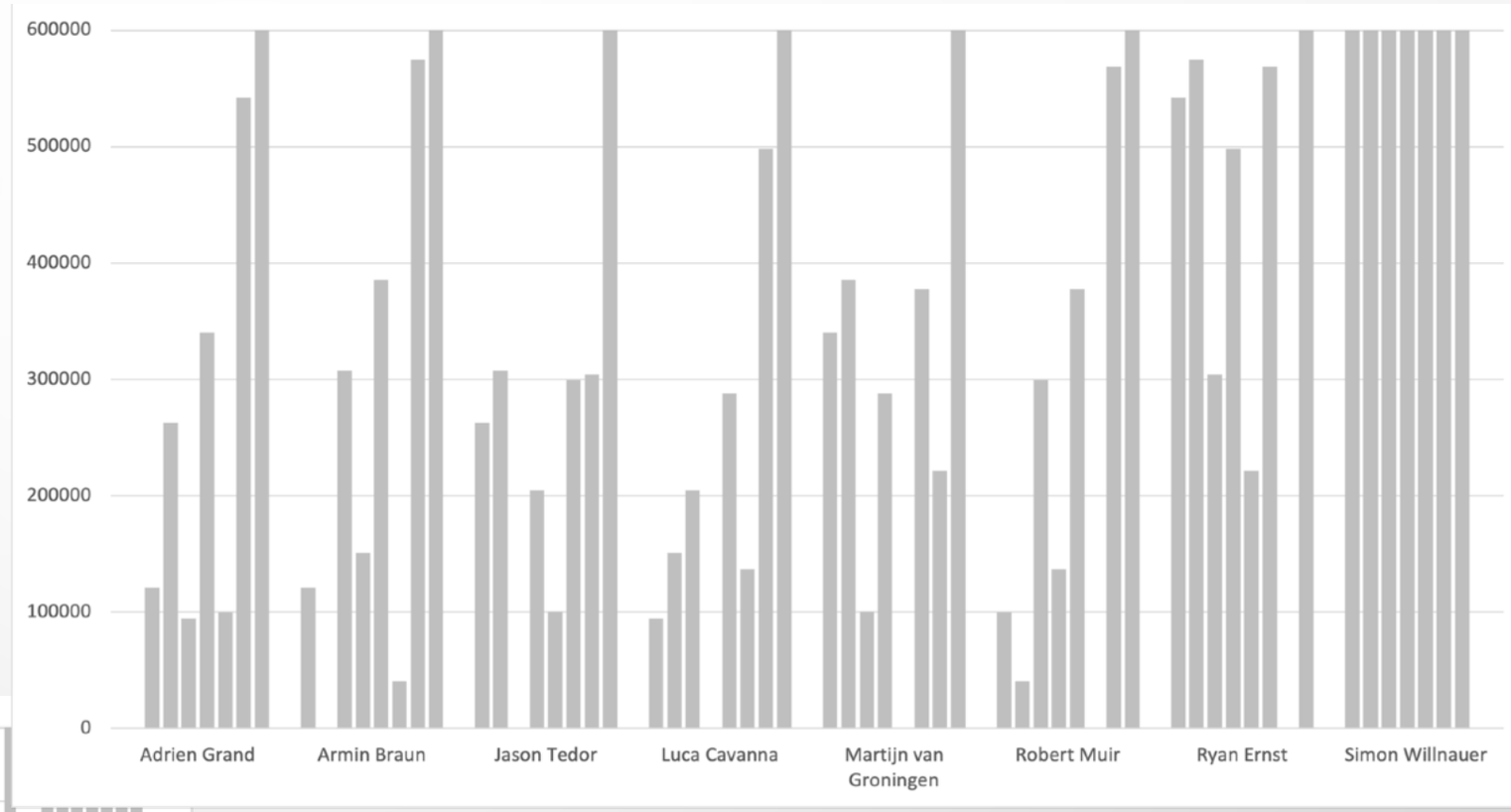
1,4,8 sim. per cosine

3,5,7 sim. per cosine

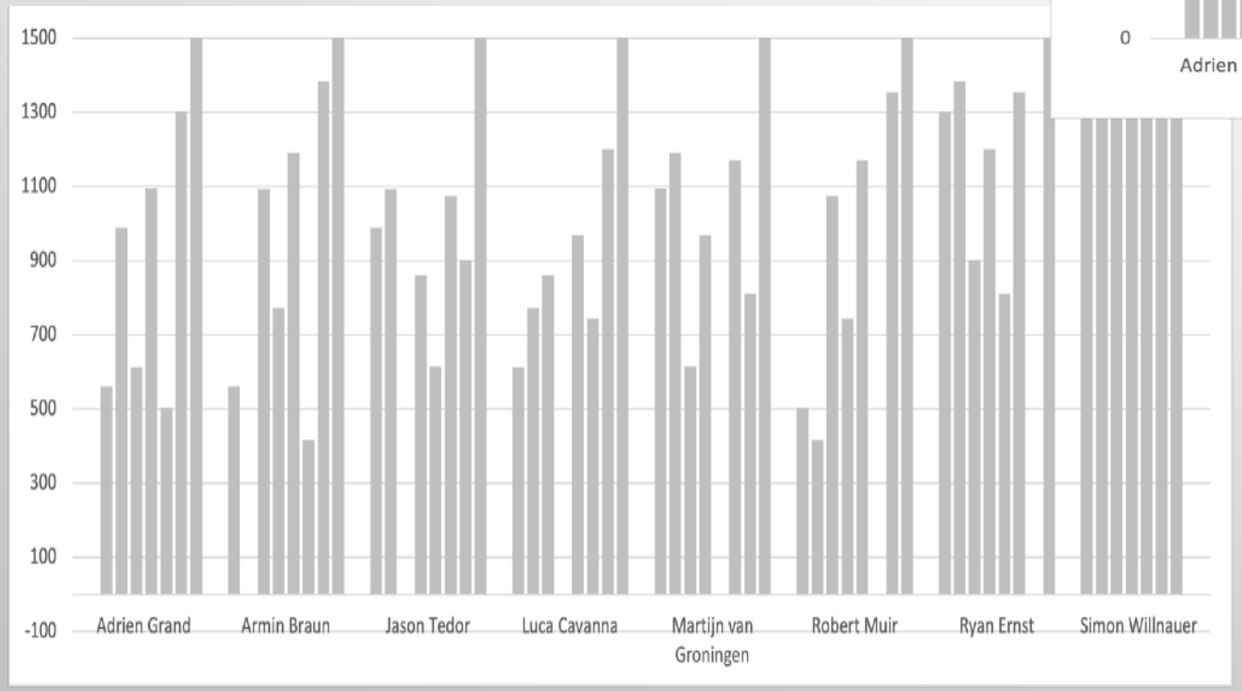
$$J(A,B) = \frac{|A \cap B|}{|A \cup B|}$$

DISTANCE

Euclidean



Manhatten



Discussion

- Cosine and Jaccard are similar.
- Cosine auto-normalizes.
- Jaccard ignores ranking order — plain sets are used.
- Absolute values have weight with distance approaches.
- Normalization may be preferable.

What hidden variables are involved?
Can we control them?

LET’S DISCUSS …