Ekaterina Pek

# Corpus-based Empirical Research in Software Engineering

Dissertation

Koblenz, October 2013

Department of Computer Science
Universität Koblenz-Landau

Ekaterina Pek

Corpus-based Empirical Research in Software Engineering

When someone is seeking, it happens quite easily that he only sees the thing that he is seeking; that he is unable to find anything, unable to absorb anything, because he is only thinking of the thing he is seeking, because he has a goal, because he is obsessed with his goal. Seeking means: to have a goal; but finding means: to be free, to be receptive, to have no goal.

Herman Hesse, Siddhartha[1]

[1] Translation by Hilda Rosner

# Acknowledgments

First of all, I am very grateful to my supervisor, Prof. Dr. Ralf Lämmel, for his guidance, support, and patience during these years. I am very glad that I had a chance to work with Ralf, learn from his experience and wisdom, and grow professionally under his supervision, as well as appreciate his friendly personality.

I am grateful to my co-authors: Jean-Marie Favre, Dragan Gašević, Rufus Linke, Jürgen Starek, and Andrei Varanovich, working with whom was a great intellectual pleasure.

No man is an island: I would like to thank my ex-boyfriend, Vladimir, for his great help and support during my PhD; my then-colleagues at the University of Koblenz for their company, thoughts, and time: my gratitude goes especially to Andrei Varanovich, Claudia Schon, and Sabine Hülstrunk.

I gratefully acknowledge the support of my work by the Rhineland Palatinate's Research Initiative 2008-2011, the ADAPT research focus, and the University of Koblenz-Landau.

August 2013 *Ekaterina Pek*

# Abstract

In the recent years, Software Engineering research has shown the rise of interest in the empirical studies. Such studies are often based on empirical evidence derived from *corpora*—collections of software artifacts. While there are established forms of carrying out empirical research (experiments, case studies, surveys, etc.), the common task of preparing the underlying collection of software artifacts is typically addressed in ad hoc manner.

In this thesis, by means of a literature survey we show how frequently Software Engineering research employs software corpora and using a developed classification scheme we discuss their characteristics. Addressing the lack of methodology, we suggest a method of corpus (re-)engineering and apply it to an existing collection of Java projects.

We report two extensive empirical studies, where we perform a broad and diverse range of analyses on the language for privacy preferences (P3P) and on object-oriented application programming interfaces (APIs). In both cases, we are driven by the data at hand—by the corpus itself—discovering the actual usage of the languages.

# Zusammenfassung

In den letzten Jahren gibt es im Bereich Software Engineering ein steigendes Interesse an empirischen Studien. Solche Studien stützen sich häufig auf empirische Daten aus Corpora—Sammlungen von Software-Artefakten. Während es etablierte Formen der Durchführung solcher Studien gibt, wie z.B. Experimente, Fallstudien und Umfragen, geschieht die Vorbereitung der zugrunde liegenden Sammlung von Software-Artefakten in der Regel ad hoc.

In der vorliegenden Arbeit wird mittels einer Literaturrecherche gezeigt, wie häufig die Forschung im Bereich Software Engineering Software Corpora benutzt. Es wird ein Klassifikationsschema entwickelt, um Eigenschaften von Corpora zu beschreiben und zu diskutieren. Es wird auch erstmals eine Methode des Corpus (Re-)Engineering entwickelt und auf eine bestehende Sammlung von Java-Projekten angewendet.

Die Arbeit legt zwei umfassende empirische Studien vor, in denen eine umfangreiche und breit angelegte Analysenreihe zu den Sprachen Privacy Preferences (P3P) und objektorientierte Programmierschnittstellen (APIs) durchgeführt wird. Beide Studien stützen sich allein auf die vorliegenden Daten der Corpora und decken dadurch die tatsächliche Nutzung der Sprachen auf.

# Contents

# 1 Introduction

In this chapter, with the means of a motivating example, we identify research areas of interest, briefly discuss their challenges, formulate research goals, devise the plan of attack, and, finally, outline the structure of the thesis.

## 1.1 Research Context

### 1.1.1 Motivating example

To motivate our research, we use an early example of an empirical study of FORTRAN programs [115] done by Knuth and his team in 1971. Knuth studied a sample of applications "in an attempt to discover quantitatively 'what programmers really do'." In other words, Knuth's study discovers the actual usage of the language (*as is*) in contrast to the recommended usage of the language (*as should be*). The goal of the study was to provide food for thought for compiler writers.

To collect a sample of FORTRAN applications, different strategies were used:

- "rummage in the waste-baskets and the recycling bins" (to obtain punched cards with programs);
- "probe randomly among the semi-protected files stored on disks looking for source text";
- finally, post a man by the card reader asking users for a copy of their decks.

Each of the methods was reported to be unsatisfactory in its own way: waste-baskets contained undebugged programs, asking users for a copy involved a full explanation of the research objectives to each of the users. Nonetheless, a collection emerged, to which Knuth and his team added some classical benchmarks, well-known libraries, and programs of their own group. They then proceeded to statistically analyze the collected 250,000 cards (representing 440 programs) in order to get "a fairly clear conception of how FORTRAN is being used, and what compilers can do about it."

For that, Knuth and his team used a combination of static and dynamic analyses. For instance, they statically calculated how often different types of statements occur and what are the specifics of their usage (how deep loops are, what is the nature of assignments, what is the format of 'if' statements, etc.). The results showed that most of the time, "compilers do surprisingly simple things." Knuth and his team have also profiled running programs by counting how many times statements were executed: thus, they have refined information on statement usage (e.g., that assignments in the replacement style, i.e., $A = B$, occur more often in the initialization sections and not in the loops). Then, for a small subset of 17 random programs, Knuth and his team have closely analyzed the most time-consuming parts of the programs and have manually translated them into machine language using a collection of optimizations—to find that in comparison to the original compiled version, they have gained a four- or five-fold increase in speed.

Altogether, Knuth makes a point that compiler writers should be aware not only of the best and the worst cases, but also of an average case of programs. He argues that complimentary to the common point of view—programmers should be influenced by what their compilers do—there should be an alternative point of view stating that the design of compilers (and therefore of languages that they represent) should be strongly influenced by what programmers do.

Generalized, we find this stance logically and practically appealing: the actual usage of a software language should be taken into account when developing it. Prescriptive approach (how the language should be used—e.g., documentation and tutorials) should go hand in hand with descriptive approach (how the language is actually used—e.g., Knuth's study). Such feedback loop allows language engineers to make informed decisions about future course of the language.

In our work, we follow the same motivation when understanding usage of different software languages. We identify challenges of such research and the ways to overcome them.

### 1.1.2 Research areas

Using Knuth's study as a representative example of the kind of research we report in our thesis, we identify the research areas that our work falls into.

#### Software Language Engineering

The focus of Knuth's study is on a programming language, FORTRAN, and its users (programmers) as well as engineers (compiler writers). In that, the study belongs to the research area of Software Language Engineering, which is concerned with the software languages—artificial languages used in software development. The definition of a software language that we consider in our thesis is intentionally broad: it includes general-purpose programming languages along with domain-specific languages, modeling and metamodeling languages, as well as implicit approaches to language definition such as application programming interfaces (APIs).[1]

[1] Cf. the description of the scope of the International Software Language Engineering conference at `http://planet-sl.org/sleconf/`

Software Language Engineering is considered to be about the systematic design, implementation, deployment, and evolution of software languages [113]—i.e., generally, the prescriptive approach dominates in the research area. Such studies as Knuth's complement the typical approach by providing data about the actual usage of software languages—information essential for insights and reasoning about the current state of affairs.

#### Empirical Research

Empirical research is usually perceived as taking one of the established forms with well-defined protocol of the study and applied techniques: controlled and quasi-experiments, exploratory and confirmatory case studies, survey, ethnography, and action research [167, 170]. In a broader sense, which we consider in our thesis, empirical research includes any research based on collected evidence (quoted from [167], emphasis ours): "Empirical research seeks to explore, describe, predict, and explain natural, social, or cognitive phenomena by using evidence based on observation or experience. It involves obtaining and interpreting evidence by, e.g., experimentation, systematic observation, interviews or surveys, or *by the careful examination of documents or artifacts*."

Knuth's study falls within such broadly understood, de facto empirical research: it uses collected software artifacts to derive and analyze empirical evidence with the purpose of exploring and describing the typical use of the FORTRAN language so that to predict the input for compilers.

## 1.2 Problem Statement

As Knuth's example shows, empirical study of a software language involves two distinct phases: collecting the evidence[2] (punch cards) and analyzing it to answer the questions at hand (how FORTRAN is being used). Below, we discuss each phase, identify its intrinsic challenges, and connect each phase with its research area(s) (see Fig. 1.1).

### 1.2.1 Phase I – Collecting the evidence

Knuth's study has a software language as its object of interest and in order to draw conclusions about the language, Knuth analyzes the language instances—programs—obtained via various ways from the programmers. The obstacles that Knuth had to overcome when collecting the evidence are not exceptional: the more sophisticated and demanding the applied analysis is, the more time the researcher spends preparing the collection of empirical evidence for it. The lack of common

[2] The term 'evidence' can take one of the two meanings: i) in a stricter sense, data derived analytically from the collected artifacts; ii) in a broader sense, it means collected software artifacts, too. Throughout the thesis, we imply the second, broader, definition.

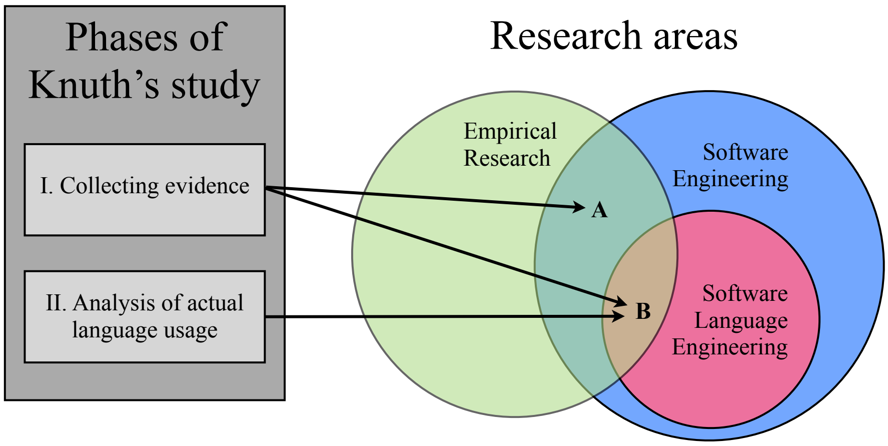


**Figure 1.1.** Phases of the motivating study and their correspondence to research areas

methodology covering this basic step makes researchers to possibly repeat the same actions collecting possibly the same evidence. Based on the level of existing demand, having public shared collections might be an option.

In Knuth's study, the phase of collecting the evidence serves the purpose of analyzing actual usage of a programming language. In other words, the evidence is collected for an empirical task of Software Language Engineering (area B on Fig. 1.1). Nonetheless, collecting the evidence is often required for empirical tasks of Software Engineering as such (area A on Fig. 1.1), especially when those are of practical kind, explicitly using software in their analyses: program comprehension, maintenance, reverse engineering, and re-engineering.

### 1.2.2 Phase II – Analyzing the evidence

We have identified Knuth's study as an empirical task of software language engineering (area B on Fig. 1.1) with the aim to discover the actual usage of the language. Software Language Engineering typically treats languages as software and for that adopts concepts and techniques from Software Engineering. For the actual usage analysis, these might not be fully suitable and sufficient as there are intrinsic characteristics of a language that are different from that of software. Indeed, in his study, Knuth applied Software Engineering techniques such as statical and dynamic analyses to collect the raw data (by parsing, profiling, debugging) but the applied analyses providing the insights were defined by the structure of the language.

### 1.2.3 Research goals of the thesis

Using Knuth's example, we have shown how generalized phases of such study relate to the research areas. Namely, we have identified collecting empirical evidence as a common task for empirical research in both Software Engineering and Software Language Engineering. We have also established that empirical analysis of the actual language usage belongs to Software Language Engineering. Based on the observed challenges in the identified phases, we pose two main research goals of the thesis that can be linked back to the phases of the motivating example of Knuth's study, though in the reversed order:

*1. Develop and apply techniques to empirically analyze actual usage of languages.*

*2. Understand the usage of empirical evidence in Software Engineering research.*

Below we break down the goals into the plan of attack.

- To address the first goal, we perform empirical studies on actual language usage so that to understand the necessities of such research and gain hands-on experience. For that, we analyze actual usage of several software languages. In each case, the research tasks vary depending on the application domain of the language. During these studies, we develop and apply different kinds of analyses that are tailored to the software languages as such.
- To address the second goal, we need to understand how common is the task of collecting empirical evidence in Software Engineering research and what are the characteristics of the collected evidence. For that, we analyze the existing research by performing literature surveys of published papers.
- To bring the two research goals together, based on our experience gained in the empirical studies of actual language usage, we identify obstacles in the process of collecting empirical evidence and ways to overcome them. Namely, we identify requirements and obstacles we have encountered during the empirical studies on API usage analysis[3] and summarize our knowledge in a method of corpus (re-)engineering.

## 1.3 Outline of the Thesis

Figure 1.2 shows the structure of the thesis and connections between its components. The main two parts, Part II and Part III, correspond to the research goals of the thesis. Each part consists of two chapters. Arrows between chapters and parts show flow of knowledge. For instance, the experience gained during studies on actual language usage influenced our stance on corpus engineering. The developed methodology of corpus engineering and its result (a corpus)—in its turn—were used in the advanced study of API usage.

[3] On the definition of APIs as domain-specific languages see Section 2.3.2.

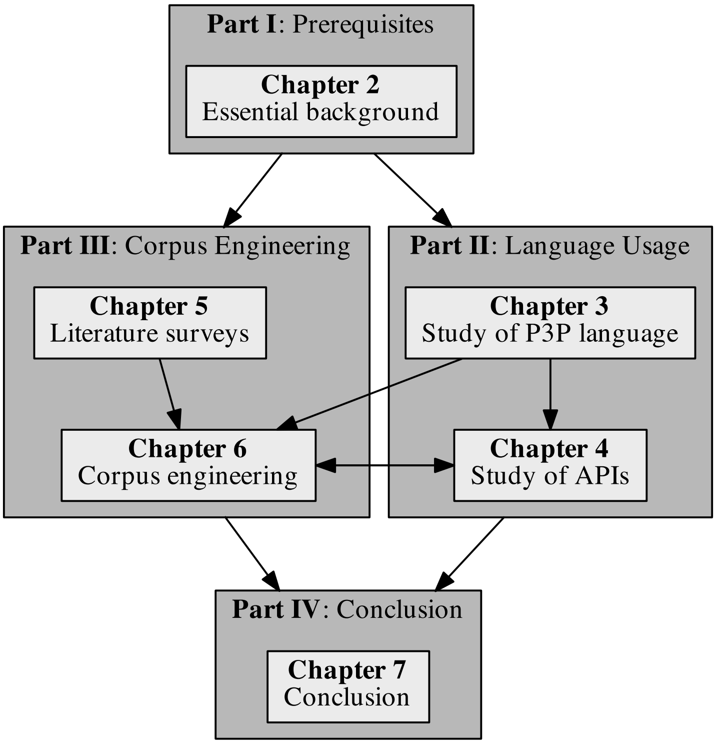


**Figure 1.2.** Connections between the chapters of the thesis

The detailed break-down of the thesis structure is as follows:

*Part I, Prerequisites*

Chapter 2 provides essential background for the rest of the thesis. We discuss Software Linguistics and the actual usage of a language. We provide an overview of the languages studied in the thesis. We introduce terminology for corpus engineering. We identify and compare possible ways of conducting a literature survey.

*Part II, Language Use*

This part presents the results of our research on language use. The studies coming from different application domains and having different motivations demonstrate the same language-centric approach to a problem.

Chapter 3 presents an empirical study of the P3P language, a domain-specific language for privacy policies. We selected this language because there is growing recognition that users of web-based systems want to understand, if not to control, which of their data is stored, by whom, for what purpose, for what duration, and with whom it is shared. We devise and apply methods to study usage profiles, correctness of policies, metrics, cloning, and language extensions.

Chapter 4 presents an empirical study of API usage. We begin with an initial exploration of the topic in the context of open-source Java projects, where we demonstrate examples of large-scale API usage analysis and detect different styles of usage (framework-like vs. library-like). We investigate further framework usage by developing a *framework profile*—a range of reuse-related metrics to measure the as-implemented design and its usage—and applying it to the Microsoft .NET Framework. Finally, based on the developed analyses, we introduce a catalogue of exploration activities to capture API usage accompanied by a tool.

*Part III, Corpus Engineering*

This part describes our study on existing use of software artifacts in empirical Software Engineering and our method for corpus (re-)engineering. The literature surveys assess the existing demand for corpora in contemporary research—which motivates our efforts in providing matching supply in the area of our expertise.

Chapter 5 describes literature surveys that we carry out in order to understand the existing usage of empirical evidence in Software Engineering. We collect and analyze published papers, extracting signs and characteristics of used empirical evidence. We discover that more than 80% of papers in Software Engineering research use software projects as empirical evidence.

Chapter 6 describes our effort on corpus (re-)engineering. The discovered demand for corpora motivates us to provide matching supply. In this chapter, we identify obstacles to corpus adoption based on our own experience (see Chapter 4) and develop a method for comprehending and improving corpus content, producing a complete, automatically buildable corpus, with extended metadata. We apply the method to the Qualitas corpus [175], whose adoption is thereby simplified.

*Part IV, Conclusion*

Chapter 7 concludes the thesis.

## 1.4 Contributions of the Thesis

We list the contributions of the thesis breaking them down into concepts, application, and tangible deliverables.

### 1.4.1 Literature surveys on usage of empirical evidence

*Concepts*: we adopt and adapt Grounded Theory in a literature survey.
*Application*: we perform three literature surveys on usage of empirical evidence.
*Tangible deliverables*: the collection of coded papers.

### 1.4.2 Corpus engineering

*Concepts*: we develop a method for corpus (re-)engineering.
*Application*: we demonstrate our method on a popular existing corpus.
*Tangible deliverables*: an improved version of the corpus.

### 1.4.3 Language studies

*Concepts*: we develop analyses from the perspective of Software Linguistics.
*Application*: we perform three empirical studies applying the developed analyses.

Furthermore, contributions to the application domains of the researched languages:

**P3P privacy policies**

*Application*: we
- analyze validation levels and constraints of the language;
- analyze common, dominating policies;
- develop and apply a range of metrics to capture different aspects of policies.

*Tangible deliverables*: a corpus of P3P policies.

**APIs**

*Application*: we
- develop and apply fact extraction for APIs that operates on resolved types;
- develop and apply a range of metrics to capture reuse characteristics of APIs;
- identify and describe explorational insights for API usage analysis.

*Tangible deliverables*: a tool for API-/project-centric code inspections.

## 1.5 Supporting Publications

The thesis is supported by the following publications, listed in the chronological order:

1 Ralf Lämmel and Ekaterina Pek. Vivisection of a non-executable, domain-specific language - Understanding (the usage of) the P3P language. In *Proceedings of the 18th International Conference on Program Comprehension (ICPC)*, pages 104–113. IEEE Computer Society, 2010.
2 Jean-Marie Favre, Dragan Gašević, Ralf Lämmel, and Ekaterina Pek. Empirical language analysis in software linguistics. In Brian A. Malloy, Steffen Staab, and Mark van den Brand, editors, *Proceedings of the 3rd International Conference on Software Language Engineering (SLE)*, volume 6563 of *Lecture Notes in Computer Science*, pages 316–326. Springer, 2011.
3 Ralf Lämmel, Rufus Linke, Ekaterina Pek, and Andrei Varanovich. A framework profile of .NET. In Martin Pinzger, Denys Poshyvanyk, and Jim Buckley, editors, *Proceedings of the 18th Working Conference on Reverse Engineering (WCRE)*, pages 141–150. IEEE Computer Society, 2011.
4 Ralf Lämmel, Ekaterina Pek, and Jürgen Starek. Large-scale, AST-based API-usage analysis of open-source Java projects. In William C. Chu, W. Eric Wong, Mathew J. Palakal, and Chih-Cheng Hung, editors, *Proceedings of the 26th Symposium on Applied Computing (SAC)*, pages 1317–1324. ACM, 2011.
5 Ralf Lämmel and Ekaterina Pek. Understanding privacy policies - A study in empirical analysis of language usage. *Empirical Software Engineering*, 18(2):310–374, 2013.
6 Coen De Roover, Ralf Lämmel and Ekaterina Pek. Multi-dimensional exploration of API usage. In *Proceedings of the 18th International Conference on Program Comprehension (ICPC)*, 2013. 10 pages. To appear.

The following work being under submission:

7 Ekaterina Pek and Ralf Lämmel. A literature survey on empirical software engineering research. 10 pages.

The relation between publications and chapters is as follows:

- Chapter 3 is supported by publications [1, 5].
- Chapter 4 is supported by publications [3, 4, 6].
- Chapter 5 is supported by publication [2]. Main part of the chapter (the final survey) is the work under submission [7].
- Chapter 6 is supported by publication [6].

# Part I

# Prerequisites

2

# Essential Background

In this chapter, we provide essential background for the rest of the thesis. We discuss Software Linguistics and the actual usage of a language. We provide an overview of the languages studied in the thesis. We introduce terminology for corpus engineering. We identify and compare possible ways of conducting a literature survey.

**Road-map of the chapter**

- Section 2.1 discusses Software Linguistics.
- Section 2.2 provides terminology for actual usage of languages.
- Section 2.3 introduces software languages being analyzed.
- Section 2.4 discusses corpus engineering.
- Section 2.5 gives an overview of survey methodology.

## 2.1 Software Linguistics

In this section, we discuss Software Linguistics, a research stance complementary to the one adopted in the community of Software Language Engineering. We discuss how existing science of Natural Linguistics can be inspirational for Software Linguistics: we consider examples of sub-disciplines not necessarily exercised in this thesis but helpful for understanding the spirit in which it is done.

### 2.1.1 Stances in Software Language Engineering[1]

#### Software Language Engineering, or "Software Languages are Software too"

Software language descriptions and processors are pieces of software. Hence, all kinds of Software Engineering concepts and techniques can be adopted to software languages. Software Language Engineering is about the systematic design, implementation, deployment, and evolution of software languages [113]. Clearly, software language descriptions have particular properties, when compared to other kinds of artifacts in software engineering. Hence, traditional software engineering life-cycles, methods, set of qualities and constraints must be genuinely adapted. If we think about the distinction of software languages vs. natural languages, then Software Language Engineering can be compared to the established field of *Natural Language Engineering* [46, 68].

#### Software Linguistics, or "Software Languages are Languages too"

Software Language Engineering practices should be informed by scientific knowledge. In the case of natural languages, linguistics is the science of languages [45]. Hence, it is worth to see which concepts, research methods, perhaps even techniques or results from the field of linguistics could be adopted to the study of software languages. In this manner, we obtain "Software Linguistics". The term *Software Linguistics* was introduced by Misek-Falkoff in 1982 [140]. This term and the whole concept of adopting linguistics for software (programming) languages has not seen much interest. We note that Software Linguistics should not be confused with Computational Linguistics—the former is "linguistics for software languages"; the latter is (simply speaking) "computing for linguistics" (for natural languages).

Below, we give examples of the broader understanding and definition of Software Linguistics—through references to Natural Linguistics. We have found that the mature, scientific framework provided by linguistics can be reused for software languages—even though many techniques related to natural languages may not be (directly) applicable. In fact, one can systematically mine Software Linguistics from resources such as "The Cambridge encyclopedia of language" [45].

[1] This subsection is based on own publication [2].

### 2.1.2 Examples of Linguistic Sub-disciplines[2]

#### Comparative linguistics

Comparative linguistics studies, compares and classifies languages according to their features using either a quantitative or qualitative approach. It aims at identifying patterns that are recurrent in different languages, but also differences and relationships between languages. Comparative linguistics may also be applied to software languages. For instance, "Programming Linguistics" [50] compares programming languages in terms of commonalities, relationships, and differences while discussing basic matters of syntax, semantics, and styles. "The comparison of programming languages: A linguistic approach" [79] goes deeper into linguistic aspects.

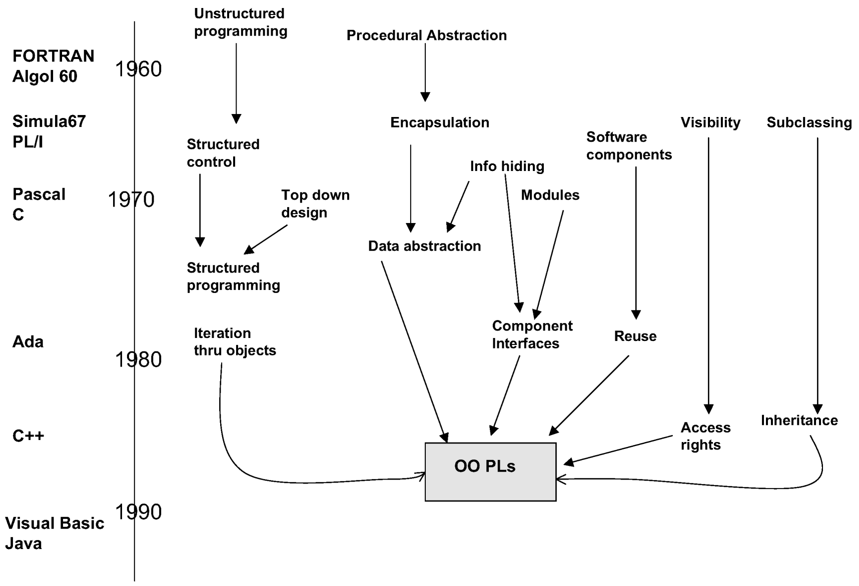


**Figure 2.1.** Historical linguistics: Time line for data abstraction as presented by Ryder et al. in [19]

#### Historical linguistics

Historical linguistics studies the history and evolution of languages—often with the goal of identifying language families, that is, languages that derive from a common ancestor. Part of this research compensates for the lost origin of natural languages.

[2] This subsection is based on own publication [2].

In the case of software languages, history is often well documented. Consider, for example, the History of Programming Languages (HOPL) conference series. HOPL focuses on programming languages rather than software languages in general. Also, HOPL does not stipulate systematic linguistics studies; a typical paper relies on historical accounts by the primary language designers. Some reports, though, provide a large set of qualitative information regarding the evolution of dialects of languages, e.g., see Figure 2.1 showing the time line for data abstraction as presented by Ryder et al. in [19]. The *impact* of the evolution of software languages on software evolution, though real, is not well understood [62], and deserves more research inspired by linguistics.

#### Geo-linguistics

Geo-linguistics studies the intersection of geography and linguistics. Studies can, for example, take into account the distribution of languages or dialects over countries, continents, and regions. In work of Steele and Gabriel [169], for instance, the emergence of dialects of Lisp are considered in terms of geographical zones.

#### Socio-linguistics

Socio-linguistics studies languages as social and societal phenomena. The design of software languages and dialects is often directly linked to such phenomena, too. For instance, in [169] Steele and Gabriel conclude “Overall, the evolution of Lisp has been guided more by institutional rivalry, one-upsmanship, and the glee born of technical cleverness that is characteristic of the hacker culture than by sober assessments of technical requirements”. The field of socio-linguistics for software remains largely unexplored, but see [27] for a related account.

## 2.2 Actual Usage of Language

In this section, we explore the part of Natural Linguistics related to the actual usage of a language, identify what we can borrow for Software Linguistics, and root the identified terms and ideas in the research area of our interest.

### 2.2.1 Prescriptive and Descriptive Linguistics

A part of Natural Linguistics is concerned with developing models of linguistic knowledge; it looks for formalisms that are suited to represent the phenomena of the language. The points of view, though, could differ: represent the phenomena *as they are* or *as they should be*. The latter line of practice, called prescriptive linguistics, aims to establish a standard language, e.g., for education and mass media. The former line of practice, called descriptive linguistics, aims to reveal the actual usage of the language [147].

While prescriptive linguistics follows deductive reasoning, descriptive linguistics follows inductive reasoning. The approaches complement each other, because observing data allows to infer theory (inductive reasoning), but the resulted theory cannot be tested by the same method and it requires deductive reasoning. Deductive reasoning acts in the opposite direction: assuming the theory to be true, it predicts the data to be observed. By comparing predictions with the actual data, one can assess the validity of the theory. Methods used in each reasoning process adhere to the nature of reasoning: inductive methods operate on large amounts of data, they are descriptive and hard to harvest for coherent explanation; deductive methods use logics to draw conclusions from assumed premises [67].

The descriptive approach, which follows inductive reasoning, therefore, is often quantitative. Empirical, data-driven methods allow answering questions *how much* (the degree to which) and *how often* (the frequency with which) data has certain characteristics. The prescriptive approach, which follows deductive reasoning, is often qualitative. Phenomenological, theory-driven methods allow answering questions *how* and *why* data has those characteristics. Usually, qualitative research follows-up quantitative: in order to understand how or why a particular phenomenon occurs, researchers first need to register its existence [88].

### 2.2.2 Measuring Actual Usage

In Natural Linguistics, prescriptive linguistics needs to have data about the language before distilling it to the rules of the language, since they do not exist originally. With artificial languages, such as software languages, prescriptive linguistics comes first, in the form of grammar, syntax, documentation, etc. Understanding actual usage of software languages is therefore important because it provides feedback to the language designers, developers of language-processing software (e.g., of compilers), users of the language. The descriptive approach in Software Linguistics has more weight compared to Natural Linguistics, because evidence provided by it (when substantial enough) can change the standards of the language—which can be enforced in a stricter way than with natural languages.

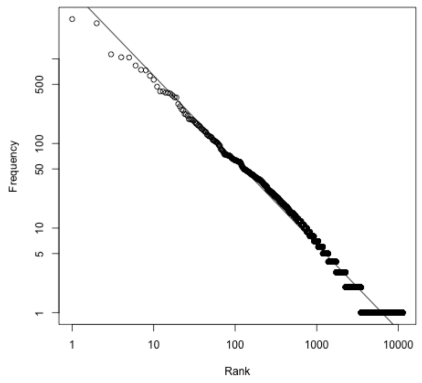
Capturing the actual usage of a language implies working with real-world quantitative data, which can be overwhelming, and—without appropriate methods—it is hard to understand what the data shows.

#### Frequency count

Frequency count is the most basic statistical measure, showing how many times a word occurs in a text. Frequency counts can be used as an absolute number or as percentage. Absolute numbers are often used to describe one dataset on its own: frequency lists allow, for instance, finding the least common terms so that to replace them with synonyms. Also, using absolute numbers, one can observe statistical regularities in the usage of the language. Percentage, on the other hand, allows generalizations and comparisons [84]. Speaking about the data in relative terms facilitates better comprehension. Cf., for instance, “The word ‘of’ occurs in the text 122 times”

| Rank | Frequency | Word |
|---|---|---|
| 1 | 2944 | the |
| 2 | 2632 | of |
| 3 | 1130 | to |
| 4 | 1043 | a |
| 5 | 1036 | in |
| 6 | 833 | and |
| 7 | 741 | for |
| 8 | 731 | is |
| 9 | 631 | that |
| 10 | 566 | are |

**Top 10 most frequent words**



**Zipf's law**

**Figure 2.2.** Frequency of word occurrences in the current thesis

versus “The top 100 of the most common words account for half of this text.” Percentages are also helpful when comparing different datasets or subgroups of the same dataset—of different sizes.

Frequency counts are important for practical reasons. For instance, as it was mentioned, to detect rarely used words for possible replacement. It was also noted that distribution of how frequently words occur follows certain laws, the most famous law being Zipf's law, which states a constant relationship between the rank of a word in the frequency list and the frequency with which it is used in the text.

Fig. 2.2 illustrates the law: it shows top 10 most frequent words from the current thesis along with the plot of frequency against rank on double logarithmic scale. Knowing how frequencies are distributed may help adjusting the way the data is handled: e.g., Breslau et al. [31] developed a simple model where probability of hyperlinks on a webpage follows a Zipf-like distribution; based on that model, they adjusted the web cache replacement algorithm and showed that it performs best on the real trace data. However, establishing if the data follows power-law distribution (which, effectively, Zipf's law is), is a difficult task [36][3] and, therefore, can be error-prone.

## Coverage

The task of descriptive linguistics in the area of software languages differs from that in natural languages. For instance, vocabulary of a software language may be finite

[3] See also a blog post by one of the same authors: `http://vserver1.cscs.lsa.umich.edu/~crshalizi/weblog/491.html`

while that of a natural language is countable. In practice, that means it is possible to calculate the *coverage* of a software language by the given instances of its usage. It may be useful in tasks of re-designing the language: understanding which parts are never or rarely used and planning language adjustments accounting for it. For instance, one of the findings in the empirical study of Java binary code made by Collberg et al. [38]. was that `finally` clause of Java exception handlers is rarely used, while processing of this construct is sophisticated and hinders verification of Java programs.

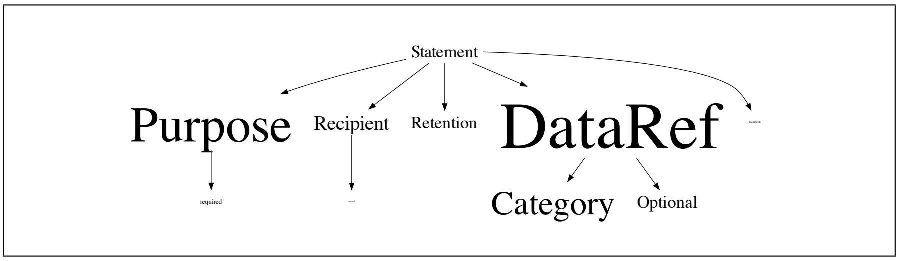


**Figure 2.3.** Frequency of usage for P3P's syntactical domains (early appearance of Fig. 3.11)

Fig. 2.3 shows an example of coverage for the P3P language (see Section 3.4.1 for detailed discussion). Without any prior knowledge about the language, its domain, syntax, and structure, we still can make a few observations based on the obvious fact that elements of the language are used unevenly. First, we see from the presented coverage that the P3P language is mostly used to express knowledge about data, its categories, and some purpose (even without proper understanding what purpose means, we still can see that it is important). Second, we observe that there are few language elements that are rarely used (in fact, even hardly visible on the figure). From that, we can provide the following high-level conclusions: i) any changes in the language related to representation of data, categories, and purposes will have a high impact on the existing P3P adopters; ii) a qualitative study may explain, whether design of the language was over-protective (providing options that are not demanded in reality) or dissemination of the language was not successful (meaning that the users do not understand/know about these options).

## Footprint

In everyday speech or writing, borrowing words or even mixing languages is a common affair. Sometimes, after an extended period of use, borrowed words become officially assimilated into the language and are included into the dictionaries—as happened, for instance, with the words "taco" and "sushi" that originated from Spanish and Japanese languages, but now appear in the English dictionaries such as Merriam-Webster[4]. Sometimes, though, use of another language in the native tongue (or vice

[4] `http://www.merriam-webster.com/`

versa) leads to creation of macaronic languages, where words and syntax of different languages are intertwined. This often happens in the context of immigration: for instance, a flow of immigrants from Post-Soviet countries to Germany in early 1990s led to creation of Deutschrussisch, or Nemrus. Such languages exist only due to their use; they do not have textbooks or grammar rules.

In contrast to natural languages, software languages have clear and strong boundaries; it is not possible to casually blend one software language into another. Still, software language can be embedded—often, the host language is a general-purpose programming language and the embedded language is a domain-specific one. In such cases, a useful measure as *footprint* comes to help: measuring "contamination" of programs written in the host language—with the embedded language.

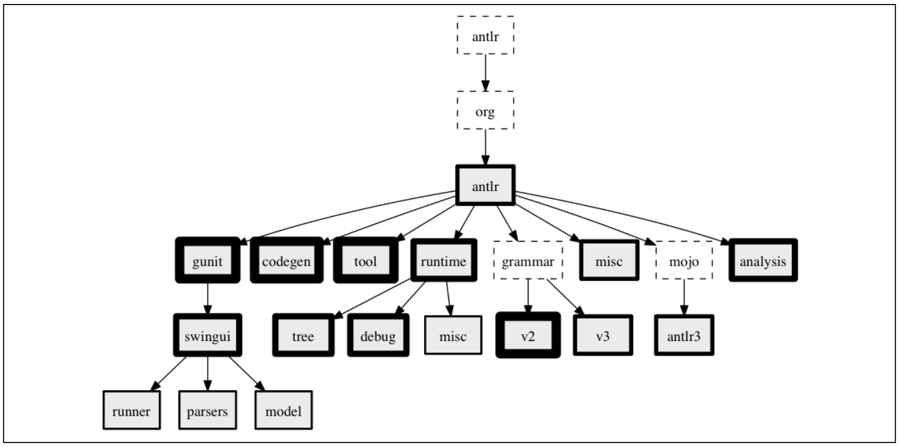


**Figure 2.4.** Footprint of APIs in ANTLR project

For instance, when programming with APIs (application programming interfaces)[5], it is useful to be able to assess API usage within a project. An example of footprint with respect to API usage is presented on Fig. 2.4. The figure represents a project-centric view provided by the tool Exapus (see Section 4.4). The view shows "contamination" of the source code of a project with references to APIs. We see the package tree of the source code of an example Java project, ANTLR[6]. Packages that do not contain any types are marked with the dotted line, otherwise the width of the border is based on the total number of references to various APIs found in the package. Such view allows quick assessment of the status of a project, when updating versions of the used libraries or migrating between APIs. For instance, we observe that there are a few packages with small number of references to APIs. As a more specific and more interesting observation, we notice that subpackages of the same parent package reference APIs differently. Within `runtime` package, subpackage `misc`

[5] On the definition of APIs as domain-specific languages see Section 2.3.2.
[6] `http://www.antlr.org/`

references various APIs less than subpackages `tree` and `debug`: a possible explanation (based only on the names of the packages) is that utility for runtime-related tasks that `misc` subpackage contains is most probably internal, project-related functionality (therefore, no references to APIs). More intriguing is the situation with `grammar` package: its subpackages `v2` and `v3` (apparently, representing versions) vary significantly in API consumption based on references. Without further investigation, it is hard to suggest an explanation for this phenomenon: it might be, for instance, that types in the next version of grammar derive from the previous version, therefore, hiding the API consumption via references.

Such footprint of API usage is the simplest (yet powerful) measure. We can further vary the scope so that to investigate a particular API or even its part. More sophisticated forms of footprint can be done by considering API types implemented or extended by the project.

## 2.3 Languages Under Study

In this section we briefly describe the languages that will be the subject of empirical studies presented in Part II of the thesis (Chapters 3 and 4).

### 2.3.1 Platform for Privacy Preferences[7]

There is growing recognition that users of web-based systems want to understand, if not to control, which of their data is stored, by whom, for what purpose, for what duration, and with whom it is shared. W3C's standardization effort on the Platform for Privacy Preferences (P3P) specifically produced the *P3P language* [189, 42] for privacy policies, which can be viewed as a simple, domain-specific language for privacy concerns of the users.

The P3P effort was started in 1996 and closed in 2006. P3P's impact in the wild has been limited [164, 43]: browser developers were slow to support the negotiation protocol of P3P so that it was even dropped from the standard; lack of industry participation resulted in the discontinuation of the standardization effort for the second, more matured version of P3P; these days, even major websites have invalid or essentially useless policies and some policies are removed.

There may be several reasons for such limited impact [194, 136]. One may speculate about lack of stable knowledge of the web-privacy domain or lack of incentive for the industry to support web privacy. (See [164, 43] for the discussion of such causes.) We adopt a language engineer's point of view to better inform the discussion of limited impact of P3P and future language-based developments in the web-privacy domain. For that, we perform a vivisection of P3P.

[7] This subsection is based on own publication [5].

### P3P samples

Let us quickly introduce P3P by means of samples. The following two samples, except for minor variations, are very common in practice.

```
<POLICY>
  <STATEMENT>
    <NON-IDENTIFIABLE/>
  </STATEMENT>
</POLICY>
```

**Figure 2.5.** P3P sample 'full privacy'

The policy in Fig. 2.5 applies to a website with 'full privacy'. This policy, without any data references and with the special NON-IDENTIFIABLE element, signifies that the underlying web-based system does not collect any data.[8]

Usually a P3P policy consists of any number of *statements*—each of which signifies *data references* for the collected data, *purposes* of collecting data, *recipients* that receive the data, a *retention* level, which defines how long data is stored, and a few other directives.

The policy in Fig. 2.6 applies to a website which performs 'logging' but collects no other data. More in detail, the specific data reference #dynamic.clickstream signifies that the website collects and stores information for Web-server access logs, and #dynamic.http is about information carried by the HTTP protocol which is not covered by #dynamic.clickstream. User data, in the more narrow sense, is not collected and stored. The purposes admin and develop mean that (weblog data) is collected for the website's benefit—the user will not be contacted or affected in any way. The purpose current is whatever user and system consider the more specific purpose of the system—be it filling in and sending out birthday cards. The only recipient of the data is 'ours', which essentially corresponds to the website. The data may be stored indefinitely.

## 2.3.2 Application Programming Interfaces

The term API stands for "application programming interface" and in the broadest sense means bundled code intended for reuse. Code reuse is the way of taking benefit of existing solutions: in its primitive form, it manifests itself as copy-and-paste; in its thought-through form, it exists as APIs. APIs are designed to be conveniently reusable, for that code is grouped into methods, classes, packages —i.e., given a structure that encompasses functionality, part of which is exposed for external consuming.

[8] Please note that the NON-IDENTIFIABLE element can also be used in a statement in combination with data references. In such case, the element signifies that "all of the data referenced by that STATEMENT will be anonymized upon collection" [189].

```
<POLICY>
  <STATEMENT>
    <PURPOSE><admin/><current/><develop/></PURPOSE>
    <RECIPIENT><ours/></RECIPIENT>
    <RETENTION><indefinitely/></RETENTION>
    <DATA-GROUP>
      <DATA ref="#dynamic.clickstream"/>
      <DATA ref="#dynamic.http"/>
    </DATA-GROUP>
  </STATEMENT>
</POLICY>
```

**Figure 2.6.** P3P sample 'logging only'

What we collectively call APIs in this thesis, differ in their style and purpose of bundling the code for reuse (examples are taken from [171]): they might be libraries (as "standard" library in C), frameworks (as .NET), software development kits (as Android SDK), APIs (as Google Map API), to name a few. Libraries can be seen as the simplest style of reusable code consumption: plain instantiation of appropriate objects and invocation of the methods. Frameworks leverage inheritance as means of control and achievement of user's goals: implementing an interface or extending a class gives the user opportunity to build-in into the existing frame of functionality. Software development kits provide more than just reuse of functionality, they provide a set of development tools, a platform for creation of the software. APIs in the strictest sense are the public interfaces of existing software, which allow using the software programmatically.

Typically, the user of an API sees and interacts only with the public part of an API, not knowing (or caring) what happens in the hidden implementation or how it is arranged. From this point of view, APIs can be seen as embedded (or internal—according to Fowler's classification[9]) domain-specific languages: there is a general purpose programming language (a "host" language), like Java or C#, which provides generic language constructs (iteration, conditionals, functions, etc.), and there is an API, which provides domain-specific primitives. This combination allows software engineers to write code in the familiar context, with the language they know, using API primitives to work with the domain that the API represents on a higher level of abstraction.

There are two complementary points of view on APIs. In the first place, an API can be seen as a finite set of public types and public methods exposed for the benefit of the API user. This basic point of view gives an idea of the API structure, its size, and available functionality; it allows understanding *what* is there to use in the API—i.e., it represents a *grammar* of the API as a language. Secondly, an API can be seen continuously, as a sequence of actions: implement or extend an interface or a type, instantiate an object, call a method. This point of view facilitates correct consumption

[9] http://martinfowler.com/articles/languageWorkbench.html#InternalDsl

of the API; it allows understanding *how* to use the API—i.e., it represents a *protocol* of the API as a process.

The first, grammar-like, view on APIs is usually facilitated by public documentation, as, for instance, Javadoc generates, where all the types and methods, return types and parameters are listed and explained. The second, protocol-like, view on APIs is usually facilitated by examples, tutorials, or demos provided by the API developers. Research-wise, the first view gives rise to studies on how to reflect the information about the usage in the grammar-like representation of APIs (e.g., apply font scaling in the documentation with respect to popularity of types and methods). The second view gives rise to research that seeks to mine the information about protocol aspect of APIs from their usage (e.g., detect usage patterns).

In our thesis, we are taking the first point of view: we treat APIs as domain-specific languages and we are interested in their usage with respect to their grammar.

## 2.4 Corpus Engineering

In this section, we introduce terminology necessary to discuss the phase of collecting empirical evidence—once again, referring and borrowing from Natural Linguistics as we see fit and beneficial for Software Linguistics.

### 2.4.1 Corpora in Natural Linguistics

In Natural Linguistics, samples of the language under the study are often organized into collections that are called *corpora*. A branch of Natural Linguistics, corpus linguistics, deals with all aspects of designing, producing, annotating, analyzing, and sharing corpora. Producing a useful, natural linguistics corpus could be an effort that goes far beyond what individual researchers or teams can do. There are international associations who support sharing, e.g., the European Language Resources Association (ELRA)[10]. Some of these organizations charge a sum of money for providing their corpora.

Using corpora is beneficial for different reasons. From the technical point of view, it saves time and effort for individual researchers, it increases the quality of research, it makes replication of studies easier. From the research point of view, the contents of corpora provide evidence that has not been biased by the researcher. Corpora may also make possible discovery of phenomena that otherwise are not observable [94].

Corpora are usually collected in a systematic way, following certain extra-linguistic principles (certain time span, certain type of texts, etc.). Corpora are also often contain metadata that makes it easier to work with the texts: annotations, markup, tagging parts of the speech, statistical information.

[10] `http://www.elra.info/`

### Types of corpora

Corpora in Natural Linguistics can contain texts written in one or many languages, i.e., be *mono-* or *multilingual*. The basic type of a corpus is a *sample corpus* (other types can be seen as the sample type with additional features). A sample corpus is supposed to show the general usage of a language: represent the usage of language features and elements proportionally to what can actually be found in reality. A *reference corpus* is supposed to reliably represent all the features of a language (i.e., reference corpora tend to be quite large). Usually, corpora do not include time dimension into their design explicitly, they are 'snapshots' even if they cover a substantial period. There are two types of corpora that are designed with time dimension taken into account: *diachronic* and *monitor* corpora. The former contain several snapshots at different intervals of time (covering altogether a considerably long period); the latter are continuous and growing with the time. There are also *special* corpora that are not intended to represent the general usage of a language, rather to facilitate some knowledge acquisition about a particular phenomenon. Often, corpora are intended to capture the normal usage of the language, meaning that only accomplished native speakers are to be taken into account when gathering the data, in order to represent a standard of the language. Such corpora are called *normative*. There are further classifications of corpora in Natural Linguistics (see [30]), but we omit them here as they are outside of the context of our study.

Since it is hard to operate on large collections of plain text, in order to facilitate research corpora are marked-up, annotated, or tagged, with additional metadata, as, for instance, part of the speech—verb, noun, etc.—which are hard to detect automatically. While having metadata leads to machine-readable corpora and extensible research, queries based on annotations can reveal only the facts that are capturable by the metadata that was identified earlier.

## 2.4.2 Corpora in Software Linguistics

Software Language Engineering is not the only area of Software Engineering that makes use of collections of software artifacts. Often, practical studies concerned with such topics as program comprehension, maintenance, reverse engineering and re-engineering use collections of software artifacts. Figure 2.7 presents a schematic depiction of such research.

The process of collecting software artifacts, though frequent in research, does not have an explicit methodology. The steps involved are nonetheless easy to distinguish. The process starts with identifying a set of software artifacts to be collected and pre-processed; the obtained corpus is then subject to fact extraction; facts are usually extracted once or on a regular basis and stored in a relational database; the extracted facts are then analyzed and the results are interpreted. The specifics of research questions influence each step of the process—choice of the type and instances of software artifacts, extracted data, applied analyses—and it is probably impractical to generalize the whole process. However, the phase of collection and pre-processing artifacts should be possible to standardize either methodologically or by using established collections for fact extraction.

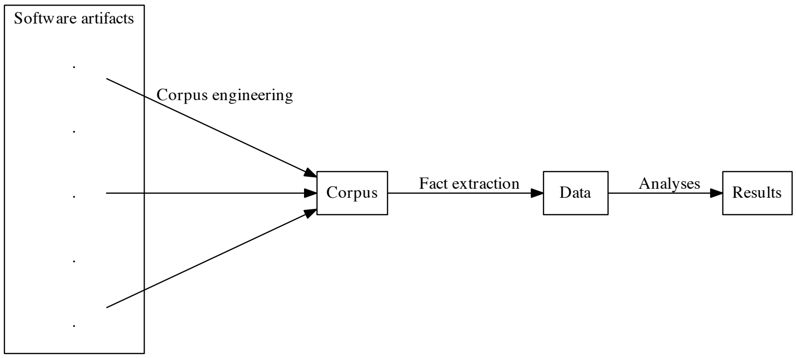


**Figure 2.7.** Simplified depiction of corpus-based Software Engineering research

## Terminology

Let us introduce terminology that we will use further throughout the thesis. A *corpus* is a collection of *software artifacts* that, generally speaking, can be of any nature. We name *corpus engineering* the tasks related to the phase of collecting and preprocessing the artifacts. *Unit* of the corpus captures the term in which contents of the corpus are described: images, XML files, software projects. Our assumption (that we confirm by the literature surveys) is that Software Engineering research often makes use of corpora that consist of software projects (programs, systems). We refer to such collections as *project-based corpora* and will focus on them particularly in this thesis. Project-based corpora can be *mono-* or *multilingual*, when the units (i.e., projects) written accordingly in one or more languages. These corpora contain projects in one of the *code forms*—source code, binary code, or both. In project-based corpora, projects can be the high-level units, in the context of which research is concerned with some additional units (terms): tests, bugs, defects, features, etc.

Following the existing classification of corpora in Natural Linguistics, let us call the project-based corpora that were created to be a diverse representation of projects written in a specific language—*sample* corpora. Often, though, the corpora users are interested in only well-established, real-world projects—we will call such corpora *normative*. Those corpora that were created with an additional requirement (e.g., to cover a specific application domain), we will call *special* corpora. Corpora that contain several versions of the same project(s) we will call *evolutional* (not distinguishing between diachronic and monitor corpora for now, as the state of corpus engineering in Software Engineering is not sufficiently developed for finer terminology).

Historically, corpus engineering is an expensive, time-consuming effort, neither appreciated nor rewarded by the community. The existing established corpora are typically a side-effect of research where a corpus was needed to solve the task at hand; afterwards the combination of altruism and pragmatism motivated additional effort for sharing.

We believe that recently there appeared a growing need for sample normative corpora, when researchers became interested in large-scale corpus-based or corpus-driven studies. For instance, Baxter et al. analyzed statistically structure of 56 Java projects [24] in the corpus-driven style. That study gave rise to the effort of Qualitas Corpus [175][11], a curated collection of Java code—a sample normative corpus, the first of its kind in our opinion (see other corpus engineering efforts discussed in Section 6.5).

## Corpus engineering tasks

There are several corpus engineering tasks that general in their nature when a project-based corpus is concerned, regardless of what other characteristics it should have. We outline these tasks and some guidelines how to address them below.

**Collecting the data.** The following are examples of requirements to be considered before identifying appropriate projects:

- What language projects should be written in?
- Which code form is needed: source code or binary code?
- Should the corpus be diverse along the following characteristics or specific: size, application domain, maturity, etc.
- Are there any additional requirements to the corpus: e.g., should projects be buildable, runnable, have unit tests, etc.
- Is evolution aspect needed?
  - if yes: What kind of evolution measure is needed (releases, versions, commits, etc.)?
  - if no: Which snapshot of projects to consider (latest version, specific date, etc.)?

**Providing additional contents.** Physically obtaining the needed data can be a multi-step process. In the most simple case, projects in their source or binary code form are downloaded from their repository or homepage. Such distributions may be incomplete with respect to used libraries, identifying and acquiring which is a separate process. Initially identified requirements to the contents of the corpus may include broader context than only code form: e.g., a corpus might also include data about bugs, defects, commits, emails, and other artifacts of the project's ecosystem. Obtaining such contents is a separate process that includes identifying sources for the additional contents and ways to collect them (e.g., crawling online sources).

**Sanitizing the data.** While in Natural Linguistics, "actual usage of language" means "as is" (including possible errors and deviations), in Software Linguistics, the definition of a software language is an operational one (involving compilers, interpreters,

[11] `http://qualitascorpus.com/`

and other language-processing software), and except for certain instances of research, studies mostly are based on the correct usage of the language. Therefore, either the corpus should be filtered to contain only valid samples or there should be means of distinguishing different levels of validity.

It is also important for the corpus to be aware of the contents of the projects. For instance, source code form may additionally contain files of non-programmatic nature: documentation, images, and so on. It may contain files directly used in the project but non-executable or written in other languages: configurations, XML schemas, etc. Both source and binary code forms usually depend on libraries, i.e., binary files containing reusable code exercised in the project, which maybe included into the project's distribution or were added on the previous step of corpus engineering. Finally, source code itself may be heterogeneous and contain auxiliary code, for testing, demo, or other purposes, as well as partial clones of different size from other projects (copy-and-paste approach in functionality reuse).

**Providing metadata.** Having metadata in a corpus facilitates fact extraction. Likewise in Natural Linguistics, using corpus metadata in research has benefit (easier processing, delegation of quality, comparability) and drawbacks (restriction of the possible research questions, lack of details)—i.e., corpus-based versus corpus-driven approach.

One example of metadata useful for Software Linguistics can be marking up parts of the projects during the sanitizing phase. Tagging parts of the projects may include any custom needs: identifying and labeling API usage, design-specific parts of the system, etc. A corpus can also provide statistical information: values of different metrics for the code (size, complexity, etc.) Metadata also can be fine-grained (specified on the level of individual types or methods) or aggregated (on the level of packages, directories).

### 2.4.3 Corpora versus Software Repositories

One might argue that there is no need to introduce the new terminology as collections of software artifacts already exist in different forms, one of which—frequently used in Software Engineering research—is software repositories.

The main difference that we see between such collections of software artifacts and corpora is their target group, the stakeholders involved. While the software repositories are tailored towards the developers, the corpora are designed for the researchers—both may contain the same projects but serve completely different goals (developing the code vs. investigating the code) which influences greatly the way of representing and maintaining the respective collections. If to use an analogy from the field of Natural Linguistics, libraries represent vast collections of books written in natural languages—yet they are not corpora, only the possible raw material for them. The same is true for software repositories: the contain software artifacts for the primary use, while corpora contain them for the secondary, meta-use.

In Software Engineering research, software repositories are often the main source for producing throw-away corpora that are used only for the needs of the current publication in the pipeline. This is the tendency that we believe is possible and important to eliminate for the sake of quality and reproducibility of reasearch (if not for the mere comfort of the researchers).

## 2.5 Literature Surveys

As discussed in Introduction, Section 1.2.3, we aim to assess the usage of empirical evidence in Software Engineering by discovering the signs of it in the existing research—namely, in the published papers, via a literature survey. In this section, we describe the possible methods to conduct a literature survey and comment on their applicability to our research. In the end, we outline the resulting method that we use in the thesis.

### 2.5.1 Systematic Literature Reviews

It is often said that science is “standing on the shoulders of giants,” meaning that one should be aware of the previous work and use it when possible for further expansion of knowledge. Practical areas of science, such as Software Engineering, have specific challenges related to this desired continuous flow of research. In order to be able to assess and integrate results obtained by other researchers, one needs to be able to identify research context, applied methods, used empirical evidence, and so on. This necessity is addressed by the evidence-based paradigm, which provides guidelines for systematic collection and analysis of available empirical data w.r.t. some specific phenomenon. Such approach allows getting a broader and deeper knowledge about the phenomenon in question than an individual study. The main method of evidence-based practice is a systematic literature review. It is a literature survey that allows gathering, extracting, and integrating results existing in the research area in focus.

The evidence-based paradigm originated in medicine, where studies are often done in a rigorous statistical manner (experiments, quasi-experiments, etc.). The approach was adopted for Software Engineering by Kitchenham et al. in 2004 [112, 109].

Systematic literature reviews usually have a specific research question in focus that allows to distinguish relevant studies. The search strategy of a systematic review usually consists of i) identifying one or more digital libraries along with an appropriate set of keywords/keyphrases; ii) identifying conferences and/or journals and a time span for the published papers; iii) combination of both. The search strategy allows collecting studies—as many as possible—so that to further assess them w.r.t. previously developed inclusion and exclusion criteria. The papers identified as relevant are then subject to data extraction: collecting the required information to be synthesized at the final stage. Each step of the review is previously documented in a review protocol: the research question(s), the search strategy, the inclusion and exclusion criteria, data extraction strategy, and so on. The review is to follow the guidelines of the protocol at all times—such strictness is to reduce the possibility of researcher bias.

#### Applicability

The major reason why systematic literature review is not a suitable choice for our survey is its rigorousness. The goal of our survey is to discover the existing phenomenon of which we yet do not know much. Therefore, it is hard if not impossible to prepare beforehand an appropriate data extraction strategy that would cover all existing characteristics of empirical evidence used in Software Engineering.

Nonetheless, we borrow from the method of systematic review the most important idea of systematically documenting the made decisions for better reproducibility/assessment of the quality of the survey: e.g., used search strategy, applied methods, etc.

### 2.5.2 Content and Meta-analysis

We also consider two research methods commonly used in social and health sciences, namely, *content analysis* [117] and *meta-analysis* [75]. Content analysis is a qualitative research method commonly used for systematically studying large amounts of communication content such as news, articles, books, videos, or blogs. The key characteristic is that the analyzed content is *categorized* by researchers. Systematic literature surveys [41] often leverage content analysis.

Meta-analysis is another research method used for a systematic analysis of research literature. Unlike content analysis, meta-analysis is a quantitative method, and it is focused on research studies that have highly related research hypotheses. The main goal of a meta-analysis is to aggregate and statistically analyze findings of several research studies. Meta-analysis uses stricter inclusion criteria than content analysis: measured outcomes, and sufficient data to calculate effect sizes. These specifics of meta-analysis challenge its application to empirical software engineering [110].

#### Applicability

The subject of our survey is not exclusively of qualitative or quantitative nature: we both need to be able to detect the categories, in which to speak about used empirical evidence, and to be able to extract their related structural and countable characteristics.

From content and meta-analysis, we borrow the essential idea of *coding* the papers w.r.t. a *coding schema* and performing basic statistical measurements on discovered quantities.

### 2.5.3 Grounded Theory

Grounded theory was developed by Glaser and Strauss in their study of death in the context of patient caring [73]. Their methodology [74], nowadays often used in social sciences, was primarily tailored towards qualitative studies working on the large amounts of data collected from various sources: reports, formal and informal interviews, observations, etc. The data is reviewed and (extensively) coded by the researchers; codes are grouped into concepts; concepts are united within categories; the latter are the basis for a theory—a reverse-engineered hypothesis. This approach is bottom-up, as opposed to a typically used top-down scientific perspective that starts with a hypothesis to be checked. Later, Glaser and Strauss split in their understanding to what extent the coding should be open (on the differences between Glaserian and Straussarian grounded theory see [12]). Following his understanding of openness, Glaser also suggested that grounded theory can be used in quantitative studies [72].

#### Applicability

Grounded theory is more suited to our needs as in our research task we do not have exact definitions for the phenomenon that we intend to capture. Grounded theory may be seen as inexact method, allowing too much freedom, but the general mechanism in its basis—bottom-up direction of the processing of the data that facilitates the emergence of theory—essentially is the same process that leads to an insight and formulation of a research hypothesis "behind the stages" of research.

We use the core idea of grounded theory: we let the coding schema to emerge from the data. We use our pre-existing knowledge about the domain to form an idea of the coding scheme for the survey. During the coding phase, we accumulate new knowledge and allow it to influence the development of the schema.

### 2.5.4 Survey Process[12]

We describe the main steps in our survey process integrating all borrowings from discussed methodologies (the literature survey itself is reported in Section 5.3). We also discuss deviations of the pilot studies that we performed when looking for the right methodology (the pilot studies are reported in Section 5.2). There are two main reasons to the adjustment followed after the pilot studies. First, we have realized that empirical evidence in the form of projects, programs, software is not specific to the area of Software Language Engineering research; that Software Engineering as such often makes use of it. Second, we have come to the conclusion that the most complete scheme that we are able to devise can come only after having considered all the papers; neither theoretical considerations nor expertise can help covering all possible characteristics—only the data itself can.

**1. Research questions**. Initially, our research goal was to characterize research in Software Language Engineering with an objective of streamlining future research by identifying common characteristics. After the pilot studies we refined our goal to the following: discover and understand the use of empirical evidence in Software Engineering with the same objective. More specifically, we seek to understand how often it is used and what are the common characteristics. (The refined research questions are posed in Section 5.3.1)

[12] This subsection is based on own publication [2].

**2. Search strategy**. Initially, we used a selection of papers that were gathered via bibliographical references, starting with a seed—a set of hand-picked papers that we have identified as “appropriate.” For the second pilot study, we decided to collect papers from the proceedings of certain conferences. We identified the list of the conferences from our experience, including those conferences whose topic implies (to our experience) use of empirical evidence. We also decided to take into account only the most recent research, because it became apparent that empirical research became stronger in the community over the years. For the final survey, we adjusted the list to contain only conferences with proceedings of comparable sizes.

**3. Inclusion criteria**. In both pilot studies, we used inclusion criteria. We started with two requirements: i) the paper discusses usage of a software language, and ii) the paper reports empirical results on such usage based on a corpus. After being exposed to the papers and after distilling own experience, we decided to remove the first requirement, taking into account only the fact of some empirical research (understood broadly) reported in the papers. We did so, because we observed that many papers in Software Engineering research in general make use of empirical evidence, such as projects, programs, etc. After the second pilot study, it became apparent that almost all of the papers make use of empirical evidence. This observation led to the decision to restrain from using any inclusion criteria, taking all the collected papers into account.

**4. Data evaluation**. Initially, we were interested in characterizing not only the used empirical evidence, but also the research—focusing only on Software Language Engineering. After we made our scope broader, including Software Engineering in general, it became an overwhelming task, to characterize the research as such, therefore, we have restricted our interest only to the characteristics of the used empirical evidence. In the pilot studies, we have followed a pre-defined coding schema that came from our experience.

**5. Data analysis**. We use basic statistical methods to represent the results: frequency counts, percentages, percentiles. The frequencies describe the level of presence of certain types of empirical evidence and their characteristics. The quantitative description of the current situation in Software Engineering w.r.t. use of empirical evidence helps to discover common ground. Such analysis can also lead to new insights and allows posing new research questions for possible follow-up quantitative as well as qualitative research.

**6. Dissemination**. The approach to the literature survey, its topic, the developed scheme, and the results—we see our effort as the first of its kind. We strongly believe in the bottom-up approach of discovering the methodology for use of empirical evidence in Software Engineering.

# Part II

# Language Usage

# 3 A Study of P3P Language

In this chapter, we present an empirical study of the P3P language, a domain-specific language for privacy policies. We selected this language because there is growing recognition that users of web-based systems want to understand, if not to control, which of their data is stored, by whom, for what purpose, for what duration, and with whom it is shared. We devise and apply methods to study usage profiles, correctness of policies, metrics, cloning, and language extensions.

## Road-map of the chapter

- Section 3.1 briefly introduces P3P language and motivates the study.
- Section 3.2 describes the methodology: research questions, corpus, analyses.
- Section 3.3 presents the essential language definition of P3P.
- Section 3.4 prepares, executes, and interprets analyses on the corpus.
- Section 3.5 discusses threats to validity for this empirical study.
- Section 3.6 discusses related work.
- Section 3.7 concludes the chapter.

## Reproducibility

The reported study relies on a P3P corpus obtained from the Internet. The corpus and various artifacts related to the performed analyses are available online on a supplementary website for reproducibility's sake and for the benefit of anyone who needs an organized corpus of P3P policies[1].

## Related publications

Research presented in this chapter underwent the peer-reviewing procedure and was published in the proceedings of International Conference on Program Comprehension in 2010 [1] and subsequently in the special issue of Empirical Software Engineering journal in 2013 [5].

[1] `http://softlang.uni-koblenz.de/p3p`

## 3.1 Introduction

(A brief introduction to the language is provided in Part I, Prerequisites, Section 2.3.1.)

Let us clarify here our general background as well as the reasons for interest in P3P and language usage thereof. Such background provides context for the research questions to follow.

That is, initially, we were interested in privacy awareness of web-based systems or, in fact, enforcement of privacy policies for such systems [105, 17, 13, 86, 142, 161, 64, 104, 119, 129, 155]. It quickly became clear that P3P is the most advanced language for privacy policies—most advanced in terms of standardization and adoption. Hence, we started to study P3P in a conservative manner—by consulting the specification, additional documentation, and available examples. We also quickly learned about empirical studies of P3P [57, 153, 44, 154], which turned out to be focused on language adoption, and, to a lesser degree, syntactical validity of policies as well as compliance of policies with legislation or human-readable policy descriptions.

Not even the combination of all the aforementioned resources helped us to understand P3P sufficiently from the perspectives of language design as well as language and system engineering; neither could we sufficiently grasp the domain of web privacy in this manner. As a result, we embarked on a different kind of empirical study based on the following research questions: *What part of the vocabulary is used? Is the language correctly used? What is a significant policy? What are common policies? What language extensions circulate?*

## 3.2 Methodology of the Study

The empirical study of P3P language usage is centered around a number of research questions that we describe first. The study relies on a corpus of P3P policies in the wild; the corpus' compilation is described next. The study leverages a number of analyses to gather data towards answering the research questions; these analyses are briefly introduced last. (§3.4 covers all analyses in detail.)

### 3.2.1 Research Questions

The following questions were designed to help understand P3P from the perspectives of language design as well as language engineering; better understanding of the web-privacy domain was an objective, too.

#### What part of the vocabulary is used?

P3P provides a rich vocabulary for the privacy domain. Basic understanding of the domain may benefit from insight into the coverage of the vocabulary in the wild.

Uncovered vocabulary may be superfluous or misunderstood. Frequently used vocabulary suggests a common profile of P3P usage. This research question directly relates to an open challenge with privacy policies according to [164] (see “Recommendations for the Future”; “Keep it simple”).

**Is the language correctly used?**

Like for most other software languages, correct use of the P3P language is not sufficiently constrained by a context-free grammar or an XML schema. There are additional constraints that can be checked for policies. To give a simple example, ‘no retention’ (say, no storage) arguably cannot be combined with ‘public’ as recipient. It is common that the definition of a software language (specifically, when it is a programming language) also provides a type system or a static semantics, possibly extended by rules of pragmatics to describe constraints for correct use. In the case of P3P, there is no comprehensive specification of such constraints available, but some proposals can be found in different sources. The question arises whether policy authors make correct use of the language in terms of obeying these additional constraints.

**What is a significant policy?**

We assume that a correct and precise policy gives insight into the system (the website) to which it is attached. In some limited sense, the policy may be regarded as model of the system. Here, we adopt a reverse engineer’s point of view. For instance, a policy may provide some degree of insight into the data model of the system, the internal or external components or services making up the system, and, of course, the system’s ability to provide its services in the view of specific data that is optionally or mandatorily provided by the website’s user and possibly stored by the system. Accordingly, the question arises what significant policies look like in the wild—significant in terms of serving as an interesting or complex model in the aforementioned sense. Clearly, such complexity may be diametral to privacy, but we are not interested here in the policies promising the most privacy, rather, we are interested in the policies promising the most insight into the attached systems. This question guides us specifically in studying software metrics [63] for P3P.

**What are common policies?**

There are indicators that some policies may be common. In particular, policy authoring tools provide templates that may be used, perhaps even “as is”, except for resolving entity/identity information. These templates are meant to cover simple privacy scenarios for websites; think of the introductory P3P examples. Also, the online discoverability of policies implies that policy authors may easily reuse existing policies, if they see fit. Further, the P3P syntax is so simple, when compared to programming language standards, that one may expect that policy authors naturally end up with equivalent or similar policies. This question guides us specifically to perform clone detection [116] for P3P.

**What language extensions circulate?**

A remarkable feature of the P3P design is that it anticipates the need of extending the P3P language and supports language extension with a designated language mechanism. In different terms, P3P is hence prepared for ‘growing a language’ [100]. The principal feasibility of such an extension mechanism follows from the simplicity of P3P with its emphasis on a vocabulary as opposed to the rich language constructs of programming languages. By studying the usage of said mechanism we gain insight into language extensions that are proposed in the wild, thereby suggesting potential improvements for future policy languages.

## 3.2.2 Corpus under Study

We carried out our empirical language analysis of P3P on a corpus that was obtained from the Internet. In the following, we identify the source of the corpus, we describe the process of corpus retrieval and list size measures for the corpus and subordinated artifacts. Please note that we made the corpus readily available online in an open-source project.[2]

**Open Directory Project**

We used the Open Directory Project (ODP) as the source for finding websites with policies. ODP is a global, openly available, community-moderated, actively maintained directory of companies, consortia, and other entities with a web presence that may be reasonably expected to potentially declare a privacy policy.[3] For convenience’s sake, Appendix A.1.1 provides additional information about ODP’s characteristics. We downloaded the ODP as of 7 August 2010. To the best of our knowledge, ODP is the most comprehensive website catalog of its kind, and hence, it suggests itself as an obvious source for an empirical study like ours. We refer to the threats to validity discussion in §3.5 for reflections on the choice of the ODP source.

We extracted all website URLs from (the downloaded) ODP files. Some URLs occurred more than once. Most URLs used distinct domains; this can be viewed as a simple indicator of diversity. The corresponding numbers are listed in Table 3.1.

| | |
|---|---|
| # Website URLs | 4,009,337 |
| # Distinct website URLs | 3,860,709 |
| # Distinct domains | 2,957,657 |

**Table 3.1.** Results of URL scraping

[2] `http://slps.svn.sourceforge.net/viewvc/slps/topics/privacy/p3p/`

[3] ODP website: `http://www.dmoz.org/`

### Locating and downloading policies

We followed all URLs from ODP to locate policies on the corresponding websites and to download the located policies. These efforts were performed over Aug–Sep 2010 (categories *Regional* and *World*), and Dec 2009–Jan 2010 (all other categories except 'Kids and Teens', which is not included in our corpus due to a lately discovered omission).

The mechanism of finding out whether a site uses P3P and locating the actual policy commences as follows. The site is checked whether a *policy reference file* exists. If so, the policy reference file is processed, and all referenced policies are fetched. In fact, the reference file may use URIs to refer to policies (say, policy files), or policies may be embedded into the reference file. All those policies were retrieved by us.[4] The corresponding numbers are summarized in Table 3.2. Again, numbers of reference files and policies are annotated with the number of distinct domains; this can be viewed as a simple indicator of diversity.

| | **# Files** | **# Distinct domains** |
|---|---|---|
| Located policy reference files | 50,776 | 43,195 |
| Referenced policy files | 13,536 | 8,736 |
| Downloaded policies | 8,768 | 7,746 |

**Table 3.2.** Results of policy location and downloading

We note that reference files may directly contain policies; in these cases, we extracted the policies. One can observe that only few entries of ODP have an associated policy reference file: 3,860,709 website URLs vs. 50,776 policy reference files. Numbers are further decreasing as follows. We could not retrieve policy files for each reference file due to unresolvable references, i.e., failing downloads. Also, there is aliasing involved, i.e., several policy reference files point to the same policy.

### Schema-based validation

The extracted policies are still subject to validation. Each policy file is supposed to contain one or more policies. However, not all policy files contain XML. That is, files may happen to be empty or contain non-XML text. Further, not all policy files with XML content are well-formed XML, and not all well-formed XML content from policy files are valid with regard to P3P's XML schema. Counts of files on varying levels of validity are shown in Table 3.3. Again, numbers of files are annotated

[4] According to W3C's P3P specification [189], the policy reference file may be located by either of the following options. The file may be located in a predefined "well-known" location, which is essentially [WebSiteURL]`/w3c/p3p.xml`. Also, the website may contain the HTML/XHTML tag `<link>` indicating the location of the file. Finally, an HTTP response from the server may contain the reference.

with the number of distinct domains in an attempt to provide a simple indicator of diversity.

| Criterion | # Files | # Policies | # Distinct domains |
|---|---|---|---|
| All downloads | 8,768 | — | 7,746 |
| XML | 7,899 | — | 7,554 |
| Well-formed XML | 7,675 | — | 7,371 |
| **Schema-validated XML** | **5,905** | **6,182** | 5,673 |

**Table 3.3.** Stages towards syntactical validity

**We define our P3P corpus to consist only of those 6,182 policies that are valid with regard to P3P's XML schema.** For completeness' sake, we mention that some P3P tools tolerate ill-formed or schema-invalid XML to some extent. Such tolerance is tool-specific though, and we do not try to use obviously incorrect policies in this study.

For convenience's sake, Appendix A.1.1 provides additional information about the diversity of the corpus in terms of top-level domains and ODP's website categories. These considerations are inspired by studies of P3P adoption both geographically and in terms of website categories [57, 153, 44, 154]. Adoption and geographic or categorical diversity are not directly of interest in our study, but we do provide such extra diversity data so that others can assess the generality of our results.

### 3.2.3 Leveraged Analyses

Driven by the research questions and our overall background in software language engineering, we leverage several analyses. These are all original analyses as far as P3P is concerned. Except for the analysis of language extensions, these analyses are generally useful in studying language usage in an empirical manner. That is, other domain-specific languages may also be subjected to this scheme.

#### Analysis of vocabulary

We use simple coverage and frequency analysis to determine unused, used, and frequently used vocabulary, thereby addressing the research question of §3.2.1. This analysis is applied to the basic enumeration types of P3P, i.e., purposes, recipients, and retention, but also to P3P's Base Data Schema (BDS).

#### Analysis of constraints

Various resources, including the (latest but provisional) P3P specification, stipulate different additional constraints. We collect and implement these constraints so that we can determine violations by the corpus, thereby providing data for the research question on correct use of P3P; see §3.2.1. We select some of these constraints for the notion of *semantical validity*, which we require for some of the other analyses in the study.

#### Analysis of metrics

We provide different metrics that measure the size of a policy. One metric directly measures the size of a suitable syntactical representation. Another metric abstracts from the representation, and measures instead the size of a normal form that views a policy as the extensional set of facts that it declares. Yet another metric focuses on the size of the data model that is referenced by a policy. Finally, there is metric focused on P3P's finite domains for purposes, recipients, and categories. These different metrics help us to identify significant policies according to the research question of §3.2.1.

#### Analysis of cloning

Clones of complete policies are found using different detection techniques. *Textual clones* are literally identical policy files, and they are necessarily the result of copy-and-paste behavior because of the entity/identity information that is included into each policy. *Syntactical clones* are policies equal in terms of their essential (say, abstract) syntactical structure. *Semantical clones* are policies equal in terms of the extensional set of facts that they declare. These clone types help us studying diversity of policies according to the research question of §3.2.1.

#### Analysis of extensions

We use a simple form of grammar inference to aggregate extensions from the policies in the corpus. We also study the diversity of the websites that host policies with extensions.

## 3.3 The Essence of P3P

This section describes the essence of P3P as far as the study is concerned. (We use the term 'essence' here in reference to a common style used elsewhere in programming language research [190, 166, 184, 28].) Some of the details are subjective in that they do not directly follow from W3C's specification. We point out such subjective elements clearly when we encounter them.

First, we give a brief summary of P3P's platform idea as it clarifies the potential roles of website policies, user preferences, and the process to integrate them. Second, we define an abstract syntax for P3P, which captures the "semantically relevant" part of P3P in a certain sense. This abstract syntax is the foundation for a simple notion of *syntactical equality*. Third, we define a normal form for P3P based on previous work [197]. The normal form of a policy corresponds to the extensional set of facts implied by the policy, thereby giving rise to the notion of *semantical equality*. Finally, we suggest a partial order on normally formed policies so that policies cannot just be tested for equality, but, in fact, they can be compared with regard to the *degree of exposure* (in the sense of privacy). This is helpful, for example, when we study common policies later on.

### 3.3.1 Language versus Platform

P3P, as a *language*, is a non-executable, domain-specific, XML-based language for privacy policies to be declared by websites for the benefit of website users [189, 42]. P3P, as a *platform*, "*enables Web sites to express their privacy practices in a standard format that can be retrieved automatically and interpreted easily by user agents. P3P user agents will allow users to be informed of site practices (in both machine- and human-readable formats) and to automate decision-making.*" [189].

Let us provide additional details here. The entity of a website is supposed to define and publish a privacy policy for the website. The P3P platform suggests that privacy policies adhere to the P3P language, and there is the option of generating human-readable descriptions from P3P policies. The basic scenario is that the user is responsible for consulting a website's privacy policy in order to understand whether to use the site and what data to expose. The more advanced scenario is that the user actually documents privacy preferences in a designated language so that user preferences and website policy can be checked for compliance by a tool. This process can be automated and integrated into the browsers experience [90]. The process may also involve additional elements of negotiation or dispute resolution, but we skip this aspect here because not even the basic use of a language for user preferences and the integration of compliance checking into a browser has seen much adoption in practice [43].

The P3P specification does not commit to a specific language for user preferences, but W3C's *APPEL* [188] is explicitly mentioned as an option; other options have been proposed [14, 15]. Fundamentally, languages for policies versus preferences differ as follows. A policy language is assumed to make *statements about data* being optionally or mandatorily required by the website for certain purposes, to be stored for a certain duration, and to be shared with certain recipients. In contrast, preferences can be thought of as *constraints or rules* to be applied to policies which are hence to be accepted or rejected.

### 3.3.2 Syntax of P3P

In our study, we are only concerned with the formal statements about data collection, use, storage, and sharing. We are usually not concerned with the *entity* that issues the policy or a policy's *consequences*, which are given in natural language. The remaining P3P syntax is described in Fig. 3.1; it can be seen as an abstract syntax derived

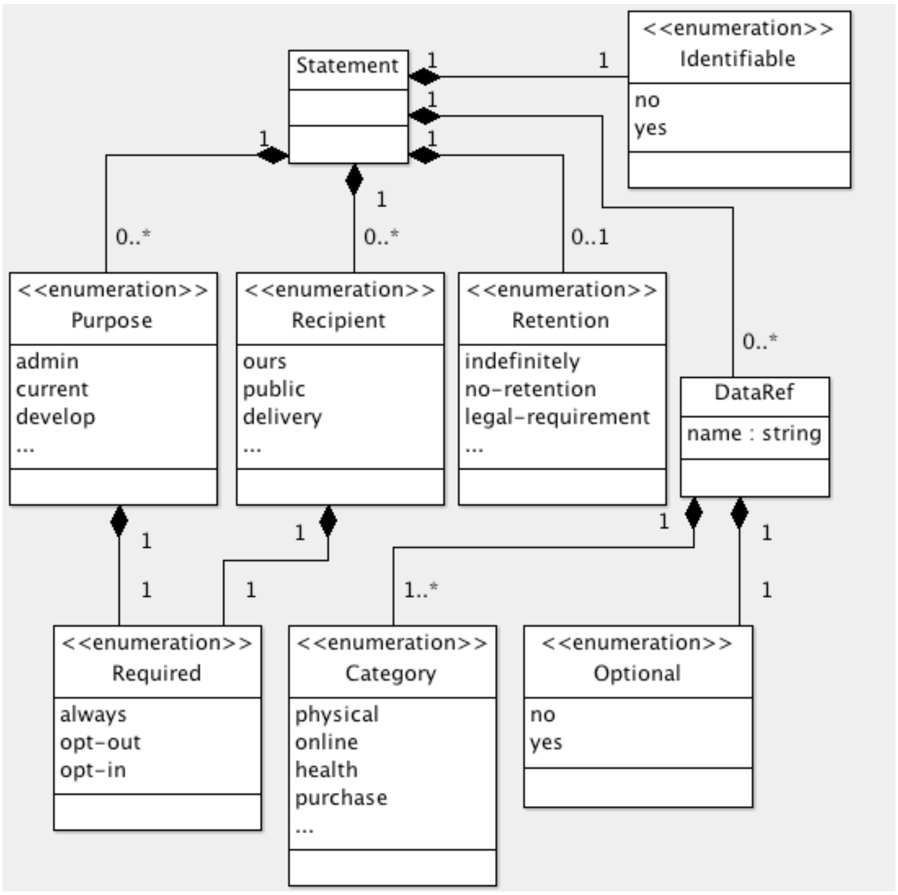


**Figure 3.1.** The study's abstract syntax of P3P

from comprehensive, concrete syntax definitions.[5] It is important to note that this abstract syntax has no counter part in the P3P specification. Instead, we have designed this abstract syntax to precisely include those language elements that are of interest for our purposes.

According to the figure, a P3P policy consists of any number of statements, each of which signifies data references for the collected data, purposes of collecting data, recipients that receive the data, a retention level, which defines how long data is stored, and a few other directives. The figure clarifies that the major syntacti-

[5] Comprehensive, concrete syntax definitions of P3P are available in different forms:

- XSD: `http://www.w3.org/2002/01/P3Pv1.xsd`
- Relax NG: `http://yupotan.sppd.ne.jp/relax-ng/p3pv1.rng`
- RDF: `http://www.w3.org/TR/p3p-rdfschema/`

```
user
  |-name
  |   |-prefix
  |   |-given
  |   |-middle
  |   |-family
  |   |-suffix
  |   |-nickname
  |-bdate
  |   |-ymd.year
  |   ...
  |-login
  |   |-id
  |   |-password
  ...
```

The hierarchy on the left shows part of the Base Data Schema (BDS) of P3P—as it is part of the P3P specification. The shown snippet deals with user data in a narrow sense. Policies typically suffice with data items from the BDS, but policies can also define a Custom Data Schema (CDS) with data items that are specific to the policy. Such schemas are defined in XSD (XML Schema) with extensions for P3P. BDS and CDS only define names and hierarchical organization but no types are assigned to the data items.

**Figure 3.2.** P3P data schema hierarchy (sample from Base Data Schema)

```
<DATA ref="#dynamic.miscdata">
  <CATEGORIES>
    <political/>
  </CATEGORIES>
</DATA>
```

The P3P snippet on the left looks like a regular data reference, but the specific data item in question, i.e., '#dynamic.miscdata, is a 'variable-category data element'. The category 'political' is assigned to it, which proxies for "Membership in or affiliation with groups such as religious organizations, trade unions, professional associations, political parties, etc." [189].

**Figure 3.3.** Variable-category data elements in P3P

cal domains are finite (i.e., enumeration types) and there is no recursion involved. We should add that P3P is somewhat prepared to deal with unforeseen concepts. There is, for example, a purpose called 'other-purpose', and there is a general extension mechanism that we discuss in §3.4.5.

*Data references* either refer to the Base Data Schema (BDS) of P3P or a Custom Data Schema (CDS), which is provided with the policy. The BDS, which is part of the P3P specification[6,7], organizes user data, business data, third-party data, and so-called dynamic data (with diverse roles) in a hierarchical manner; see Fig. 3.2 for an illustration.

Data references can be annotated with *categories* such as 'purchase' or 'health'. In fact, most data references in the BDS are implicitly associated with one or more categories. For instance, user.name is implicitly associated with category 'physical', i.e., physical contact information. Some data elements of the BDS are variable though—specifically in the #dynamic branch. References to such 'variable-category

[6] http://www.w3.org/TR/P3P11/#base_data_structure
[7] http://www.w3.org/TR/P3P11/#schema_detail

data elements' [189] must be explicitly associated with one or more categories when they are used in a policy; see Fig. 3.3 for an illustration. The specification further motivates this mechanism as follows[7]: "In some cases, there is a need to specify data elements that do not have fixed values that a user might type in or store in a repository. In the P3P base data schema, all such elements are grouped under the class of dynamic data. Sites may refer to the types of data they collect using the dynamic data set only, rather than enumerating all of the specific data elements." This means that the hierarchical refinement of the data element is not specified which implies that 'variable-category data elements' are fundamentally different from regular data items.

A *retention level* can be anywhere from 'no retention' to 'indefinite retention'.

Let us explain a few values for *recipients*. The entity in charge of the system is referred to as the recipient 'ours'. (We simplify here the meaning of 'ours', when compared to the P3P specification.[8]) Recipients with equal privacy practices as 'ours' are referred to as 'same'. The recipient 'public' essentially models the fact that collected data may be released to a public forum such as a bulletin board or a public directory.

Let us also explain a few values for *purposes*. The value 'admin' signifies that data is collected for the purpose of the technical support of the website and its computer system. The value 'telemarketing' signifies that the collected information may be used to contact the individual via a voice telephone call for promotion of a product or service.

Purposes and recipients may be qualified by an attribute *required* which regulates whether associated data can be used for a given purpose or shared with a given recipient. These are the possible values for *required*: 'always', 'opt-in' and 'opt-out'. For instance, a purpose with 'opt-out' for a certain data reference means that "data may be used for this purpose unless the user requests that it not be used in this way" [189]. The concrete syntax assumes 'always' as default.

Data references are qualified by an attribute *optional*; the values are 'yes' versus 'no'—the user of the website may or may not omit the data, respectively. The concrete syntax assumes 'no' (i.e., non-optional) as default. Finally, statements are qualified by an attribute *identifiable*; the values are 'yes' versus 'no'. The concrete syntax uses as an extra element `<NON-IDENTIFIABLE/>` to signify non-identifiability; hence, all data is identifiable by default. When the statement is labeled with *identifiable*='no', then this means either that "there is no data collected under this STATEMENT, or that all of the data referenced by that STATEMENT will be anonymized upon collection."[9]

[8] "Ourselves and/or entities acting as our agents or entities for whom we are acting as an agent: An agent in this instance is defined as a third party that processes data only on behalf of the service provider for the completion of the stated purposes. (e.g., the service provider and its printing bureau which prints address labels and does nothing further with the information.)" [189]

[9] `http://www.w3.org/TR/P3P11/#NON-IDENTIFIABLE`

Data-centric style

```
<POLICY>
  <STATEMENT>
    <PURPOSE><admin/><current/><develop/></PURPOSE>
    <RECIPIENT><ours/></RECIPIENT>
    <RETENTION><indefinitely/></RETENTION>
    <DATA-GROUP>
      <DATA ref="#dynamic.clickstream"/>
   </DATA-GROUP>
  </STATEMENT>
  <STATEMENT>
    <PURPOSE><admin/><current/><develop/></PURPOSE>
    <RECIPIENT><ours/></RECIPIENT>
    <RETENTION><indefinitely/></RETENTION>
    <DATA-GROUP>
     <DATA ref="#dynamic.http"/>
    </DATA-GROUP>
  </STATEMENT>
</POLICY>
```

Purpose-centric style

```
<POLICY>
  <STATEMENT>
    <PURPOSE><admin/></PURPOSE>
    <RECIPIENT><ours/></RECIPIENT>
    <RETENTION><indefinitely/></RETENTION>
    <DATA-GROUP>
      <DATA ref="#dynamic.clickstream"/>
     <DATA ref="#dynamic.http"/>
   </DATA-GROUP>
  </STATEMENT>
  <STATEMENT>
    <PURPOSE><current/></PURPOSE>
    ... continue as with the previous statement ...
  </STATEMENT>
  <STATEMENT>
    <PURPOSE><develop/></PURPOSE>
    ... continue as with the previous statement ...
  </STATEMENT>
</POLICY>
```

**Figure 3.4.** Additional representations of the sample ‘logging only’ (refer to Fig. 2.6)

## 3.3.3 A Normal Form

When we study policies in terms of metrics, cloning, and others, we prefer to use a normal form that neutralizes syntactical diversity. That is, the normal form of a policy corresponds to the extensional set of facts implied by the policy, thereby giving rise to the notion of *semantical equality*.

### Semantical equality

Consider again the ’logging only’ policy in Fig. 2.6. Now consider the two variations of the same policy in Fig. 3.4. The original formulation used a single statement; the two variations use multiple statements. One can consider the original formulation as

- r-purpose($\underline{\text{data,purpose}}$,required)
- r-recipient($\underline{\text{data,recipient}}$,required)
- r-retention($\underline{\text{data}}$,retention)
- r-data($\underline{\text{data}}$,optional,identifiable)
- r-category($\underline{\text{data,category}}$)

**Figure 3.5.** Relational schema for P3P—adopted from YuLA04

a factored formulation in that two data references and three purposes are grouped together in a single statement whereas the first variation maintains the grouping of purposes while it splits up the group of data references into multiple statements—likewise for the second variation. We assume that all these three formulations are semantically equivalent in that they declare the same extensional set of facts; *the policies admit the same data to be collected, for the same purposes, to be shared with the same recipients, and subject to the same retention.* The P3P specification does not directly describe such semantical equality, but previous research on the formal semantics of P3P [197] formalizes such an equality based on an appropriate normal form.

Following [197], the extensional set of facts can actually be represented in P3P syntax itself. We say that a policy is in normal form, if all statements are restricted to single data references, purposes, recipients, and categories; see the "*" cardinalities in Fig. 3.1. All the variations on 'logging only' are not in normal form because some grouping is used; the normal form requires six statements. Normalization (i.e., derivation of the normal form) is achieved by replacing each statement with one or more groups by a number of statements exercising all combinations of elements from the groups; see [197] for the trivial normalization algorithm.

## A relational schema for the normal form

Further following [197], we use a relational schema to specify extensional sets of facts more concisely and to precisely capture key constraints; see Fig. 3.5 for the schema. There are different relations to describe—in a point-wise manner—what purposes, recipients, retention levels, and other properties are associated with the individual data references.

Our schema deviates from the one in [197] only in so far that we take into account the possibility of non-identifiable data. Hence, we added a column to r-data that tracks whether a data-reference is identifiable.

The primary keys are $\underline{\text{underlined}}$ in the figure. The key constraints model consistency constraints for a policy. We say that a policy is *semantically valid* if it has a normal form such that the key constraints are maintained. For instance, within a given policy, a data reference cannot be both optional and non-optional. That is, optionality is a property associated with a data reference; it cannot be specialized per purpose or recipient.

The P3P specification is not fully conclusive on the appropriateness of key constraints. On the issue of optionality, we can read, for example: "optional indicates

| | |
|---|---|
| r-purpose | (#dynamic.clickstream, admin, always) |
| | (#dynamic.clickstream, current, always) |
| | (#dynamic.clickstream, develop, always) |
| | (#dynamic.http, admin, always) |
| | (#dynamic.http, current, always) |
| | (#dynamic.http, develop, always) |
| r-recipient | (#dynamic.clickstream, ours, always) |
| | (#dynamic.http, ours, always) |
| r-retention | (#dynamic.clickstream, indefinitely) |
| | (#dynamic.http, indefinitely) |
| r-data | (#dynamic.clickstream, no, yes) |
| | (#dynamic.http, no, yes) |
| r-category | — |

**Figure 3.6.** The 'logging only' policy in normal form according to Fig. 3.5

whether or not the site requires visitors to submit this data element to access a resource or complete a transaction".[10] This explantation may be interpreted as general requirement as opposed to a purpose/recipient-specific requirement.

It is important to note that the normal form of [197] with its key constraint was never confirmed by W3C. Other options of semantics are discussed in [197]. For instance, one can think of a semantics such that optionality of a data item is specific to purpose. We stick here to the semantics that is fully developed and ultimately favored in [197]. This may lead to an under-approximation of the set of all semantically valid policies, which means that perhaps more policies than necessary are excluded from some analysis.

As an illustration, Fig. 3.6 shows the normalized 'logging only' sample policy as an instance of the schema—as opposed to using concrete P3P syntax. We relate each data item to corresponding values of purpose, recipient, retention, etc. For each purpose and recipient we restored the default value of attribute *required* (which is "always"), and for each data item we restored the default value of attribute *optional* (which is "no"); we also encode whether a data item is identifiable, which it is (because the NON-IDENTIFIABLE element is not used in the concrete syntax).

## Folding and subsumption

The notion of normal form, which we have described so far by adoption of [197], can be conservatively improved. That is, there are policies that are semantically equivalent in an intuitive sense, but their normal forms are not (yet) the same. Complications arise from the fact that the P3P schemas (both BDS and any CDS) organize data items in a hierarchical manner. Accordingly, we advance normalization to apply the two following rules whenever possible.

[10] `http://www.w3.org/TR/P3P11/#DATA`

| | Before folding | After folding |
|---|---|---|
| r-purpose | (#user.login.id, current, always)<br>(#user.login.password, current, always) | (#user.login, current, always) |
| r-recipient | (#user.login.id, ours, always)<br>(#user.login.password, ours, always) | (#user.login, ours, always) |
| r-retention | (#user.login.id, indefinitely)<br>(#user.login.password, indefinitely) | (#user.login, indefinitely) |
| r-data | (#user.login.id, no, yes)<br>(#user.login.password, no, yes) | (#user.login, no, yes) |
| r-category | — | — |

**Figure 3.7.** Example of folding in normal form

| | Before subsumption | After subsumption |
|---|---|---|
| r-purpose | (#user.name, current, always)<br>(#user.name.given, current, always) | (#user.name, current, always) |
| r-recipient | (#user.name, ours, always)<br>(#user.name.given, ours, always) | (#user.name, ours, always) |
| r-retention | (#user.name, indefinitely)<br>(#user.name.given, indefinitely) | (#user.name, indefinitely) |
| r-data | (#user.name, no, yes)<br>(#user.name.given, no, yes) | (#user.name, no, yes) |
| r-category | — | — |

**Figure 3.8.** Example of subsumption in normal form

We apply *folding* so that a set of data references with a common parent (in the sense of the hierarchical organization) and the same properties for purposes, recipients, retention, and categories is replaced by a data reference to the parent, if the set is complete, i.e., it contains all children of the given ancestor. Likewise, we apply *subsumption* so that a data reference is omitted if there is already a data reference to the ancestor with the same properties.

For instance, in Fig. 3.7, data references *user.login.id* and *user.login.password* have the same values in all tuples; furthermore, these two references are the only leaves of the node *user.login* in the BDS. Thus, we can apply folding and reduce the number of tuples, introducing the grouping node instead of its leaves. In Fig. 3.8, data reference *user.name.given* is a leaf of the node *user.name* in the BDS; furthermore, it has the same properties as the parent node. Thus, we can apply subsumption and reduce the number of tuples by removing redundant ones.

```
<POLICY>
  <STATEMENT>
    <PURPOSE><admin/><current/><develop/></PURPOSE>
    <RECIPIENT><ours/></RECIPIENT>
    <RETENTION><legal-requirement/></RETENTION>
    <DATA-GROUP>
      <DATA ref="#dynamic.clickstream"/>
   </DATA-GROUP>
  </STATEMENT>
</POLICY>
```

**Figure 3.9.** A variation on "logging only" with less exposure

## 3.3.4 Degree of Exposure

By now, we are able to test for semantical equality by checking that the given policies have the same normal form. If we represent the normal form as relations, then the test for semantical equality essentially is a test for equality of relations (i.e., sets of tuples). Conceptually, these relations declare exposure in the sense of privacy—what data can be collected, for what purposes, with whom can it be shared, and what level of retention must be provided.

Interestingly, we can compare policies by leveraging (a special variation on) the subset partial order on the normal-form relations. We refer to this partial order as *degree of exposure*. Such a partial order is helpful in comparing similar policies and understanding diversity within the corpus; see §3.4.4 in particular.

Consider the policy in Fig. 3.9. It lacks the data reference #dynamic.http; hence, data collection is limited to information typically stored in Web-server access logs without the additional information from the http protocol as it is specifically covered by #dynamic.http. The variation also differs with regard to the retention level: it specifies 'legal-requirement' instead of 'indefinitely'; hence, data storage is more limited. Both differences, by themselves and in combination imply that there is 'less exposure'.

The example with the different retention levels gives rise to a more general idea: a partial order can be imposed on the various enumeration types of P3P. The idea is that the different values are partially ordered in terms of degree of exposure. See Fig. 3.10 for our proposal; exposure increases as one climbs up the Hasse diagrams. We overload the symbol '$\leq_{sem}$' to denote these partial orders.

Consider, for example, optionality: if a user is not obliged to provide certain data, then there is less exposure (say, more privacy is maintained), compared to the case when such data is required.

The P3P specification does not describe a partial order like ours. The proposal is meant to be relatively conservative in that we do not impose an order on specific values unless the specification suggests it. In some cases, we pragmatically declare that some values imply the same degree of exposure since we could not see that they must be incomparable. Some of the specific decisions in our proposal are presumably self-evident; others are more tedious; see Appendix A.1.2 for a more detailed motivation. The impact of our particular definition is very limited in so far that we only use it in

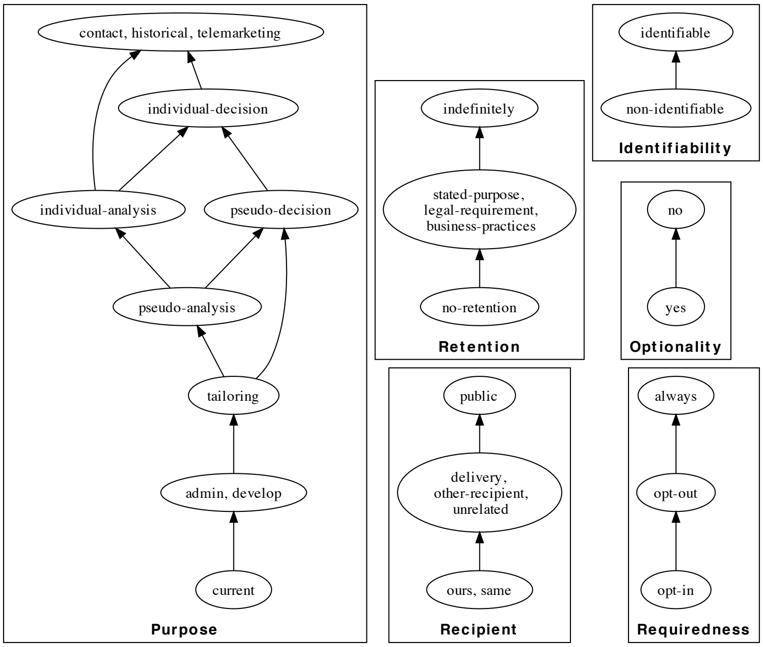


**Figure 3.10.** Partial orders for the degree of exposure per enumeration type of P3P

our informal discussion of (relationships between) common policies (see §3.4.4 in particular) and a short summary on remaining diversity among semantically distinct policies in Appendix A.1.2.

We define a binary relation '$\leq_{sem}$' on instances of the relational schema for the normal form. Informally, given two instances $i$ and $i'$, i.e., two extensional sets of facts, we say that $i \leq_{sem} i'$ if $i$ is a subset of $i'$ modulo replacing equality of individual facts by component-wise application of '$\leq_{sem}$' as defined by Fig. 3.10. Clearly, we compare facts from the same relation (say, table) and with equal data items.

Formally, '$\leq_{sem}$' is defined to be the smallest relation closed under the following rules. Let $i$, $i'$, $i''$ range over instances of the relational schema.

[Reflexivity] If $i =_{sem} i'$, then $i \leq_{sem} i'$.
[Transitivity] If $i \leq_{sem} i'$ and $i' \leq_{sem} i''$, then $i \leq_{sem} i''$.
[Subset] If all relations of $i$ are subsets of the corresponding relations of $i'$, then $i \leq_{sem} i'$.
[Value] If all relations of $i$ are equal to the corresponding relations of $i'$ except for tuples $t$ and $t'$ (from corresponding relations) of $i$ and $i'$, respectively such that $t$ and $t'$ differ in values $v$ and $v'$ for purpose, recipient, retention, identifiability,

| | |
|---|---|
| DataRef | 40979 |
| Purpose | 31254 |
| Category | 20896 |
| Recipient | 12496 |
| Optional | 10823 |
| Retention | 10623 |
| Statement | 10623 |
| Required (purpose) | 4171 |
| Identifiable | 1718 |
| Required (recipient) | 1155 |

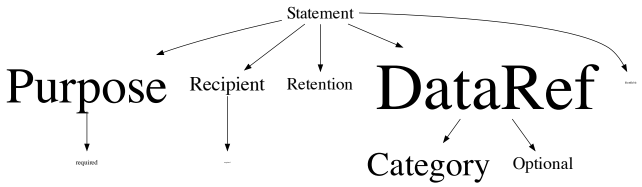


**Figure 3.11.** Frequency of use for P3P's syntactical domains

optionality, or requiredness such that $v \leq_{sem} v'$ (according to Fig. 3.10), then $i \leq_{sem} i'$.

It should be noted that Fig. 3.10 makes some values equal. The definition omits tedious issues of folding and subsumption.

## 3.4 Analyses of the Study

In this section, we address the research questions of the study (§3.2.1) by analyses of vocabulary, language constraints, metrics, clones, and language extensions. Each subsection is dedicated to one of these analyses and its underlying research question. Analyses may break down into sub-analyses. All (sub-) analyses are subject to the following scheme: we motivate the analysis and describe technicalities, we list output data, and we discuss the results while referring to the research question again.

### 3.4.1 Analysis of Vocabulary

The underlying research question is "What part of the vocabulary is used?"; see §3.2.1. The main goal is to help with profiling P3P such that the core language is identified and simplifications or designated support for the profile may be possibly inferred.

### Frequency of use for P3P's syntactical domains

In Fig. 3.11, we list frequency of use for the syntactical domains according to the abstract syntax of Fig. 3.1. We do not count any defaults (such as non-optional data items), even if they are specified explicitly. We also visualize these numbers where we exploit the fact that P3P has a very simple syntactical structure. There is no recursion. Hence, we can show the compositional structure of policies with a tree and frequencies are translated into node size by using tag cloud-like scaling techniques [106].

We observe that most statements do not use a NON-IDENTIFIABLE element. We also observe "popularity" for optionality and opt-in/out. That is, data is said to be optional in a significant number of cases; purposes are available for opt-in/out in a significant number of cases, but recipients are hardly available for opt-in/out.

### Frequency of use for values of P3P's enumeration types

Given P3P's reliance on a privacy vocabulary in the form of several finite domains (say, enumeration types), we can get insight into the P3P profile by studying frequency of use of the values of those domains; frequencies of usage are summarized in Fig. 3.12.

We observe that the unspecific purpose 'current' is highly frequent and purposes reminiscent of web logging are highly frequent as well. 'Ours' dominates the domain of recipients; other purposes are at least a factor two less frequent. The dominance of 'ours' was to be expected because each logically consistent policy must involve 'ours'. There is substantial use of the nonspecific purpose 'other-purpose', which suggests that a richer set of purposes or author-defined purposes may be needed. We also observe that indefinite retention is the most popular—which may be explainable, in part, through the popularity of weblogs, which typically assume indefinite retention. The retention level 'business-practices' is also relatively popular—these practices are not specified though in P3P, which suggests that extra language support could be useful.

### Frequency of use for data items from the BDS

P3P's Base Data Schema (BDS) expresses the assumptions of the P3P designers as to what private data website users and policy authors may possibly care about. Fig. 3.13 lists frequency of use for popular data references. Fig. 3.14 shows the distribution of numbers of references in a double-logarithmic graph.

### Discussion

Fig 3.14 confirms that few data items from the BDS account for most of the data references. The *dynamic* branch assumes the five most frequent positions in Fig. 3.13.

| Purposes | |
|---|---|
| develop | 6277 |
| current | 6121 |
| admin | 5140 |
| tailoring | 3028 |
| pseudo-analysis | 2059 |
| contact | 1779 |
| pseudo-decision | 1562 |
| individual-analysis | 1465 |
| individual-decision | 1405 |
| historical | 979 |
| other-purpose | 762 |
| telemarketing | 677 |

| Recipients | |
|---|---|
| ours | 9635 |
| delivery | 1571 |
| same | 583 |
| other-recipient | 346 |
| public | 214 |
| unrelated | 147 |

| Retentions | |
|---|---|
| indefinitely | 4336 |
| business-practices | 2617 |
| stated-purpose | 1742 |
| no-retention | 669 |
| legal-requirement | 467 |

| Implicitly declared categories | |
|---|---|
| navigation | 10247 |
| computer | 9048 |
| demographic | 8787 |
| physical | 8164 |
| online | 4240 |
| interactive | 3466 |
| uniqueid | 594 |

| Explicitly declared categories | |
|---|---|
| uniqueid | 3675 |
| navigation | 2328 |
| online | 2222 |
| state | 2022 |
| physical | 1989 |
| purchase | 1411 |
| computer | 1396 |
| demographic | 1289 |
| preference | 1268 |
| interactive | 1144 |
| content | 853 |
| other-category | 428 |
| location | 415 |
| financial | 253 |
| government | 94 |
| health | 58 |
| political | 51 |

**Figure 3.12.** Frequency of use for values of P3P's enumeration types

After 15 positions, frequency is a factor ten reduced. There are several data references to the *dynamic* and *user* branches, and one data reference to the *thirdparty* branch among those top 15. The *business* branch does not show up in Fig. 3.13.

The particular *dynamic* data references imply that policies in the corpus declare collection of data such that the highest frequency is about web navigation and interaction whereas the second and third are about variable-category data elements *cookies* and *miscdata*. Hence, we see that policies use variable-category data elements heavily. As a result, policies switch constantly between regular data references with

| | |
|---|---|
| dynamic.cookies | 3351 |
| dynamic.clickstream | 3204 |
| dynamic.http | 3058 |
| dynamic.miscdata | 2753 |
| dynamic.searchtext | 1816 |
| user.name | 995 |
| dynamic.clientevents | 726 |
| dynamic.interactionrecord | 579 |
| user.home-info | 510 |
| user.home-info.online.email | 476 |
| user.name.given | 418 |
| user.name.family | 418 |
| user.business-info.online.email | 387 |
| user.business-info | 364 |
| thirdparty.name | 361 |
| dynamic.http.useragent | 333 |
| user.home-info.telecom.telephone | 280 |
| user.bdate | 277 |
| user.home-info.postal | 262 |
| thirdparty.business-info | 237 |
| user.gender | 210 |
| user.login.password | 149 |
| user.login.id | 141 |
| user.home-info.telecom.mobile | 140 |
| dynamic.clickstream.clientip | 131 |
| dynamic.http.referer | 122 |
| thirdparty.home-info | 114 |
| user.business-info.telecom.telephone | 90 |
| user.business-info.postal.name | 72 |
| dynamic.clickstream.timestamp | 72 |
| user.jobtitle | 71 |
| dynamic.clickstream.other.httpmethod | 69 |
| dynamic.clickstream.uri | 68 |
| dynamic.clickstream.clientip.fullip | 66 |
| user.home-info.postal.postalcode | 53 |
| dynamic.clickstream.other.statuscode | 50 |
| user.home-info.postal.stateprov | 49 |
| dynamic.clickstream.other.bytes | 49 |
| user.login | 42 |
| user.home-info.postal.city | 41 |
| user.home-info.postal.country | 41 |
| ... | |

**Figure 3.13.** Frequency of use for data items from the BDS

implicitly associated categories and references to variable-category data elements with explicitly associated categories.

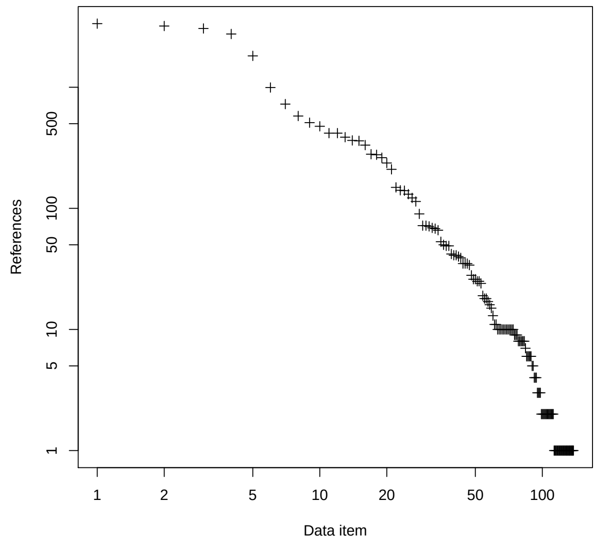


**Figure 3.14.** Distribution of frequencies for BDS data references

## Coverage of P3P's Base Data Schema

The BDS has 324 nodes. Overall, 250 (i.e., 77.16 %) of all nodes from the schema are referenced at least once. Fig. 3.16 shows how coverage of the BDS (quickly) drops if coverage of a node requires an increasing number of syntactically distinct policies to use it.

P3P's BDS expresses the assumptions of the P3P designers as to what hierarchical decomposition of private data both website users and policy authors may possibly care about. Fig. 3.15 shows the complete hierarchical structure of the BDS where the thickness of the edges represents the frequency of referring to the target node of the edge. (Again, we use tag cloud-like scaling techniques.) Hence, the most frequent data references of Fig. 3.14 correspond to the nodes (inner nodes or leaves) with thickest, incoming edges.

## Discussion

Fig. 3.16 shows that the coverage of the BDS only appears to be high. A relatively small number of policies is responsible for most of the coverage. Hence, it could be possible to refactor the BDS of a future policy language to be much simpler.

Many nodes are not referenced at all; see the dotted edges. The unreferenced nodes are typically leaf nodes—as the visualization shows. Hence, one may argue

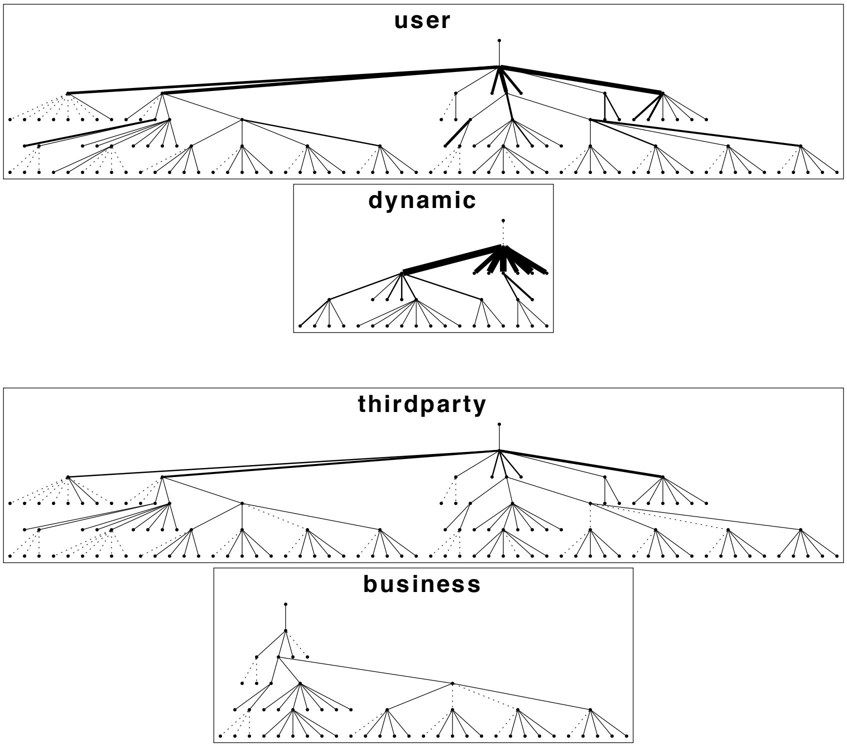


**Figure 3.15.** Coverage of P3P's Base Data Schema

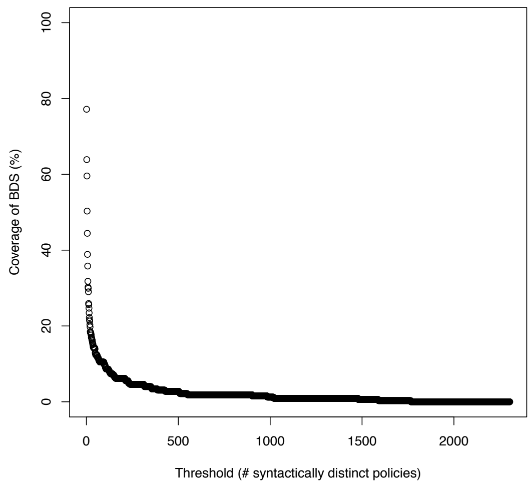
that the BDS could have been kept simpler so that it only focuses on the essential ontology of private data without breaking down all data into 'primitive' items.

## 3.4.2 Analysis of Constraints

The underlying research question is "Is the language correctly used?"; see §3.2.1. The main goal is to understand what correctness issues, if any, exist in the wild, and to find plausible explanations or even to make recommendations for future policy languages. We investigate correctness here on the grounds of language constraints for P3P as they are available from different sources.

We were interested in constraints that add to the basic XML schema-based validation for P3P. We identified two kinds of constraints in the P3P specification. First, each data reference must be resolvable to a data item in a P3P data schema. Second, the (latest and provisional) specification stipulates sanity checking rules[11] (say, co-

[11] `http://www.w3.org/TR/P3P11/#ua_sanity`



| Threshold | BDS coverage |
|---|---|
| 1 | 77.16 % |
| 2 | 63.88 % |
| 3 | 59.56 % |
| 4 | 50.30 % |
| 5 | 44.44 % |
| 6 | 38.88 % |
| 7 | 35.80 % |
| 8 | 31.79 % |
| 9 | 30.24 % |
| 10 | 29.93 % |

**Figure 3.16.** Coverage of BDS with a threshold on the number of syntactically distinct policies

herence constraints). We obtained further coherence constraints as well as the key constraints of §3.3.3 from Yu et al. article on the 'relational semantics' of P3P [197].

## Key constraints

The key constraints of the relational schema for P3P's normal form, as of §3.3.3, immediately constrain the abstract syntax of §3.3.2. We check these constraints naturally as we normalize policies by deriving relations from the P3P statements. Whenever we insert tuples into the relations, we admit identical tuples, but we do not admit tuples that violate key constraints. (This is part of the normalization algorithm [197].) We refer to Table 3.4 for violation counts for the different key constraints. We note again that the key constraints are potentially debatable; see the discussion in §3.3.3.

| Relation | # Policies | # Syn. distinct policies |
|---|---|---|
| r-data | 448 | 248 |
| r-retention | 139 | 104 |
| r-purpose | 50 | 33 |
| r-recipient | 7 | 7 |
| **Any of them** | **544** (of 6,182) | **312** (of 2,304) |

**Table 3.4.** Key-constraint violations

## Discussion

Table 3.4 shows that key constraints are violated frequently. For instance, a violation of the key constraint for r-retention would mean that a policy makes different retention promises (such as ‘no retention’ versus ‘indefinite retention’) for the same data, which could be sensible if different purposes or recipients are allowed to provide individual retention. Given the non-standardized status of the normal form, we cannot conclusively rule these violations as definite instances of incorrect use of P3P, but they definitely indicate a problem with the status of P3P formalization and validation.

Consider also the high number of violations of r-data’s key constraint. Users seem to assume that a data reference may be used with different optionality values for varying purposes. As we discussed in §3.3.3, the specification may suggest otherwise. This observation suggests that either fine-grained optionality should be explicitly provided by future policy languages or the specification should be clarified and validation should effectively enforce the restriction.

More generally, the lifecycle for P3P policies is too weakly supported. Basic schema-based validation is insufficient to find more subtle issues like the key constraints at hand or any other key constraints for that matter. The fact that key constraints are not systematically listed in the specification does not help. Future policy language should leverage stronger specifications and a lifecycle with effective validation.

We must say that we are undecided as to whether the r-data key constraint is ever to be enforced for variable-category data elements such as #dynamic.miscdata; see §3.3.2. The challenge is here that the same data element could be used multiple times in the same policy with different categories for different parts of the data model, thereby giving support to the idea that optionality may differ per occasion. This mechanism may need to be reconsidered and more strongly specified in future policy languages.

## Hierarchy constraints

There is a natural extension of r-data’s key constraint that takes into account the hierarchical organization of a data schema (typically the BDS), as suggested in [197]. That is, given two data references $r$ and $r'$ such that $r$ is a prefix of $r'$, then explicit declarations of optionality for $r$ and $r'$ must agree. For example, if a policy declares *user.name* as being non-optional, then *user.name.given* cannot be declared as being optional.

We refer to Table 3.5 for violation counts for the hierarchy constraint. We also show counts for applying the normalization rules for folding and subsumption as of §3.3.3 as these rules are clearly concerned with the hierarchy in a data schema.

## Discussion

Hierarchical inconsistency appears to be a small problem. There are considerably more applications though of the rules for folding and subsumption. While we do not

| Issue | # Policies | # Syn. distinct policies |
|---|---|---|
| Inconsistency | 17 | 15 |
| Folding | 48 | 32 |
| Subsumption | 167 | 105 |

**Table 3.5.** Hierarchy-constraint violations and applications of normalization rules

count these applications as constraint violations, they indicate problems nevertheless because they may proxy for redundant or unnecessarily verbose policies, thereby suggesting again reconsideration of the data model as well as the degree of validation so that such verbosity or redundancy may be reduced.

## Data-schema constraints

Data references may basically be inconsistent with regard to data schemas (BDS or CDS, if present). This is comparable to 'undeclared variable' issues in type checking for programming languages. We classify the issues in more detail; see Table 3.6 for an itemization. We systematically discovered these issues in developing a name resolver for P3P. Here we note that resolved P3P policies are needed, for example, to check hierarchy constraints (see §3.4.2) and to compute a data-schema-related metric (see §3.4.3).

| Issue | # Policies | # Syn. distinct policies |
|---|---|---|
| BDS legacy | 580 | 358 |
| Unresolvable data reference | 115 | 85 |
| Missing CDS | 11 | 7 |
| CDS legacy | 203 | 157 |
| **Any of them** | **848** (of 6,182) | **579** (of 2,304) |

**Table 3.6.** Data-schema constraint violations

The 'BDS legacy' issue refers to the problem that policies use the obsolete BDS of P3P 1.0 where they are supposed to use BDS of P3P 1.1 instead: "User agents are only required to validate P3P 1.1 policy data elements according to a P3P 1.1 data schema." [12] The 'Unresolvable data reference' issue refers to the problem that policies refer to data references that are simply not declared by the BDS. The 'Missing CDS' issue refers to the problem that a linked CDS is non-resolvable, and 'CDS legacy' issue refers to the problem that policies use an obsolete format for CDS where they are supposed to XSD: "Web sites using custom data schemas MUST publish these schemas in P3P1.1 format only." [12]

[12] `http://www.w3.org/TR/P3P11/#Data_Schemas_back`

## Discussion

Again, the practice of P3P validation is clearly shown to be insufficient. In particular, the attempt to reduce P3P validation to straightforward XML schema-based validation is shown to be ineffective. For example, because P3P schemas define a hierarchical namespace for data items and XML schema-based validation cannot be directly used to fully enforce correct use of data references. Further, the policies in the corpus do not use XML Schema (XSD) for their Custom Data Schemas. One reason for this common violation may be that many policies have been on the web before the revised P3P specification started to require XSD, and no update was applied to the policies.

## Coherence constraints

It is relatively easy to see that there exist certain coherence constraints for combining purposes, recipients, data references and others in a given policy. For instance, a policy would give a vacuous sense of privacy if it combined recipient ‘public’ with any retention other than ‘indefinitely’ because retention cannot be effectively terminated for data that has been released to the public. We extracted coherence constraints from the P3P specification, and our key resource [197] on the relational schema for P3P suggest several coherence constraints as well. We implemented the constraints and checked the corpus. The findings are listed in Table 3.7.

| Source | Constraint | # Policies | # Syn. distinct policies |
|---|---|---|---|
| [197] | ‘ours ...’ | 77 | 45 |
| [197] | ‘public ...’ | 40 | 24 |
| [197] | ‘historical ...’ | 6 | 6 |
| [189] | ‘contact’ | 71 | 47 |
| [189] | ‘telemarketing’ | 353 | 107 |
| [189] | ‘individual-analysis’ | 451 | 94 |
| [189] | ‘individual-decision’ | 432 | 82 |
| | **Any of them** | **898** (of 6,182) | **264** (of 2,304) |

**Table 3.7.** Coherence-constraint violations

For brevity, we explain only a few constraints here. Overall, constraints from [197] impose a common-sense coherence between the elements of a P3P statement. For example, there is a constraint *‘public recipient $\Rightarrow$ indefinite retention’* which proxies for the problem that we described above: a policy that associates the *public* recipient with a data reference should accordingly admit indefinite retention for this data reference. The sanity checks from the specification [189] focus on common-sense coherence between stated purposes and categories of the mentioned data. That is, the constraints express that data of a certain category is needed for a given purpose. For example, the ‘contact’ constraint says that a P3P statement with purpose ‘contact’ should contain at least one data element from the categories ‘physical’ or ‘online’. (Otherwise, it may just be impossible to actually contact the person.)

## Discussion

Again, the practice of P3P validation is clearly shown to be insufficient. However, it must be noted that the P3P specification does not even attempt a comprehensive suite of coherence constraints. Such constraints would not only be useful for validation; they would also be helpful in understanding the P3P vocabulary as such.

| **Constraint category** | **# Policies** | **# Syn. distinct policies** |
|---|---:|---:|
| Key | 544 | 312 |
| Hierarchy | 17 | 15 |
| Data-schema | 848 | 579 |
| Coherence | 898 | 264 |
| **Any of them** | **2,193** (of 6,182) | **1,083** (of 2,304) |

**Table 3.8.** Summary of constraint violations

## Summary of constraint violations

All kinds of violations are summarized in Table 3.8. Some of the subsequent analyses operate on the normal form. Hence, they can only deal with policies that do not violate the key constraints of the normal form. We refer to those policies as being *semantically valid.* All the other violations do not limit any of our subsequent analyses. For clarity, we summarize:

- # Syntactically valid policies: 6,182 (i.e., all policies in the corpus)
  - Syntactically distinct policies thereof: 2,304
- # Semantically valid policies: 4,869 (i.e., all policies with a normal form)
  - Semantically distinct policies thereof: 1,385[13]

[13] For precision’s sake, this number is based on the semantical equality as introduced in §3.3.3 without taking into account extra equations due to ‘$\leq_{sem}$’ as of §3.3.4. However, the difference is insignificant. That is, there is 1 policy that is unequal to any other policy at the level of normal forms while it is equal to another policy when taking into account Fig. 3.10.

### 3.4.3 Analysis of Metrics

The underlying research question is “What is a significant policy?”; see §3.2.1. The main goal is to identify significant policies, that is, policies which provide insight into the underlying web-based system. We are going to try different, straightforward metrics to this end. Ultimately, we consider those metrics to be useful, if they point to policies that we can manually validate to be significant. We do not use metrics in any sophisticated manner of the kind that we plan to validate hypotheses of correlation between metrics and other properties of policies or the underlying web-based systems.

#### Syntactical size

An obvious starting point is a metric for the *syntactical size*, denoted as $SYN(\cdot)$, which is based on the abstract syntax of P3P as defined in §3.3.2. We compare this metric with the Lines-of-Code metric (LOC) or its variation without comments (NCLOC) that is a popular, basic size metric for programs. LOC/NCLOC are not directly applicable to P3P because P3P is primarily an XML language. Hence, $SYN(\cdot)$ is defined as a tree-based node count.

Given a policy $P$, we define $SYN(P)$ as the *node count* of its tree-based representation according to P3P’s abstract syntax of Fig. 3.1. That is, we count nodes for statements, purposes, recipients, retention levels, data references, (explicitly declared) values for requiredness, optionality, non-identifiability, and categories. In particular, we do not count the description of the entity and the policy’s consequences in any way.

For instance, let us determine the syntactical size for different variations on ‘logging only’ policy. We have that $SYN(Fig.\ 2.6)$ equals 8; there is one statement, three purposes, one recipient, one retention, and two data references. Further:

- $SYN$(Data-centric style in Fig. 3.4) equals 14.
- $SYN$(Purpose-centric style in Fig. 3.4) equals 18.

| Min | 1st Q | Median | Mean | 3rd Q | Max |
|---|---|---|---|---|---|
| 2 | 11 | 20 | 25.28 | 36 | 245 |

**Table 3.9.** The $SYN(\cdot)$ scale

Table 3.9 shows that the distribution of $SYN(\cdot)$ is shifted to small node counts; see median and quartiles. We contend that P3P policies are of trivial size, when compared to program sizes of general purpose or domain-specific programming languages.

## Semantical size

Since the abstract syntax is known to enable much representational diversity for semantically equivalent policies, we may attempt a metric that abstracts from such diversity. Thus, let us consider *semantical size*, denoted as $SEM(\cdot)$ as follows. Given a policy $P$ with the normal form value $P'$ (i.e., a set of tuples that populate the relational schema of §3.3.3), we compute the semantical size of $P$ as the straight sum of the numbers of tuples in the various relations r-purpose, r-recipient, r-retention, r-data, and r-category.

Conceptually, there is a deeper argument for considering such a metric. That is, we can view the normal form as the most discrete representation of all the privacy statements in a policy (modulo folding and subsumption). Intuitively, we can compare such a metric—very roughly—with McCabe's cyclomatic complexity for programming languages [132] in so far that both metrics are concerned with counting decisions.

Let us revisit the 'logging only' policy. $SEM$(Fig. 3.6) equals 12, i.e., there are 12 tuples in the tables for the normal form. All syntactical variations on 'logging only' have the same normal form and hence the same semantical size.

| Min | 1st Q | Median | Mean | 3rd Q | Max |
|---|---|---|---|---|---|
| 0 | 10 | 24 | 31.75 | 41 | 352 |

**Table 3.10.** The $SEM(\cdot)$ scale

Table 3.10 shows that the distribution of $SEM(\cdot)$ is equally shifted to small sizes, when compared to $SYN(\cdot)$. However, it is easy to confirm that syntactical and semantical sizes of policies do not correlate. The distributions of the two size metrics are shown in Fig. 3.17. Policies are ordered by syntactical size. Both metrics are normalized to the $[0,1]$ range. The policies at the tail with low syntactical size have a semantical size of 0. Semantical size is only computed for semantically valid policies. (The invalid policies have scattered syntactical sizes, and hence, their absence cannot be 'spotted' in Fig. 3.17.) Such non-correlation should not be surprising; we merely show it here for illustration.

If we group all policies with the same syntactical size and compare the different semantical sizes per syntactical size, then we find that a maximum ratio of 22.08 between the smallest and the largest semantical size in the corpus. Hence, syntactical size can translate into very different semantical sizes. This status suggests that we should not pay too much attention to syntactical size when we try to determine significant policies. Instead, we favor greater semantical sizes.

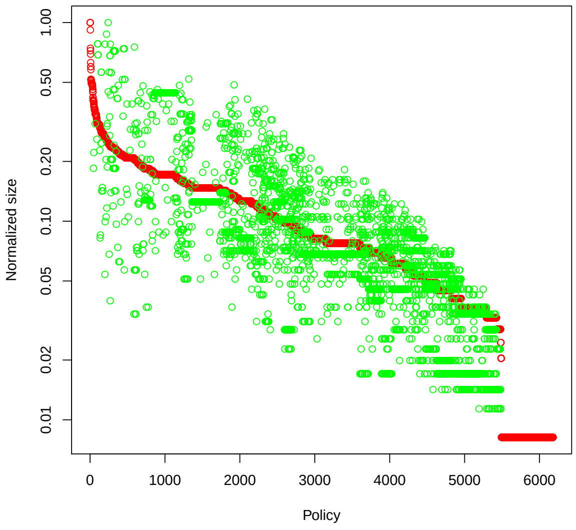


**Figure 3.17.** Syntactical versus semantical size (red and green colors respectively)

## Vocabulary size

Returning to the view that 'policies are models', we may be specifically interested in indicators for different components or services in a web-based systems. To this end, we think of the following correspondence: each *purpose* could correspond to one component of the web-based system; each *recipient* could correspond to one service port of the web-based system; each *category* for data in the policy could correspond to a domain or a concern that the system has to cover. A poorly architected system may of course not directly reveal those components, services, concerns, and domains, but they are modeling entities anyway.

Fig. 3.18 shows the distribution of these numbers for the policies in the corpus. We can fairly combine the different numbers into one metric as follows. Given a policy $P$ with numbers of distinct purposes $\#p(P)$, distinct recipients $\#r(P)$ and distinct categories $\#c(P)$, we compute the vocabulary size of $P$ on a [0,1] scale as follows:

$$VOCA(P) = (\#p(P)/12 + \#r(P)/6 + \#c(P)/17)/3$$

Here, 12, 6 and 17 are the numbers of all purposes, recipients and categories in the vocabulary of P3P. Hence, we measure the degree of coverage of the relevant part of the vocabulary, and we give equal weight to purposes, recipients and categories. Policies with only the NON-IDENTIFIABLE element may have a value of 0.0 for $VOCA(\cdot)$. Otherwise, the minimum value is approximately 0.10—in the case of one purpose, one recipient, and one category.

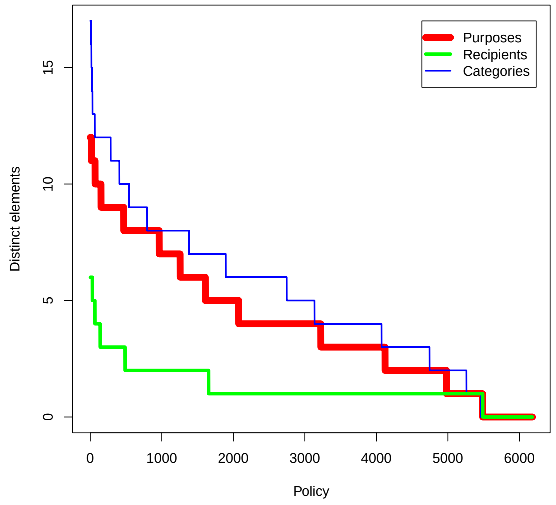


**Figure 3.18.** Distribution of vocabulary counts

For example, all three different representations of the sample policy ‘logging only’ (Fig. 2.6 on p. 23 and Fig. 3.4 on p. 46) have the same value for $VOCA(\cdot)$, namely, it is 0.139: all policies have three distinct purposes and one distinct recipient.

| Min | 1st Q | Median | Mean | 3rd Q | Max |
|---|---|---|---|---|---|
| 0 | 0.137 | 0.19 | 0.232 | 0.347 | 1 |

**Table 3.11.** The $VOCA(\cdot)$ scale

Table 3.11 shows that the [0,1] scale of the $VOCA(\cdot)$ metric is fully exercised by the corpus. When compared to the $SYN(\cdot)$ and $SEM(\cdot)$ metrics, the distribution of $VOCA(\cdot)$ is considerably less shifted to the lower end; compare the ratios of maximum to 3rd quartile for the metrics. Hence, even syntactically or semantically smaller policies exercise the vocabulary substantially.

## Data size

Finally, we introduce the metric *data size*, denoted as $DATA(\cdot)$. With the data size we expect to measure a policy’s footprint in terms of its use of the data schema. In

```
dynamic
  clickstream
    uri
      authority
      stem
      querystring
    timestamp
      ymd.year
      ymd.month
      ymd.day
      hms.hour
      hms.minute
      hms.second
      fractionsecond
      timezone
    clientip
      hostname
      partialhostname
      fullip
      partialip
    other.httpmethod
    other.bytes
    other.statuscode


dynamic
  http
    referer
      authority
      stem
      querystring
    useragent
```

**Figure 3.19.** #dynamic.clickstream and #dynamic.http

this way, we measure degree of exposure quantitatively—while just focusing on data items, abstracting from purposes, recipients, and retention level.

Given a policy $P$, we compute the data size of $P$ as follows. We begin by resolving any given data reference that is listed in a policy to a subtree in the hierarchical definition of the relevant data schema (BDS or CDS). We count all nodes, i.e., leaf and group nodes as well as the root node, for the referenced subtree. We equate $DATA(P)$ with the cumulative count of all distinct nodes that we count for the various data references in a policy. We should note that variable-category data elements escape a fair counting scheme because their hierarchical breakdown is not specified—by definition. Additionally, it is not guaranteed that multiple uses of the same variable-category data reference throughout a policy designate the same type of data. To the best of our knowledge, there is no automatic way of treating variable-category data elements more fairly. The applied counting scheme is useful in so far that it provides a *lower bound* for used data items.

Let us revisit the 'logging only' policy. All three different representations use the same two data references, which do not overlap within P3P base data schema. The referenced subtrees of the BDS are shown in Fig. 3.19. The data size is the direct sum of nodes in these two subtrees: 28.

| Min | 1st Q | Median | Mean | 3rd Q | Max |
|---|---|---|---|---|---|
| 0 | 2 | 30 | 36.57 | 33 | 286 |

**Table 3.12.** The $DATA(\cdot)$ scale

Table 3.12 shows the $DATA(\cdot)$ scale. As a point of reference, we mention that P3P's BDS contains a total of 324 nodes. (Additional data items could be referenced in principle because of the CDS in a policy.) Hence, the $DATA(\cdot)$ scale in Table 3.12 suggests that there are policies with a data-model size close to the size of the complete BDS. The distribution of $DATA(\cdot)$ is shifted though to small node counts, as the quartiles show. (The coverage of P3P's BDS by the corpus is analyzed in §3.4.1.)

## Significant policies

Table 3.13 lists the top-10 policies for all the four metrics in the order of syntactical, semantical, vocabulary, and data sizes. For vocabulary size, we also show the numbers of purposes, recipients, categories with the $(X;Y;Z)$ notation. We only show semantically valid (and then only semantically distinct) policies because it is otherwise difficult to compare them across metrics (with semantical size being included). Policies are highlighted in bold face, if they are among the top 10 in at least two of the four lists. In this manner, we prioritize policies that are 'large' in several respects. It turns out that large data size never meets large semantics or large vocabulary (when focusing on top 10). Also, we recall that we proposed to de-prioritize syntactical size. The following bold policies remain hence:[14]

- 50cent.com
- kerntrophies.com

By our simple rules, we consider these two policies as most significant and we will review them below. Let us also add astrodata.ch, which leads the table for $VOCA(\cdot)$; it exposes the maximum vocabulary size. Given that we consider vocabulary size as a proxy for an interesting model, we view maximum size of this kind to be an indicator of significance, by itself.

## Review of `http://www.astrodata.ch`

The entity of this website is Swiss ASTRODATA AG. The website provides different kinds of personal horoscopes and diagrams: partnership, health, cosmograms, etc. The site also sells books, CDs, and software. The site has a log-in system. The policy

[14] During the lifetime of the effort, some of the policies disappeared from the Internet (see Appendix A.1.1). This is also the case for the bold policies at hand. All policies are preserved in our online corpus at the `http://slps.svn.sourceforge.net/viewvc/slps/topics/privacy/p3p/`

| | Site | *SYN*(·) | *SEM*(·) | *VOCA*(·) | | *DATA*(·) |
|---|---|---|---|---|---|---|
| 1 | taxcollector.com | **95** | 78 | 0.287 | (5; 1; 5) | 42 |
| 2 | allpar.com | **85** | 109 | 0.607 | (11; 2; 10) | 71 |
| 3 | strictlyrockymountain.com | **78** | 91 | 0.553 | (9; 3; 7) | 67 |
| 4 | **raleys.com** | **76** | 242 | 0.667 | (11; 2; 13) | 92 |
| 5 | **thevacuumcenter.com** | **76** | 275 | 0.613 | (9; 2; 13) | 131 |
| 6 | 4tourist.net | **73** | 80 | 0.413 | (6; 1; 10) | 41 |
| 7 | ncl.com | **72** | 30 | 0.610 | (11; 1; 13) | 2 |
| 8 | sigmaaldrich.com | **72** | 29 | 0.420 | (7; 2; 6) | 30 |
| 9 | atlasdirect.net | **69** | 87 | 0.363 | (7; 1; 6) | 111 |
| 10 | fool.co.uk | **69** | 198 | 0.610 | (11; 1; 13) | 137 |

| | Site | *SYN*(·) | *SEM*(·) | *VOCA*(·) | | *DATA*(·) |
|---|---|---|---|---|---|---|
| 1 | keepaustinweird5k.com | 61 | **352** | 0.533 | (9; 2; 9) | 70 |
| 2 | nrc.gov | 63 | **308** | 0.420 | (5; 2; 9) | 72 |
| 3 | **thevacuumcenter.com** | 76 | **275** | 0.613 | (9; 2; 13) | 131 |
| 4 | medcompnet.com | 59 | **274** | 0.397 | (9; 2; 2) | 60 |
| 5 | taylorresearch.com | 65 | **274** | 0.363 | (5; 1; 9) | 131 |
| 6 | countryfinancial.com | 50 | **265** | 0.390 | (8; 1; 6) | 57 |
| 7 | atletix.net | 52 | **260** | 0.250 | (3; 2; 3) | 111 |
| 8 | **50cent.com** | 57 | **253** | 0.843 | (10; 6; 12) | 158 |
| 9 | **raleys.com** | 76 | **242** | 0.667 | (11; 2; 13) | 92 |
| 10 | **kerntrophies.com** | 59 | **240** | 0.767 | (10; 5; 11) | 197 |

| | Site | *SYN*(·) | *SEM*(·) | *VOCA*(·) | | *DATA*(·) |
|---|---|---|---|---|---|---|
| 1 | astrodata.ch | 57 | 42 | **1.000** | **(12; 6; 17)** | 2 |
| 2 | barmans.co.uk | 50 | 20 | **0.930** | **(11; 6; 15)** | 1 |
| 3 | kidzworld.com | 65 | 19 | **0.923** | **(10; 6; 16)** | 1 |
| 4 | foreclosurenet.net | 53 | 182 | **0.880** | **(12; 6; 11)** | 133 |
| 5 | test.quizz.biz | 36 | 18 | **0.867** | **(10; 5; 16)** | 1 |
| 6 | spraci.com | 45 | 19 | **0.863** | **(10; 6; 13)** | 1 |
| 7 | **50cent.com** | 57 | 253 | **0.843** | **(10; 6; 12)** | 158 |
| 8 | **kerntrophies.com** | 59 | 240 | **0.767** | **(10; 5; 11)** | 197 |
| 9 | argos.co.uk | 57 | 162 | **0.743** | **(11; 3; 14)** | 95 |
| 10 | ia.rediff.com | 38 | 18 | **0.717** | **(11; 4; 10)** | 1 |

| | Site | *SYN*(·) | *SEM*(·) | *VOCA*(·) | | *DATA*(·) |
|---|---|---|---|---|---|---|
| 1 | ASPXSoftware.com | 27 | 92 | 0.200 | (2; 2; 2) | **286** |
| 2 | eharmony.com | 36 | 121 | 0.600 | (10; 3; 8) | **270** |
| 3 | cashnetusa.com | 45 | 52 | 0.343 | (5; 2; 5) | **268** |
| 4 | auto-europe.co.uk | 33 | 97 | 0.237 | (4; 2; 1) | **250** |
| 5 | kawasaki.com | 46 | 169 | 0.497 | (8; 4; 3) | **250** |
| 6 | bevmo.com | 55 | 170 | 0.497 | (9; 1; 10) | **244** |
| 7 | petitesophisticate.com | 40 | 170 | 0.380 | (9; 1; 4) | **244** |
| 8 | crutchfield.com | 58 | 197 | 0.703 | (9; 4; 12) | **235** |
| 9 | internest.com | 39 | 155 | 0.400 | (7; 2; 5) | **236** |
| 10 | lampsplus.com | 47 | 156 | 0.447 | (8; 1; 9) | **236** |

**Table 3.13.** Top 10 semantically distinct policies ordered by four different metrics; see the bold column.

is shown in Fig. 3.20.[15] The policy enumerates most purposes; further it uses two variable-category data elements (see §3.3.2) and associates them with all possible

[15] `http://www.astrodata.ch/w3c/policy-general.xml`

```
<POLICY name = "anything-policy"
 discuri = "http://www.example.com/privacy/policy.html">
 ...
 <ENTITY>
  <DATA-GROUP>
   <DATA ref="#business.name">Example Corp.</DATA>
   <DATA ref="#business.contact-info.online.email">privacy@example.com</DATA>
   ...
  </DATA-GROUP>
 </ENTITY>
 ...
 <STATEMENT>
  <PURPOSE>
   <current/><admin/><develop/><tailoring/>
   <pseudo-analysis/><pseudo-decision/><individual-analysis/>
   <individual-decision/><contact/><historical/><telemarketing/>
   <other-purpose>Any other purpose we want</other-purpose>
  </PURPOSE>
  <RECIPIENT>
   <ours/><delivery/><same/><other-recipient/><unrelated/><public/>
  </RECIPIENT>
  <RETENTION><indefinitely/></RETENTION>
  <DATA-GROUP>
   <DATA ref="#dynamic.miscdata">
    <CATEGORIES>
     <physical/><online/><uniqueid/><purchase/><financial/>
     <computer/><navigation/><interactive/><demographic/>
     <content/><state/><political/><health/><preference/>
     <location/><government/>
     <other-category>Any other type of data</other-category>
    </CATEGORIES>
   </DATA>
   <DATA ref="#dynamic.cookies">
    <CATEGORIES>
     <physical/><online/><uniqueid/><purchase/><financial/>
     <computer/><navigation/><interactive/><demographic/>
     <content/><state/><political/><health/><preference/>
     <location/><government/>
     <other-category>Any other type of data</other-category>
    </CATEGORIES>
   </DATA>
  </DATA-GROUP>
 </STATEMENT>
</POLICY>
```

**Figure 3.20.** The privacy policy of astrodata.com

categories. Hence, the policy declares substantial exposure (in terms of privacy). In some sense, the policy declares universal exposure, but this claim cannot easily be formalized due to the difficult nature of variable-category data elements. Please observe the name of the policy: "anything policy".[16]

Two deficiencies of the policy are worth reporting though. The entity description uses 'sample data' rather than genuine data about ASTRODATA AG; see Fig. 3.20. Further, the policy fails in making good use of the BDS in places where it obviously could. (That is, a horoscope service is likely to require a birthday date, which is

[16] It was suggested by a reviewer during the review phase of this effort to use a new domain term to cover this sort of policy: 'NoRightsReserved' in legal language.

```
<POLICY name="WebsitePolicy"
 discuri="http://privacypolicy.umusic.com" opturi="...">
  ...
  <ENTITY>
    <DATA-GROUP>
      <DATA ref="#business.name">Universal Music Group</DATA>
      ...
    </DATA-GROUP>
  </ENTITY>
  ...
  <STATEMENT>
    <PURPOSE>
      <current/>
      <admin/>
      <develop/>
      <tailoring/>
      <pseudo-analysis/>
      <pseudo-decision/>
      <individual-analysis required="opt-in"/>
      <individual-decision required="opt-in"/>
      <contact required="opt-in"/>
      <historical/>
    </PURPOSE>
    <RECIPIENT>
      <ours/>
      <delivery/>
      <same required="opt-in"/>
      <other-recipient required="opt-in"/>
      <unrelated required="opt-in"/>
      <public required="opt-in"/>
    </RECIPIENT>
    <RETENTION><indefinitely/></RETENTION>
    <DATA-GROUP>
      <DATA ref="#user.name"/>
      <DATA ref="#user.bdate"/>
      <DATA ref="#user.gender" optional="yes"/>
      <DATA ref="#user.home-info"/>
      <DATA ref="#thirdparty.name" optional="yes"/>
      <DATA ref="#thirdparty.bdate" optional="yes"/>
      <DATA ref="#thirdparty.gender" optional="yes"/>
      <DATA ref="#thirdparty.home-info" optional="yes"/>
      <DATA ref="#dynamic.clickstream"/>
      <DATA ref="#dynamic.http"/>
      <DATA ref="#dynamic.clientevents"/>
      <DATA ref="#dynamic.searchtext"/>
      <DATA ref="#dynamic.interactionrecord"/>
      <DATA ref="#dynamic.cookies">
        <CATEGORIES>
          <physical/><online/><uniqueid/><purchase/><financial/>
          <computer/><navigation/><interactive/><demographic/>
          <content/><state/><preference/>
        </CATEGORIES>
    </DATA-GROUP>
  </STATEMENT>
</POLICY>
```

**Figure 3.21.** The privacy policy from the site 50cent.com

```
<POLICY name="WebsitePolicy"
  discuri="http://www.pdu-wc.com/policy.html" opturi="...">
  ...
  <ENTITY>
    <DATA-GROUP>
     <DATA ref="#business.name">Steve Schreiner</DATA>
     <DATA ref="#business.contact-info.online.email">...@trophytoolbox.com</DATA>
      ...
    </DATA-GROUP>
  </ENTITY>
  ...
  <STATEMENT>
    <PURPOSE>
      <current/><admin/><develop/><pseudo-analysis/>
      <pseudo-decision/><individual-analysis/><individual-decision/>
      <contact required="opt-out"/><historical/>
      <telemarketing required="opt-out"/>
    </PURPOSE>
    <RECIPIENT>
      <ours>
        <recipient-description>
          Information is only shared with our partners
          who share our privacy policy.
        </recipient-description>
      </ours>
      <delivery/><same/><other-recipient/><unrelated/>
    </RECIPIENT>
    <RETENTION><indefinitely/></RETENTION>
    <DATA-GROUP>
      <DATA ref="#user.name"/>
      <DATA ref="#user.bdate" optional="yes"/>
      <DATA ref="#user.gender" optional="yes"/>
      <DATA ref="#user.home-info"/>
      <DATA ref="#user.business-info" optional="yes"/>
      <DATA ref="#thirdparty.name"/>
      <DATA ref="#thirdparty.home-info"/>
      <DATA ref="#dynamic.clickstream"/>
      <DATA ref="#dynamic.http"/>
      <DATA ref="#dynamic.clientevents"/>
      <DATA ref="#dynamic.searchtext"/>
      <DATA ref="#dynamic.interactionrecord"/>
      <DATA ref="#dynamic.cookies">
        <CATEGORIES>
          <physical/><online/><uniqueid/><purchase/>
          <computer/><navigation/><interactive/>
          <content/><state/><preference/><location/>
        </CATEGORIES>
      </DATA>
      <DATA ref="#dynamic.miscdata">
        <CATEGORIES>
          <physical/><online/><uniqueid/><purchase/>
          <computer/><navigation/><interactive/>
          <content/><state/><preference/><location/>
        </CATEGORIES>
      </DATA>
    </DATA-GROUP>
  </STATEMENT>
</POLICY>
```

**Figure 3.22.** The privacy policy from the site kerntrophies.com

available through BDS; variable-category data elements should not be used in such a case.)

#### Review of `http://www.50cent.com`

The website is dedicated to the American rapper and actor Curtis James Jackson III. The site provides information related to the artist: news, videos, music, photos, and events. The site has a log-in system and a message board (forum). The log-in system allows users to sign-in using an account from other social networks (Facebook, Myspace, Twitter, Windows LiveID, etc). The policy is shown in Fig. 3.21.

The entity description does not directly refer to Jackson or 50cent.com, but it identifies the *Universal Music Group*[17] as the primary entity.

Most purposes are listed, but the less private ones (such as ‘contact’) admit opt-in on the side of the user. Also, several recipients are listed, but again, the less private ones (such as ‘public’) admit opt-in. Detailed data references from three of the four BDS branches are provided. The use of cookies is specified in detail with regard to the categories that apply. Hence, the policy is really significant in terms of the number of included purposes and recipients, subject to opt-in, in several cases, and in terms of the number of data references.

#### Review of `http://kerntrophies.com/`

The website is an online shop for trophies, plaques, medals, and awards. The site allows customizing of the item’s design (including engraving). The site has a log-in system. The policy is shown in Fig. 3.22. The policy is close to the one shown in Fig. 3.21: they declare the same recipients, they cover about the same purposes; see ‘tailoring’ vs. ‘telemarketing’. The site 50cent.com provides more privacy because it declares ‘opt in’ for several purposes and recipients, whereas the site kerntrophies.com uses ‘opt out’ for a few purposes. The sites are likewise close in terms of amounts of collected data, but kerntrophies.com uses the variable-category data elements *#dynamic.misc*.

It is striking to observe that nearly the same list of purposes is given in the same order in Fig. 3.21 and Fig. 3.22. Common sense may suggest that this cannot be an accident, and one policy has been derived from the other. More likely, both policies have been created with a P3P editor. That is, the shown order of purposes agrees with the order in the P3P specification. Also, only three purposes were omitted—one of them being the special purpose ‘other-purpose’. All the omitted purposes happen to be the least popular ones in the corpus; see §3.4.1.

### 3.4.4 Analysis of Cloning

The underlying research question is “What are common policies?”; see §3.2.1. Clone detection is effective in identifying common policies. We will then determine the

[17] `http://www.universalmusic.com/`

characteristics of the most common policies and the relationships between them. An appreciated side effect is that we discover some authoring and publishing habits of website entities.

If we compare clone detection for P3P with regular programming languages, then we can emphasize that trying to find 'common' Java or C++ programs would be of limited use. Trying to find common P3P policies makes sense for the following reasons. First, one can expect that policy authors may sometimes reuse entire policies from other websites. Second, P3P tools provide templates which may also stipulate the use of common policies. Third, the simplicity of the P3P language and 'typical' policies is such that we may assume to encounter common policies just by accident.

## Types of clones

Since we are interested in common policies, our focus is on *clone detection across different files* [103] (i.e., across different policy files). Our research question is not concerned with cloned policy text within the same policy, but this may be an orthogonal topic for future work.

Let us consider the classification for *types of clones* [116]: "Type 1 is an exact copy without modifications (except for whitespace and comments). Type 2 is a syntactically identical copy; only variable, type, or function identifiers have been changed. Type 3 is a copy with further modifications; statements have been changed, added, or removed." In some cases, type 4 is considered as well [159]: "Two or more code fragments that perform the same computation but are implemented by different syntactic variants." These types are of limited use when classifying inter-file P3P clones as we discuss now.

In our study, we are interested in textual, syntactical, and semantical clones. We use the term *textual clones* (abbreviated as 'txt') for policy files that are identical when comparing the text-encoded XML content of the files. Arguably, textual clones are type-1 clones. In order to find textual clones, we compare XML files for their precise textual content. If we find textual clones, then we have not just found common policies, but we also face an interesting form of copy and paste behavior. Here we note again that policies are supposed to identify the entity of the website. Hence, type-1 clones are policies from different websites with the same alleged entity. Our analysis will investigate this scenario in more detail by also taking into account domains of websites and entity URIs.

We use the term *syntactical clones* (abbreviated as 'syn') for policies that are identical in terms of the abstract syntax of §3.3.2. Hence, all formatting differences are neutralized, the entity name is eliminated, other text parts are eliminated, order of elements in collections become invariant. Arguably, syntactical clones are type-2 or type-3 clones depending on the mapping of programming language-centric terms in the above-mentioned definitions to P3P terms. The clone detection at hand is AST-based [26]. If we find syntactical clones, then we have indeed found common policies that are obviously used across different websites. Syntactical clones could be the result of i) using a template of a P3P authoring tool; ii) copy-paste-and-customize; iii) accident—especially in the case of simple policies.

| Type | # Clone groups | # Added clone groups | # Added cloned policies | # Enlarged clone groups | # Merged clone groups | % Clones |
|---|---|---|---|---|---|---|
| txt | 296 | 296 | 1,941 | – | – | 31.40 |
| syn | 496 | 280 | 1,857 | 46 | 114 | 30.04 |
| sem | 351 | 20 | 69 | 0 | 32 | 1.12 |
| **Total** | – | – | 3,867 | – | – | 62.55 |

**Table 3.14.** Numbers of clones and clone groups

We use the term *semantical clones* (abbreviated as ‘sem’) for policies that are identical in terms of the normal form of §3.3.3. The obvious value of the normal form is that we abstract from the representational diversity for policies. If we find semantical clones, then there is evidence that authors exercise the representational diversity. Arguably, semantical clones are type-4 clones—if we accept the relational semantics of our normal form as a proper semantics. Here, we note that type-4 clones are very hard to detect for common programming languages. The relational semantics of P3P is straightforward though. Our normalization approach can be compared with the cloning classification of [18] where refactorings are used to detect and possibly remove clones.

## A note on implementation

Overall, it is simple to accommodate clone detection for the chosen clone types because the P3P syntax is trivial, no parameterized clones are considered, policies are trivially small (compared to programs in mainstream languages), all reported and conceivable P3P corpora are small (compared to, for example, the numbers of programs in mainstream languages that are obtainable from open-source repositories). Hence, implementation of clone detection for textual, syntactical, and semantical clones is relatively straightforward, and no scalability challenge has to be tackled. It is hard to detect semantical clones for programming languages, but it is straightforward for P3P's finite domain-based semantics.

## Results of the analysis

Consider Table 3.14 with numbers of clones and clone groups. Column *#Clone groups* shows the number of detected clone groups of each type. Column *#Added cloned policies* shows the number of newly detected cloned policies for each type, when compared to cloned policies for the prior type. For clarity, we count the representative of a clone group as being a cloned policy. Hence, *% Clones* describes the percentages of all policies of the corpus that are not distinct in the cloning sense. Column *#Added clone groups* shows the number of new clone groups for each type, where we do not count groups that are only enlarged or merged. Column *#Enlarged clone groups* shows the number of clone groups that were obtained solely by adding newly detected cloned policies of the given type to a clone group of the prior type. Column *#Merged clone groups* shows the number of clone groups from the prior type that were merged in clone groups of the given type.

It should be noted that the number of clone groups drops from syntactical to semantical type: this is because some of the syntactically cloned policies are semantically invalid. Namely, 143 syntactical clone groups contain semantically invalid policies (that is, 28.83 % of all syntactical clone groups). The rest of the difference is because of groups being merged under the clone type.

We also compared all policies with the 6 templates of IBM's policy editor because we realized at some point that many policies were obviously edited with this prominent P3P tool. We found that all but 1 of the templates have an associated clone group. We encounter and discuss some of these clone groups when we inspect selected clone groups below.

Textual and syntactical clone detection clearly identify a substantial amount of clones. The contribution of semantic clone detection is small. In Appendix A.1.2, we consider the remaining diversity among the semantically distinct policies on the grounds of the partial order for degree of exposure as of §3.3.4.

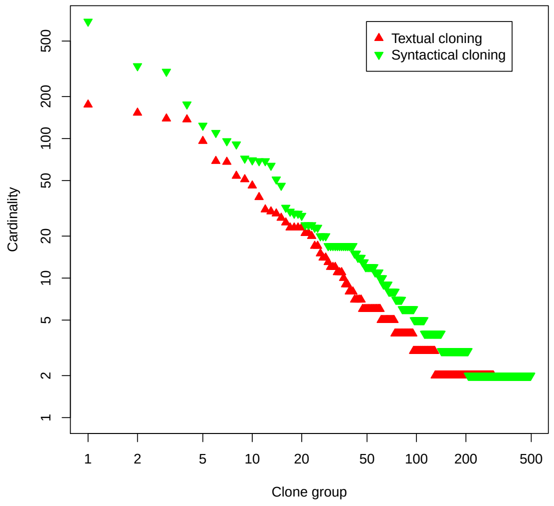


**Figure 3.23.** Distribution of clone group cardinalities

The distribution of textual and syntactical clone groups in terms of cardinality is shown in Fig. 3.23 and Table 3.15. There are a few huge clone groups; most clone

| Type | Min | 1st Q | Median | Mean | 3rd Q | Max |
|---|---|---|---|---|---|---|
| txt | 2 | 2 | 2 | 7.609 | 4 | 175 |
| syn | 2 | 2 | 2 | 8.819 | 4 | 690 |

**Table 3.15.** Cardinalities of textual and syntactical clone groups

groups are of trivial size. More precisely, there are 36 textual clone groups with cardinality $> 10$. These groups make up 12.16 % of all textual clone groups. There are 59 syntactical clone groups with cardinality $> 10$. These groups make up 11.90 % of all syntactical clone groups. Semantical clone detection is left out here because it adds very little.

| Type | Min | 1st Q | Median | Mean | 3rd Q | Max |
|---|---|---|---|---|---|---|
| txt | 2 | 9 | 16 | 22.19 | 37 | 147 |
| syn | 2 | 9 | 19 | 23.23 | 36 | 245 |
| sem | 2 | 9 | 17 | 18.78 | 28 | 99 |

**Table 3.16.** Distribution of $SYN(\cdot)$ over clones

| Type | Min | 1st Q | Median | Mean | 3rd Q | Max |
|---|---|---|---|---|---|---|
| txt | 0 | 0 | 13 | 28.35 | 29 | 253 |
| syn | 0 | 0 | 16 | 25.78 | 29 | 275 |
| sem | 0 | 6 | 20 | 29.76 | 34.25 | 275 |

**Table 3.17.** Distribution of $SEM(\cdot)$ over clones

Tables 3.16 and 3.17 summarize the distribution of syntactical and semantical sizes over clones for all types. (We consider only distinct sizes for clones at a given type.) Consider Table 3.16. The maximum for $SYN(\cdot)$ first increases from txt to syn, which means that syntactical clone detection finds clones with syntactical size near the maximum in the corpus. The maximum for $SYN(\cdot)$ subsequently drops from syn to sem, which is implied by the substantial amount of semantically invalid policies, which escape semantical clone detection. In fact, we lose the (syntactically) greater policies in this manner. Median and third quartile co-drop. Now consider Table 3.17. The maximum for $SEM(\cdot)$ first increases slightly from txt to syn, but then it remains at the same value for type sem, which means that semantical clone detection does not make any contribution in terms of finding clones of greater size than those found by textual or syntactical clone detection.

### Discussion of textual cloning

We would like to understand the nature of textual cloning. Therefore we examine the top-10 clone groups (in terms of cardinality); see Table 3.18. We list the issuing entity, cardinality, values for the metrics of §3.4.3 as well as the 'average URI distance', which we explain in a second. A semantical size of 0 indicates representatives of the 'full privacy' from the introduction. (This policy has syntactical size 2 because it consists of a statement node and a non-identifiable node underneath.) It turns out that there is one semantically invalid clone group among the top 10: No. 5 uses a reference from BDS v.1.0. It also happens to have the largest syntactical size along with the group No. 1.

| | **Entity** | **Card.** | **Avg. dist.** | $SYN(\cdot)$ | $SEM(\cdot)$ | $\boldsymbol{VOCA}(\cdot)$ | $\boldsymbol{DATA}(\cdot)$ |
|---|---|---|---|---|---|---|---|
| 1 | John Hensley | 175 | 1.000 | 42 | 156 | 0.650 (9; 3; 12) | 188 |
| 2 | CybrHost | 139 | 1.000 | 2 | 0 | 0.000 (0; 0; 0) | 0 |
| 3 | PhotoBiz | 137 | 1.000 | 2 | 0 | 0.000 (0; 0; 0) | 0 |
| 4 | Boatventures | 96 | 1.000 | 15 | 16 | 0.163 (2; 1; 3) | 3 |
| 5 | Real Estate | 69 | 1.000 | 42 | – | 0.363 (7; 1; 6) | – |
| 6 | Hilton Hotels | 68 | 0.779 | 9 | 6 | 0.123 (2; 1; 1) | 1 |
| 7 | NASA | 54 | 0.000 | 13 | 29 | 0.180 (4; 1; 1) | 30 |
| 8 | Rezidor SAS | 51 | 0.000 | 12 | 16 | 0.163 (2; 1; 3) | 29 |
| 9 | Bravenet | 46 | 0.000 | 17 | 16 | 0.123 (2; 1; 1) | 29 |
| 10 | Wetpaint | 38 | 0.237 | 19 | 24 | 0.180 (4; 1; 1) | 30 |
| | **Average** | – | 0.602 | 17.30 | 26.30 | – | – |

**Table 3.18.** Top 10 textual clone groups

### Intra- versus inter-domain cloning

We would like to understand systematically whether cloning happens within a given domain (say, an organization) or across domains. These different scenarios correspond to different kinds of potential copy-and-paste-like authoring behaviors. We introduce the measure of *URI distance* for a policy; this is the 'distance' between site URI and the entity URI (the so-called discuri). We set this distance to 0 if both URIs appear to be about the same entity—subject to advanced matching of domain names (such as in the case of `www.microsoft.com` and `server1.microsoft.de`), and to 1 otherwise. Given a clone group, we can also determine its *average URI distance*; this is simply the average of the URL distances for the members of the clone group; this measure is on a [0,1] scale. Please note that the entity URI would be the same for all members of a textual clone group; only the site URIs could differ.

Let us illustrate the subtle point of URI distance with a particularly clear example not included in the top 10 list. The site `p3pedit.com` presents a software product, P3PEdit, of the company `codeinfusion.com`. Incidentally, P3PEdit is a P3P tool. The site of the product has a P3P policy embedded into the reference file `http://p3pedit.com/w3c/p3p.xml`, but the "discuri" attribute points to the site of the company. This is not against the P3P specification, but demonstrates cross-domain copy and paste.

We have inspected the top 10 clone groups and member websites, and characterize them as follows. No. 1 textual clone group is similar to the policies shown in Fig. 3.22 and Fig. 3.21: non-trivial composition of purposes, recipients, and data. According to the website mentioned in the policy, the site offers real estate software, technology services, and websites for agents, brokers, and realtor associations, which probably explains such high cardinality of the group. It also happens to have the largest $\mathit{SYN}(\cdot)$ along with the group No. 5.

No. 2 is the '*full privacy*' policy of a web-site hosting service. An average URI distance of 1 indicates that *all* clones appear on domains apparently unrelated to the hosting service. We hypothesize that many hosted web sites reuse (perhaps implicitly) a default policy without customizing it for their own domains. Several of the other top 10 are about web hosting, too—with similar URI distances. In contrast, No. 7, i.e., the NASA policy, is never cloned outside NASA's web space.

To summarize, it is common practice to copy and paste policies from elsewhere, as is, without even customizing website-specific information. Copy and paste within the same domain or across different domains owned by the same entity makes sense, of course. Only 3 from the top 10 clone groups are clear-cut cases of websites on domains that can be associated with a single entity.

## Discussion of syntactical cloning

We determine the top 10 syntactical clone groups as those with the greatest cardinality after removing all textual clones; see Table 3.19. Thereby we compensate for the effect that several syntactical clone groups are extensions of textual clone groups. It turns out that there are two semantically invalid clone groups among the top 10: No. 6 has conflicting optionality attributes for one data reference, No. 9 uses a reference from BDS v.1.0.

No. 1 describes data collection of these forms: access logs, search strings, cookies, and user-related information (name and business email address). This appears to be an important e-commerce scenario that does not involve online purchasing. It also has the biggest $\mathit{SEM}(\cdot)$ amongst these top 10 groups. Hence, our cloning study may have detected a domain concept that does not have an explicit domain term. However, it is possible that the policy derives from a template that we are not aware of. It is also possible that heavy cross-website reuse has happened.

No. 2 shows us that there are even more instances of the '*full privacy*' policy in the corpus. If we combine textual and syntactical cloning, then there are 690 occurrences of this policy in the corpus. This is 11.16 % of *all* policies. Incidentally, the policy is not in the suite of online samples of the primary textbook on P3P [42].[18]

[18] `http://p3pbook.com/examples.html`

| | **Sample entity** | **Card.** | **Avg. dist.** | *SYN*$(\cdot)$ | *SEM*$(\cdot)$ | ***VOCA***$(\cdot)$ | ***DATA***$(\cdot)$ |
|---|---|---|---|---|---|---|---|
| 1 | Accountancy | 331 | 0.000 | 36 | 44 | 0.367 (8; 2; 2) | 33 |
| 2 | CybrHost | 284 | 0.109 | 2 | 0 | 0.000 (0; 0; 0) | 0 |
| 3 | IBM tracking | 218 | 0.124 | 19 | 24 | 0.180 (4; 1; 1) | 30 |
| 4 | Johnston Press | 116 | 0.586 | 20 | 24 | 0.220 (4; 1; 3) | 30 |
| 5 | IBM logging | 102 | 0.098 | 9 | 12 | 0.137 (3; 1; 0) | 28 |
| 6 | IBM purchase | 67 | 0.104 | 51 | – | 0.363 (5; 2; 6) | – |
| 7 | Beach Suites | 64 | 0.125 | 31 | 29 | 0.353 (6; 2; 4) | 30 |
| 8 | 1066 Pools Ltd | 30 | 0.000 | 24 | 10 | 0.377 (6; 1; 8) | 1 |
| 9 | Art of War | 24 | 0.083 | 12 | 17 | 0.117 (1; 1; 2) | 30 |
| 10 | WebSolutions | 20 | 0.000 | 31 | – | 0.283 (7; 1; 2) | – |
| | **Average** | – | 0.123 | 23.50 | 16.00 | – | – |

**Table 3.19.** Top 10 syntactical clone groups

IBM's P3P Policy Editor[19] provides 6 templates for P3P authors. The idea is that these policies may cover some typical cases, and they could be customized if needed. We use these templates as markers in our clone detection process. We found that all but 1 of the templates have an associated clone group. Some of them occur in the discussed table, and hence we discuss them here.

No. 3 is described by IBM's tool as following: "This template is designed for sites which track users on a non-identifiable basis through use of cookies. It also assumes standard server access logs will be collected." No. 3 is similar to No. 1. More precisely, we have that No. 3 $\leq_{sem}$ No. 1 (see §3.3.4) because No. 1 collects additional user data and cookies are admitted in a more general way.

No. 4 is a minor variation on No. 3; both policies differ in terms of categories for cookies. Neither policy is 'less or equal' (in terms of '$\leq_{sem}$') than the other policy.

No. 5 is described by IBM's tool as follows: "A template privacy policy for sites which only collect standard server access logs for site operation purposes." No. 5 is similar to No. 3. More precisely, we have that No. 5 $\leq_{sem}$ No. 3 because No. 5 describes only data collection for access logs and declares even less data to be collected. In §2.3.1, we coined the name 'logging only' for this policy.

No. 6 is described by IBM's tool as following: "This template captures the data typical sites collect as part of an on-line shopping experience." This group has the biggest *SYN*$(\cdot)$ among top 10. We count this policy as semantically invalid because of conflicting optionality attributes for different occurrences of #dynamic.miscdata; see §3.4.2. No. 6 adds facts about purchase-related data based on #dynamic.miscdata on top of No. 5. (We cannot apply '$\leq_{sem}$' though, since No. 6 has no normal form.)

To summarize, there is an extra of 30.04 % of the corpus that is cloned in syntactical but not textual sense. We assume that this percentages corresponds to common needs of websites combined with the relative simplicity of P3P and the relatively

[19] `http://www.alphaworks.ibm.com/tech/p3peditor`

small size of policies as well as the availability of reusable templates. This kind of cloning gives hope that a relatively small number of standardized policies could be potentially sufficient. In the top 10 of Table 3.19, 3 templates of IBM's Policy Editor appear.

### 3.4.5 Analysis of Extensions

The underlying research question is "What language extensions circulate?"; see §3.2.1. The main goal is to analyze the use of P3P's extension mechanism and to infer candidate language extensions to be considered by future languages for privacy policies.

```
<RETENTION>
  <legal-requirement/>
  <EXTENSION optional="no">
    <use-duration>
      <one-year/>
    </use-duration>
    <retention-basis>
      <internal-company-regulation> [Text] </internal-company-regulation>
    </retention-basis>
  </EXTENSION>
</RETENTION>
```

**Figure 3.24.** Example of EXTENSION usage

#### The essence of extensions

P3P's extension mechanism allows P3P tool providers and policy authors to add and use extra constructs. For instance, one might want to add a more detailed specification of retention, say in terms of a specific duration for storage of data; see Fig. 3.24 for such an example.

Technically, P3P admits extension elements immediately below P3P elements for statements, purposes, recipient, retention levels (as in the example), and data references (in fact, data groups). Extensions are effectively named—through designated XML tags; see `legal-requirement` in the example. Also, the further structure of each extension element is an XML tree. All XML elements can be qualified by policy-specific XML namespaces.

Procedurally, the use of the extension mechanisms is challenging since policy tools and website users may need (want) to understand the extensions, and there is no obvious technical solution to that problem. P3P admits this challenge with its classification for optional versus mandatory extensions: "A mandatory extension to the P3P syntax means that applications that do not understand this extension cannot understand the meaning of the whole policy (or policy reference file, or data schema)

| | |
|---|---|
| delivery | 1364 |
| cert | 440 |
| complaint | 414 |
| content | 400 |
| statement | 346 |
| age | 232 |
| proxy-agreement | 138 |
| payback | 88 |
| statement-group | 3 |

**Figure 3.25.** Frequency of top-level primitive extensions (w/o group-info)

| | |
|---|---|
| delivery | 341 |
| ppurpose | 253 |
| collection-method | 244 |
| destruction-method | 244 |
| retention-basis | 244 |
| use-duration | 243 |
| recipient-description | 236 |
| recipient-name | 236 |
| recipient-duration | 62 |
| same | 62 |
| data-group | 11 |
| jurisdiction | 1 |
| purpose | 1 |
| retension-duration | 1 |

**Figure 3.26.** Frequency of top-level structured extensions

containing it." [20] This classification focuses on the tool view. However, even an optional extension would need to be specified for the benefit of website users, and the specification may need to be rigorous enough to be used in the context of policy enforcement. We assume that such requirements may partially discourage the use of P3P's extension mechanism. This is a challenge for future approaches to language-based privacy.

## Frequency analysis

Let us analyse usage of the extension mechanism. We begin by analyzing the frequency of top-level tags found in the corpus. Fig. 3.25 lists all *primitive extensions* by which we mean extensions of the form `<tag/>` (possibly including attributes but no child elements). Fig. 3.26 lists all *structured extensions* by which we mean extensions of the form `<tag>...</tag>`. (In fact, we omit one dominant tag, as

[20] `http://www.w3.org/TR/P3P11/#extension`

we discuss below.) Table 3.20 summarizes some global numbers on the usage of the extension mechanism.

### Discussion

The P3P standard illustrates a few proposals for extensions. It turns out that one of these proposals, 'group-info', accounts for most extension usage in the corpus. The group-info tag lives in an XML namespace by IBM, and it provides a comment facility. There are only 162 policies (of them 151 syntactically distinct) that use extensions other than group-info. We suggest that further scrutiny should focus on those 162 policies.

| | |
|---|---|
| # Extensions | 7631 |
| # Distinct extension tags | 23 |
| # Policies with extensions | 3076 (of all syntactically valid 6,182 policies) |
| # Syntactically distinct policies with extensions | 1199 (of all syntactically distinct 2,304 policies) |

**Table 3.20.** Basic numbers about extension usage

### Grammar inference

Fig. 3.27 shows an inferred grammar for all the extensions in the corpus. We use EBNF-like notation (as opposed XML-centric XSD notation) for conciseness' sake. The extensions are grouped by the rooting element (statement, purpose, recipient, retention, and data-group); see the first rule in each block in the figure. Grammar rules for top-level tags of extensions are collected as iterated choices over extra extension elements, which are defined by extra productions in each block. Many extensions use the standard XML namespace of P3P—as illustrated in Fig. 3.27 with the appropriate font. (This is a debatable practice because the extending entity does not have control over the namespace.)

### Discussion

We observe that the complexity of the inferred grammar suggests that *there are more proposed extensions than baseline language elements*. That is, there are 87 different XML tags introduced by extension elements in the corpus. In contrast, there are only 50 different XML tags for P3P statements according to its XSD-based syntax definition. Hence, we obtain a ratio of extension : baseline of 1.74.

Several of the proposed extensions extend P3P's privacy vocabulary in an obvious manner, e.g.:

- How long exactly is information stored?

*statement* : (***collection-method*** | ***destruction-method*** | group-info | statement-group)*

*collection-method* = (***other-method*** | **delivery** | **document** | **purnish** | **qnaboard** | **subscription** | ...)*  
*destruction-method* = (***other-method*** | **format** | **shatter**)*  
*other-method* ≐ ε

*purpose* : (***ppurpose*** | ***purpose*** | **age** | **cert** | **complaint** | **content** | **payback** | **proxy-agreement** | **statement**)*

*ppurpose* ≐ (**ACCOUNT** | **BROWSING** | **delivery** | **FEEDBACK** | **finmgt** | **government** | **login** | **marketing** | ...)*  
*purpose* = (**FEEDBACK** | **government**)*

*recipient* : (***delivery*** | ***jurisdiction*** | ***recipient-description*** | ***recipient-duration*** | ***recipient-name*** | ***same*** | **delivery**)*

*delivery* ≐ ε  
*jurisdiction* = (***long-description***)*  
*long-description* ≐ ε  
*other-duration* ≐ ε  
*other-purpose* ≐ ε  
*recipient-description* = (***other-purpose*** | **admin** | **age** | **agency** | **cert** | **complaint** | **content** | **delivery** | ...)*  
*recipient-duration* = (***other-duration*** | **five-year** | **instance** | **one-year** | **six-month**)*  
*recipient-name* ≐ ε  
*same* ≐ ε

*retention* : (***retension-duration*** | ***retention-basis*** | ***use-duration***)*

*credit-privacy-law* = (**one-year**)*  
*e-trade-law* = (**five-year**)*  
*internal-company-regulation* ≐ ε  
*other-basis* ≐ ε  
*other-duration* ≐ ε  
*retension-duration* = (***other-duration***)*  
*retention-basis* ≐ (***credit-privacy-law*** | ***e-trade-law*** | ***internal-company-regulation*** | ...)*  
*use-duration* = (***other-duration*** | **five-year** | **instance** | **one-month** | **one-year** | ...)*

*data-group* : (data-group)*

*categories* = (**COMPUTER** | **UNIQUEID**)*  
*category* = (**COMPUTER** | demographic | navigation | purchase | state | **UNIQUEID**)*  
*data-group* = (datatype)*  
*datatype* = (dynamic | user)*  
*dynamic* = (miscdata)*  
*home-info* = (telecom)*  
*miscdata* = (category)*  
*telecom* = (**TELEPHONE**)*  
*user* = (home-info)*

(Some productions had to be cut off for scalability of presentation.) Underlined tags correspond to elements without content. For brevity, attributes of extensions are not shown in the grammar. When '≐' is used as a rule separator, then text content instead of structured content was encountered for the element in question. All tags that use consistently P3P's XML namespace are shown in boldface. All tags that are used with different XML namespaces are shown in capitals and boldface. All the other tags use a unique XML namespace (different from P3P's).

**Figure 3.27.** The (truncated) extension grammar for the corpus

- How is information destroyed after the retention period expired?
- How to describe recipients in detail?

Appendix A.1.2 provides additional information about the diversity of extensions with regard to the domains of the policies. Thereby, we have found that most extensions can be traced to Korea. We can only hypothesize about this finding. One possible reason could be a particular, national legislation. We mention this finding again in the threats to validity discussion in §3.5.

## 3.5 Threats to Validity

Let us discuss several limitations of the trustworthiness of the results of our study. We discuss limitations by their possible origin.

### The Corpus

There are several possible approaches to collecting the initial set of websites to be checked for P3P policies. One way is to aim for most visited or most mentioned websites and to collect them from sources such as top-lists and rankings [57, 153]. Another option is to obtain "typical" search terms (via logs of search engines) and to collect websites showing up in the results of searching with such terms [44]. Yet another option is to collect a random sample of websites by Web crawling, starting with some seed (a list of URLs) and then visiting in a recursive manner all hyperlinks occurring on retrieved pages. In all cases the idea is to get a somehow characteristic set of websites: most visited, typical, or random—later this allows to generalize results for that group of websites.

We wanted to collect as many P3P policies as possible, from a single trustworthy source. The Open Directory Project positions itself as "the largest, most comprehensive human-edited directory of the Web."[21] We consider this effort to be sufficient for filtering out abandoned or insignificant websites and also for providing diversity. As language engineers, we do not care if an observed language phenomenon arises from websites that might be not frequently visited or do not show up in top searches. We are curious to investigate any phenomenon if it appears through a reasonable resource like ODP.

Due to a lately discovered omission, we did not retrieve websites and therefore policies from ODP's category 'Kids and Teens'.

### The Tools

We use homegrown tools to run all analyses in the study. While some of the actions are quite simple (e.g., calculation of a metric based on a provided formula), others need an empirical evaluation. Clone detection is probably the most important example. To address this problem, we manually checked 2 % of cloned policies of each

[21] `http://www.dmoz.org/docs/en/about.html`

type. In that, we followed the procedure of empirical evaluation described by Falke et al. [61]: a human oracle validates a number of clones selected randomly in an automated fashion. We did not find any false positives in our check.

### Reliability of the Results

Aside the aforementioned technical aspects that might threaten the validity of our results, the major threats are in the decisions that we had to make during the process of taming the P3P language. While we argue in favor of our decisions when introducing them, we would like to mention some of them here, without repeating the pro-reasons.

**Normal form**. According to the assumed normal form, 21.24 % of our corpus is semantically invalid. With a different normal form, we may be able to reduce the number of semantically invalid policies, and thereby exclude less policies from some of the analyses. We assume though that we picked the most established normal form, as we discussed in §3.3.3.

**Variable-category data elements**. In all our analyses, we treat data references with variable categories the same way we treat data references with fixed categories. That is, we count them as single references to specific data items. This may be suboptimal, as pointed out, for example, in §3.4.3. Hence, the reliability of our results for policies with variable-category data elements is threatened.

**Privacy order**. We aligned elements of P3P language by their meaning that we discovered from the specification of the language. While we did our best during this judgement, it might be that we failed to understand the nuances of definitions. For instance, in a recent work by Ghazinour and Barker, they apply lattice-based structure to P3P elements [70], and, e.g., the resulting chain for retention differs from our. Ghazinour and Barker start with "legal-requirement" as the bottom element, then place "no-retention", "stated-purpose", "business-practices", and "indefinitely" as the top element. The impact of ill-conceived partial order is very limited because we leverage the specific definition only informally when we discuss relationships between significant and common policies and in the auxiliary discussion of the diversity of semantically distinct policies in Appendix A.1.2.

**Extensions**. Most of the found extensions can be traced to Korea. Without an explanation of this situation, it may be that the observed, substantial use of extensions is not representative for P3P authoring generally.

## 3.6 Related Work

We relate our P3P effort to the general area of empirical program analysis. We also explain how our work differs from other P3P studies that are essentially studies on P3P adoption. We briefly mention other literature that critically discusses P3P, and we comment on the situation in the web-privacy domain more broadly.

### Empirical Program Analysis

There is early work on simple static and dynamic program analysis for FORTRAN, Cobol, and Pascal with the objective of informing compiler implementors and possibly language designers [115, 157, 34, 40]. Static analysis is typically based on simple structural program properties. Dynamic analysis may examine aspects such as depth of recursion. We also measure structural properties. Dynamic analysis is not directly relevant for P3P. We pick up the extra opportunity of semantics-based measurements. Our work may be of interest for P3P tool providers and language designers, and we provide insights into P3P's use cases.

In [160], the use of APL language features is found to obey an 80-20 rule. We have checked that the use of P3P's base data schema obeys such a rule, too. That is, about 80 % of all P3P policies in the corpus (in fact, 84.15 %, i.e., 5,202 policies) use only 20 % of all data references.

In recent years, Java programs (or byte-code programs) have been empirically analyzed [71, 24, 38]. The work of [24] makes a laudable effort to deeply study the mathematical distribution of simple structural properties. In contrast, our research questions are not concerned with the discovery of distributions.

The idea to empirically analyze policies (as opposed to programs) was also inspired by our earlier work on understanding the usage of XML schemas [120]. In this context, we also had to come up with non-classical forms of metrics and usage analyses for language elements.

### Studies of P3P Adoption

Several studies have analyzed P3P's adoption; we consider the following list as representative of this line of work: [57, 153, 44, 154]. The studies differ in objective, the method of obtaining a corpus, and analytical techniques. Table 3.21 provides a summary.

In three out of four cases, these earlier studies involved multiple sources such as ranking lists for websites, and they also used search engines, e.g., by locating websites through ranked lists of search terms. Our reliance on a single source is a threat to validity, but given the size of the corpus and the reputation of ODP, we should have covered many important websites. Still our distributions may be biased in that we may look at more unpopular sites than earlier studies.

Previous work, as listed above, has not analyzed metrics, cloning, semantics-based properties, P3P extensions, and language coverage (except for some basic considerations in [44]). This is largely a consequence of a focus on adoption as opposed to our focus on language-usage analysis as well as our background in software-language engineering.

### Critical Discussion of P3P

All above-mentioned work on P3P adoption regularly involves simple forms of validity checking, and the reported degrees of invalidity also along evolution of websites

| **Property** | **[57]** | **[153]** | **[44]** | **[154]** |
|---|---|---|---|---|
| Time stamp | 12/2006? | 11/2005 | 12/2006 | 11/2006 |
| Size of corpus | 3846? | 1482? | 3846? | 3282 |
| Number of sources | 11 | 6 | 5 | 1 |
| Syntax error rates | ○ | ○ | ● | ○ |
| Boundary breakdown | – | ● | – | ● |
| Website evolution | – | ● | ● | – |
| P3P language coverage | – | – | ○ | – |
| Privacy classification | ● | – | ○ | – |
| Legislation | – | – | – | ● |
| HRPP interpretation | – | – | ● | – |

Question marks in the cells mean that we are not confident that the available descriptions could be unambiguously interpreted; '●' means that the study focuses on this issue; '○' means that the study touches upon the issue, but does not focus on it.

**Legend of properties**:

- 'Time stamp'—the date when the corpus was determined.
- 'Size of corpus'—number of policies considered by the study.
- 'Number of sources'—number of directories, rankings, etc.
- 'Syntax error rates'—analysis of syntactical validity.
- 'Boundary breakdown'—analysis of geographical or linguistic distribution.
- 'Website evolution'—analysis of changed, corrected, added, and removed policies.
- 'P3P language coverage'—analysis of vocabulary usage as in §3.2.1.
- 'Privacy classification'—analysis of privacy profiles.
- 'Legislation'—analysis of compliance of policy with applicable
- 'HRPP interpretation'—analysis of HRPP (Human Readable Privacy Policy)

**Table 3.21.** Related work on P3P adoption

can be regarded as one theme of critical discussion of P3P. It may just be necessary that the deployment process for policies would involve mandatory validation.

From a language engineer's point of view, a major issue with P3P is the lack of a standardized, sufficiently formal, and comprehensive semantics. One critically relies on such a semantics for complete validation of policies and, even more so, for policy enforcement. For instance, W3C's *APPEL* language [188] targets the problem of user preferences on top of P3P. Because of the weak specification of P3P, APPEL also ends up querying P3P in an 'unstable' manner: Query results differ for 'obviously' meaning-preserving variations of policies. APPEL's definition has been criticized widely [90, 14, 15, 197] and alternative languages for preferences have been proposed [14, 15], but ultimately it is P3P that needs to be more strongly defined.

**Web-privacy Domain**

P3P is a machine-readable language allowing to express data collection practices of websites. P3P was designed to make it easier for customers to become aware of and understand websites' policies.

An alternative approach to expressing privacy policies in a human-friendly format uses seals (say, badges or pictograms) placed on pages of websites. These seal programs, e.g., TRUSTe [183], are meant to guarantee trustworthiness of the website which have to meet certain requirements for different types of seals subject to a certification mechanism. It should be interesting to compare the types of seals considered by these approaches with our findings of common policies.

The idea of expressing privacy practices of a website with a concise visual form such as a pictogram is appealing and inspired other attempts. For example, within the community of Mozilla's users, there is an initiative of reducing the complexity of privacy policies and having visual aids for quick indication. (Likewise, Creative Commons licenses provide simple, standardized alternatives to the paradigm of traditional copyright [143].) There is also an effort by the law firm Hunton & Williams—layered privacy notices that provide users with a high-level summary of a privacy policy [178, 179]. Another effort on information design that improves comprehensibility of privacy policies exploits the idea of nutrition label that lists facts about the product. The authors suggest to present information in a form of a colored grid [107].

## 3.7 Conclusion

With the possible exception of P3P, there is no widely adopted, language-based approach to privacy in web-based systems. It is very common that web-based systems address privacy essentially by means of human-readable policies, settings dialogs, and perhaps seal programs such as TRUSTe. Nevertheless, language-based solutions may be needed to enable flexible, transparent, and enforceable privacy policies for web-based systems.

Recent communications by P3P advocates and experts [164, 43] made a number of observations about the adoption issues with P3P as well as design or lifecycle problems with the language; these communications also made a few proposals on how the web-privacy domain should proceed and what lessons to learn from P3P. To some extent, the communications also take advantage of empirical research, but our study in empirical analysis of language usage for P3P is the first that analyzes the language from a software language engineer's point of view.

Our analysis of correct use for standardized and provisional language constraints suggests that future language-based solutions must be more rigorous in specifying the language and enforcing correct use effectively. Our analysis of circulating P3P extensions based on P3P's extension mechanism has revealed that the mechanism has not been broadly adopted. However, the available examples of extensions clearly hint at the need of additional details to be covered by privacy policies. We propose that future language-based solutions must provide a more effective extension mechanism. Further analyses led to results that identified common policies and suggested simplifications of the vocabulary.

Future empirical research should deliver a comparison of P3P's vocabulary with vocabulary used elsewhere for web privacy such as in approaches that use human-readable policies, settings dialogues, or other approaches that rely on certification and seals, as discussed in §3.6. It is easy to find data points that show misalignment between P3P's vocabulary and the needs of major websites. Consider, for example, Facebook's privacy settings of sharing user information: everyone, or only friends, or friends of friends, or sets of certain people—these categories and their relationships are not modeled by P3P. Empirical research may help here in deriving a more comprehensive or more extensible vocabulary.

In the present effort, the issues of language adoption or perhaps even decline were not of central interest. It may be interesting though to further study the causes for P3P's decline. For instance, our clone detection results showed that there seem to exist many websites that do not even customize policies to their entity. We also refer to Appendix A.1.1, where we provide some data on the disappearance of policies over time (in terms of online availability) for the corpus. Can we further expand on these and other symptoms in ultimately concluding that many policies were never considered 'actionable' or they were designed to be extremely 'permissive' without usefully promising privacy?

We hope that Software Language Engineering, as a discipline, will continue to respond to the need for improved, possibly language-based solutions for the web-privacy domain. As our study revealed, there are interesting challenges regarding, for example, the specification and usability of policy languages, the procedural validation of policies within their lifecycle, and the effective provision of an extension mechanism.

# 4

# A Study of APIs

In this chapter, we present an empirical study of API usage. We begin with an initial exploration of the topic in the context of open-source Java projects, where we demonstrate examples of large-scale API usage analysis and detect different styles of usage (framework-like vs. library-like). We investigate further framework usage by developing a *framework profile*—a range of reuse-related metrics to measure the as-implemented design and its usage—and applying it to the Microsoft .NET Framework. Finally, based on the developed analyses, we introduce a catalogue of exploration activities to capture API usage accompanied by a tool.

## Road-map of the chapter

- Section 4.1 provides brief motivation for the study.
- Section 4.2 analyzes API usage in Java context.
- Section 4.3 analyzes API usage on the example of .NET framework.
- Section 4.4 illustrates the developed intuitions as exploratory activities.
- Section 4.5 discusses threats to validity for this empirical study.
- Section 4.6 discusses related work.
- Section 4.7 concludes the chapter.

## Reproducibility

We provide additional data (details of implementation, source code of the tool, etc.) on the supplementary websites[1],[2],[3].

## Related publications

Research presented in this chapter underwent the peer-reviewing procedure and was published in the proceedings of Symposium on Applied Computing in 2011 [4], Working Conference on Reverse Engineering in 2011 [3], and International Conference on Program Comprehension in 2013 [6].

[1] `http://softlang.uni-koblenz.de/sourceforge`
[2] `http://softlang.uni-koblenz.de/dotnet/`
[3] `http://softlang.uni-koblenz.de/explore-API-usage/`

## 4.1 Introduction

(A brief introduction to APIs is provided in Part I, Prerequisites, Section 2.3.2.)

The use (and the design) of APIs is an integral part of software development. Projects are littered with usage of easily a dozen APIs; perhaps every third line of code references some API [4]. Accordingly, understanding APIs or their usage must be an important objective. Much of the existing work focuses on some form of documentation or discovery of API-usage scenarios perhaps by code completion or code search [98, 199, 172, 58]. A contrary approach is an exploration-based approach to understanding API usage in a systematic manner.

We develop this approach by examining API usage from two complementary angles of view. Application programming interfaces, APIs[4], are intended to expose specifications for routines, data structures, object classes, variables, and so on. From the point of view of the usage, APIs are essentially languages within languages. From the point of view of the intention, APIs provide reusable pieces of functionality. These considerations give rise to the following research directions when investigating API usage in the wild:

- Are all "language elements" of an API used? I.e., what is the coverage of the API by projects. And dually, what are the different "language elements" of APIs that a project makes use of? I.e., what is the footprint of APIs in the project.
- What are the ways of providing reusability? I.e., what is a classification (a profile) of potential reuse of an API. And dually, are those ways used equally in projects? I.e., what is the profile of the actual reuse of the API.

Having followed these two complementary research directions, we develop a catalogue of exploratory activities (accompanied by a tool), which can be helpful both to API developers and API users (i.e., project developers).

## 4.2 Java APIs

In this study, we follow the first research direction identified in the Section 4.1, namely:

- Are all "language elements" of an API used? I.e., what is the coverage of the API by projects. And dually, what are the different "language elements" of APIs that a project makes use of? I.e., what is the footprint of APIs in the project.

We place our research in the context of API migration. Given a programming domain, and given a couple of different APIs for that domain, it can be challenging to devise transformations or wrappers for migration from one API to the other. The APIs may differ with regard to types, methods, contracts, and protocols so that

[4] We use the term API to refer both to a public programming interface and its actual implementation as a software library for reuse.

actual API migration efforts must compromise with regard to automation and correctness [22, 20].

Several researchers, including ourselves, are working towards general techniques for reliable and scalable API migration. Because of the complexity of transformations and wrappers for migration as well as the difficulty of proving them correct, it is also advisable to leverage *diverse knowledge about actual API usage*.

In the current section, we describe an approach to large-scale *API-usage analysis* for the analysis of open-source Java projects. Our approach covers checkout, building, tagging with metadata, fact extraction, analysis, and synthesis with a large degree of automation. We describe a few examples of API-usage analysis; they are motivated by API migration. Overall, API-usage analysis helps with designing and defending mapping rules for API migration in terms of relevance and applicability.

### 4.2.1 Overview of the Approach

We rely on methods and techniques that are commonly used in reverse engineering and program understanding. In particular, we need to set up a corpus of software projects to be used for data mining; we also need to provide a fact-extraction machinery to build a database of program facts. Additionally, we need to add metadata about APIs—as we operate in *the domain of the programming domains and their APIs*. Any specific form of API-usage analysis is then to be implemented through queries on the fact base (database) that is obtained in this manner.

#### Setting up the corpus

The objectives of a specific effort on API-usage analysis should obviously affect the provision of a corpus. In this study, we are mainly interested in extracting *evidence (facts) about API usage* from as many projects as possible. While corpora of dozens of well chosen projects (such as the one of [24]) are well suited for many data mining purposes (e.g., for the analysis of simple structural properties (metrics) of Java software), they are potentially limited with regard to API features that they exercise. For this reason, we are interested in large-scale efforts where API-usage data is systematically mined from open-source repositories. In principle, one could still include specific ‘well-known’ projects manually into the resulting corpus, if this is desired.

#### Provision of a fact extractor

We need to be able to reliably link facts of API usage to the actual APIs and their types, methods, etc. Hence, fact extraction must be syntax- and type-aware. (For instance, the receiver type in a method call must be known in order to associated calls with API methods.) We use fact extraction based on *resolved ASTs*. However, this choice basically implies that we only consider built (‘buildable’) projects, which may result in a bias. Therefore, we also incorporate an additional token-based (as opposed to AST-based) fact extractor into the architecture so that some more basic analyses are still feasible.

### Addition of API metadata

Along with building many projects, one encounters many APIs. In the case of the Java platform, there are Core Java APIs and third-party APIs. Based on package and types names, one can identify these APIs, and assign names. For instance, certain types in the package `java.util` account for the 'Core Java API for collections'. One can also associate programming domains with APIs: GUI, XML, Testing, etc.

Third-party APIs reveal themselves in the process in two principle ways. First, projects may fail to build with 'class not found' errors, which are to be manually resolved by supplying (and tagging) the corresponding APIs through web search and download. Second, the corpus can also be analyzed for cross-project reuse. That is, one can automatically identify API candidates by querying for packages whose methods were called (and potentially declared and compiled) in at least two projects.

## 4.2.2 A Study of SourceForge

We will now describe the instantiation of the above approach for the study of Java libraries. As a source for Java projects—with the goal of collecting a large-scale corpus—we have selected SourceForge online repository[5], a web-based source code repository, existing since 1999.

### Project selection

Based on available metadata for all SourceForge projects, we determined the list of potential Java projects. In the present study, as a concession to scalability and simplicity of our implementation, we only downloaded projects with a SourceForge-hosted SVN source repository, and we only considered Java projects with Ant-based build management. We used a homegrown architecture for parallel checkout and building. The selected SourceForge projects were fetched in October 2008.

### Resolution of missing API packages

Obviously, SourceForge projects may fail to build for diverse reasons: wrong platform, missing configuration, missing JARs, etc. We wanted to bring up the number of buildable projects with little effort. We addressed one particular reason: 'class not found' compilation errors. To this end, our build machinery compiles a summary of unresolved class names (package names) for a full build sweep over all projects. This summary *ranks* package names by frequency of causing 'class not found' errors so that the manual effort for supplying missing APIs can be prioritized accordingly. We searched the web for API JARs using unresolved package names as search strings. We downloaded these JARs, added them to the build path, and ran the automated build again. We repeated this step until the top of the list of missing packages would contain packages referenced only by 1-2 projects. This process provided us with

[5] `http://sourceforge.net/`

1,476 built projects, where approx. 15 % of these projects were made buildable by our resolution efforts. In the end, we were able to build 90.05 % of all downloaded SourceForge projects that satisfied our initial criteria (Java, SVN, ANT). The process resulted in an API pool of 69 non-Core Java APIs.

### Fact extraction

We carried out resolved AST-based fact extraction by means of a compiler plug-in for `javac`, which is activated transparently as projects are built. In this study, we extracted facts about method declarations, method calls, and subtype relationships. We interpret the term *method* to include instance methods, static methods and constructors. All facts were stored in a relational database using a trivial relational schema.

We only used AST-based facts from projects with successful builds. For all projects, we performed token-based fact extraction to count NCLOC (non-comment lines of code) for Java sources and to determine all package names from imports. The importing facts give an indication of, for example, the APIs that are used in projects that do not build.

### Reference projects

We made an effort to identify a control group of (buildable) reference projects that could be said to represent well thought-out, actively developed and usable software. Such a control group allows us to check all trends of the analyses for the full corpus by comparison with the more trusted reference projects. Several charts in this study show all projects vs. reference projects for comparison. We automatically identified the reference projects by means of SourceForge's metadata about maturity status of the project, number of commits, and dates of first and last commits. That is, we selected projects that rate themselves as 'mature' or 'stable', have a repository created more than two years ago, and have more than 100 commits to the repository. This selection resulted in 60 reference projects out of all the 1,476 built projects.

### Size metrics for the corpus

Numbers of projects and their NCLOC sizes, and other metrics are summarized in Table 4.1 and Table 4.2. We use the metric *MC* for the number of method calls.

Figure 4.1 presents the distribution of size metrics (NCLOC, MC) for the corpus (y-axis is normalized w.r.t. the maximum of the metric in each group). As one can see, both metrics correlate reasonably. The maximum of NCLOC in the whole corpus (incl. unbuilt projects) is 25,515,312, the maximum of NCLOC among built projects is 1,985,977, which implies a factor 12.85 difference. Hence, we are missing the biggest projects currently. The maximum of MC among built projects is 228,242.

| Metric | Value |
|---|---:|
| Projects | 6,286 |
| Source files | 2,121,688 |
| LOC | 377,640,164 |
| NCLOC | 264,536,500 |
| Import statements | 14,335,066 |

**Table 4.1.** Summary of token-based analysis (with all automatically identified Java/SVN projects on SourceForge)

| Metric | Value |
|---|---:|
| Projects with attempted builds | 1,639 |
| Built projects | 1,476 |
| Packages | 46,145 |
| Classes | 198,948 |
| Methods | 1,397,099 |
| Method calls | 8,163,083 |

**Table 4.2.** Summary of AST-based analysis

| API | Domain | Core | # projects | # calls | |
|---|---|---|---:|---:|---:|
| | | | | total | distinct |
| Java Collections | Collections | yes | 1,374 | 392,639 | 406 |
| AWT | GUI | yes | 754 | 360,903 | 1,607 |
| Swing | GUI | yes | 716 | 581,363 | 3,369 |
| Reflection | Other | yes | 560 | 15,611 | 154 |
| Core XML | XML | yes | 413 | 90,415 | 537 |
| DOM | XML | yes | 324 | 52,593 | 180 |
| SAX | XML | no | 310 | 13,725 | 156 |
| log4j | Logging | no | 254 | 43,533 | 187 |
| JUnit | Testing | no | 233 | 71,481 | 1,011 |
| Comm.Logging | Logging | no | 151 | 21,996 | 88 |

**Table 4.3.** Top 10 of the known APIs (sorted by the number of projects using an API)

## Provision of API metadata

Both Core Java APIs and manually downloaded JARs were processed by us to assign metadata: name of the API, the name of a programming domain, one or more package prefixes, and potentially a white-list of API types. Table 4.3 lists the top 10 of all the 77 manually tagged APIs together with some metadata and metrics (full list of known APIs is available in Appendix A.2.1). We used a special reflection-based fact extractor for the visible types and members of the API JARs. (Alternatively,

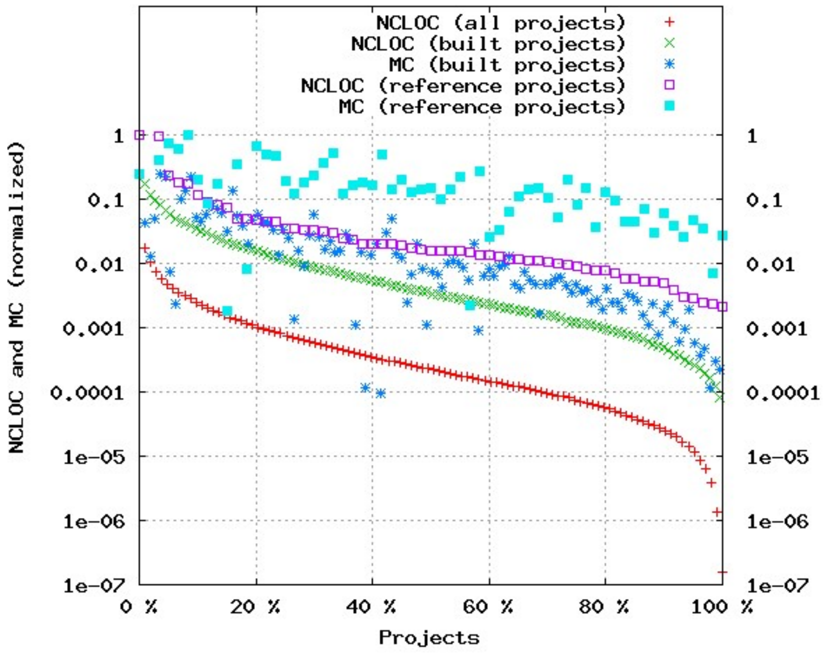


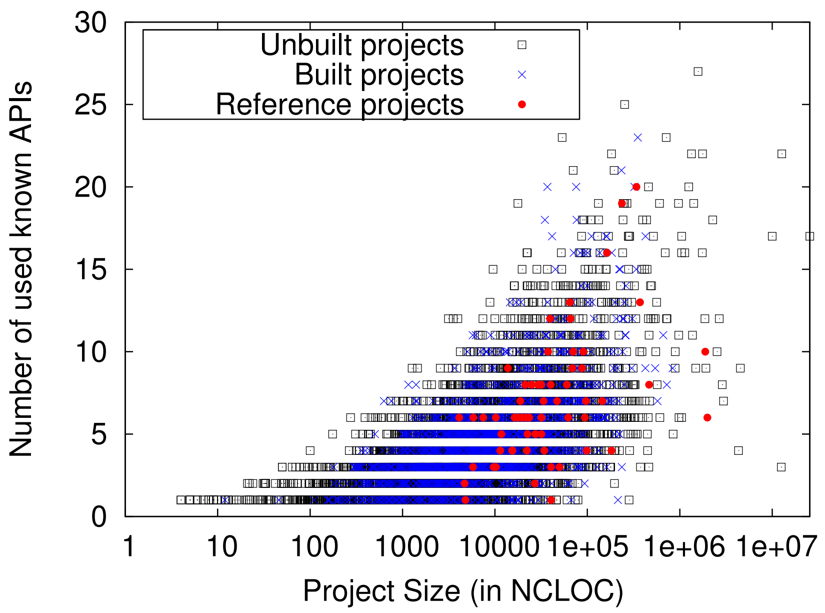
**Figure 4.1.** Size metrics (NCLOC, MC) for built and unbuilt projects (thinned out). The projects are ordered by the values for the NCLOC metric

one could also attempt to leverage API documentation, but such documentation may not be available for some of the APIs, and it would take extra effort to establish consistency between JARs and documentation.) These facts are also stored in the database, and they are leveraged by some forms of API-usage analysis.

### 4.2.3 Examples of API-usage Analysis

We will introduce a few examples of API-usage analysis. In each case, we will provide a motivation related to API migration and language conversion—before we apply the analysis to our SourceForge corpus.

Admittedly, the statistical analysis of a corpus does not directly help with any specific migration project. However, the reported analyses as such are meaningful for single projects, too (perhaps subject to refinements). For instance, we will discuss API coverage below, and such information, when obtained for a specific project, directly helps prioritizing migration efforts. In this study, we take a statistical view on the corpus to indicate the de-facto range for some of the measures of interest.



| Projects | Min | 1st Q | Median | Mean | 3rd Q | Max |
|---|---|---|---|---|---|---|
| Unbuilt | 1 | 2 | 4 | 4.4 | 6 | 27 |
| Built | 1 | 3 | 4 | 4.7 | 6 | 23 |
| Reference | 1 | 4 | 6 | 6.9 | 8 | 20 |

**Figure 4.2.** Numbers of known APIs used in the projects (reference projects are plotted on top of built projects which in turn are plotted on top of unbuilt projects)

## API footprint per project

We begin with a very simple, nevertheless informative API-usage analysis for the *footprint of API usage*. There are different dimensions of footprint. Below, we consider the numbers of used APIs and used (distinct) API methods. In the online appendix, we also consider the ratio of API calls to all calls. In extension of these numbers, we could also be interested in the 'reference projects $\times$ API pool' matrix (showing for each project the combination of APIs that it uses).

The APIs or API methods used in a project provide insight into the API-related complexity of the project. In fact, such footprint-like data serves as a proxy for the API dependence or platform dependence of a project. In [114], we mention such API dependence as a form of *software asbestos*. In the following, we simply count the number of APIs used in a project as a proxy for the difficulty of API migration. Ultimately, a more refined analysis is needed such that *specific* (known to be difficult)

API combinations are counted, and attention is payed to the status of whether these API combos are really exercised in one program scope or only separately.

In this context, we need to define what constitutes usage of an API. One option would be to count each method call with an API's type as static receiver type (in the case of an instance call), or as the hosting scope (in the case of a static call), or as the constructed type (in the case of a constructor call). Another option is to count any sort of reference to an API's types (including the aforementioned positions of API types in method calls, but counting additionally local variable declarations or argument positions of method declarations and method calls). Yet another option is to consider simply imports in a project. The latter option has the advantage that we can measure such imports very easily—even for unbuilt projects. Indeed, the following numbers were obtained by counting imports that were obtained with the token-based fact extractor.

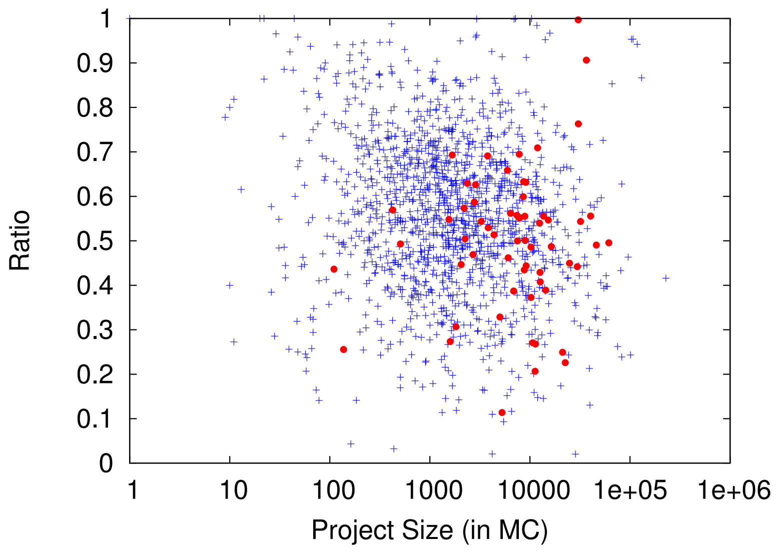
Figure 4.2 shows the number of known APIs (y-axis) that are used in the projects ordered by NCLOC-based project size (x-axis). Unbuilt, built, and reference projects are distinguished. The listed maxima and quartiles give a sense of the API footprint in projects in the wild. The set of unbuilt projects exercises a higher maximum of used APIs than the set of built projects—potentially because of a correlation between the complexity of projects in terms of the number of involved APIs and the difficulty to build those projects.

We also need to clarify how to measure usage of API methods. That is, how to precisely distinguish distinct methods so that counting uses is well defined. Particularly, in the case of instance method calls, the situation is complicated due to inheritance, overriding, 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 *static receiver type* in a call. Additionally, we may also count the call towards subtypes (subject to a polymorphism-based argument: the runtime receiver type may be a subtype) and supertypes (subject to an inheritance-based argument: the inherited implementation may be used, if not inherited). Such inclusion could also be made more precise by a global program analysis.

Fig. A.6 shows the usage of known API methods relative to all methods in a project—both in terms of calls. The smaller the ratio (the closer to zero), the lower the contribution of API calls. The quartiles show that in most projects, about each second method call is an API call. As far as instance-method calls are concerned, the figure distinguishes API vs. project-based method calls solely on the grounds of the static receiver type of methods.

Figure 4.4 shows the numbers of distinct API methods used in the built projects of the corpus; reference projects are highlighted. Methods on sub- and supertypes of static receiver types were not included. For simplification, we also considered overloaded methods as basically one method.

There is a trend of increasing API footprint with project size. Both axes are logarithmic, but project size grows more quickly than the count of distinct API methods. Most projects, even most of the largest ones, use less than 1,000 distinct API methods. As the table with maxima and quartiles shows, there are a few projects with



| Projects | Min | 1st Q | Median | Mean | 3rd Q | Max |
|---|---|---|---|---|---|---|
| All | 0.02 | 0.45 | 0.56 | 0.56 | 0.67 | 1 |
| Reference | 0.11 | 0.43 | 0.5 | 0.5 | 0.57 | 0.99 |

**Figure 4.3.** Ratio of API method calls to all method calls

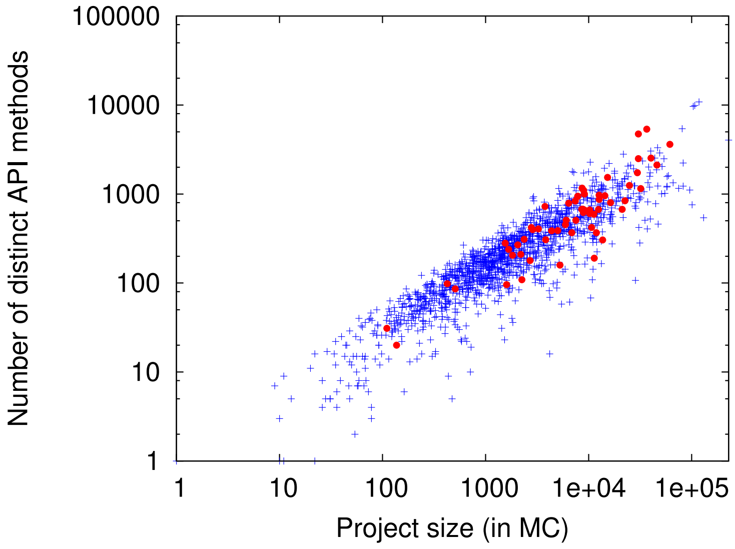
exceptionally high counts. We have verified for these projects that they essentially implement or test large frameworks (such as `ofbiz.apache.org`). That is, these outliers embody large numbers of 'self-calls' for a large number of API methods.

Additional results on API usage are provided in Appendix A.2.1

## API coverage by the corpus

An important form of API-usage analysis concerns API coverage; see, for example, the discussion of coverage in the API migration project of [22]. That is, coverage information is helpful in API migration as means to prioritize efforts, and to leave out mapping rules for obscure parts of the API. Coverage information is also helpful in improving API usability [101, 96].

As it is the case with other forms of API-usage analysis, API coverage may be considered for either a specific project, or, cumulatively, for all projects in a corpus. For instance, for any given API, we may be interested in the API types (classes and interfaces) that are exercised in projects by any means: extension, implementation, variable declaration, all kinds of method calls, and other, less obvious idioms (e.g., instance-of tests). At a more fine-grained level, we may be interested in the exercised members for each given API type. Hence, qualitative measurements focus on types



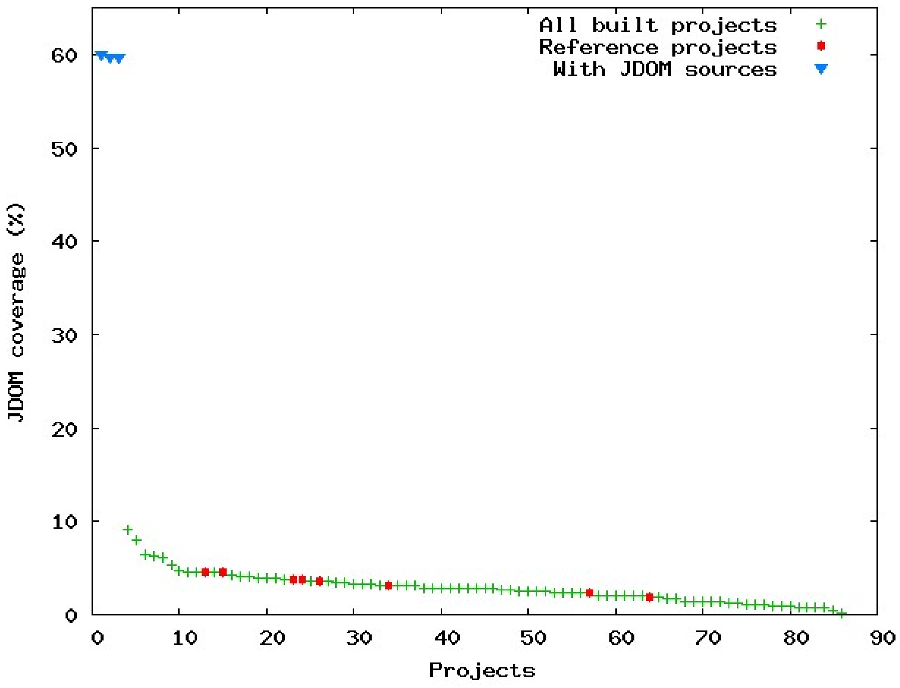
| Projects | Min | 1st Q | Median | Mean | 3rd Q | Max |
|---|---|---|---|---|---|---|
| All | 1 | 94 | 199.5 | 370.7 | 423 | 10,850 |
| Reference | 20 | 305.8 | 611 | 866.2 | 948.8 | 5,351 |

**Figure 4.4.** Numbers of distinct API methods used in the projects (without distinguishing APIs)

and members that are exercised at all, while quantitative measurements rank usage by the number of occurrences of a type or a member or other weights.

Assuming a representative corpus, further assuming appropriate criteria for detecting coverage, we may start with the naive expectation that a good API should be covered more or less by the corpus. Otherwise, the API would contain de-facto unnecessary types and methods—which is clearly not in the interest of the API designer. However, it should not come as a surprise that, in practice, APIs are not covered very well—certainly not by single meaningful projects [22], but—as our results show—not even by a substantial corpus; see below.

We have actually tried to determine two simple coverage metrics for all 77 known APIs: i) a percentage-based metrics for the *types*; ii) another percentage-based metrics for all *methods*. However, we do not feel comfortable presenting a chart of those metrics for all known APIs here. Unless considerable effort is spent on each API, such a chart may be disputable. The challenge lies in the large number of projects and APIs, the different characteristics of the APIs (e.g., in terms of their use of subtyping), aspects of cloning, and yet other problems. Some of the issues will be demonstrated for selected APIs below.



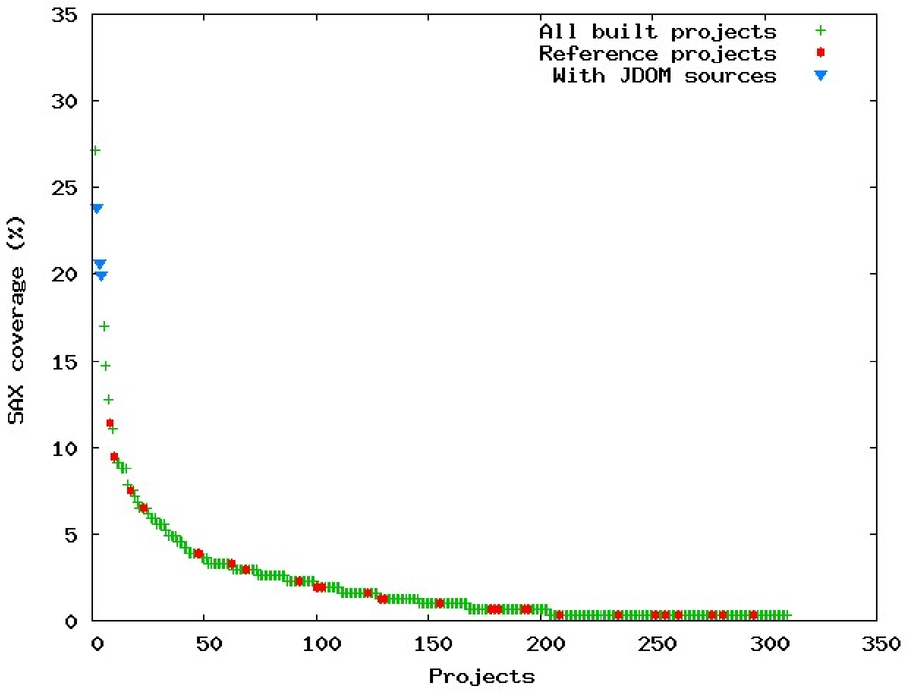
| Projects | Min | 1st Q | Median | Mean | 3rd Q | Max |
|---|---|---|---|---|---|---|
| All built | 0.16 | 1.9 | 2.8 | 4.9 | 3.7 | 59.9 |
| Reference | 1.9 | 2.9 | 3.7 | 3.4 | 3.9 | 4.6 |

**Figure 4.5.** Usage of JDOM's distinct methods

Let us investigate coverage for *specific* APIs. As our first target, we pick JDOM—a DOM-like (i.e., tree-based, in-memory) API for XML processing. We know that JDOM is a 'true library' as opposed to a framework. Regular client code should simply construct objects of the JDOM classes and invoke methods directly. We mention these characteristics because library-like APIs may be expected to show higher API coverage than framework-like APIs—if we measure coverage in terms of *called* methods, as we do here. In this study, in the case of a call to an instance method, we only count the method on the immediate static receiver type as covered. We have checked that the inclusion of super- and subtypes, as discussed earlier, does not change much the charts described below.

Initially, we measured cumulative coverage for the methods of the JDOM API to be 68.89 %. We decided to study the contribution of the different projects. There are 86 projects with JDOM usage among the built projects of the corpus. Figure 4.5 shows the percentage-based coverage metrics for the methods of the JDOM API for those JDOM-using projects. The table with maxima and quartiles gives a good indication of the relatively low usage of the JDOM API.

Obviously, 3 projects stand out with their coverage. We found that these projects should not be counted towards cumulative coverage because these projects contain JDOM clones in source form. That is, it the API calls within the API's implementation imply artificial coverage for more than half of all JDOM methods. Without those outliers, the cumulative coverage is considerably lower, only 24.10 %.



| Projects | Min | 1st Q | Median | Mean | 3rd Q | Max |
|---|---|---|---|---|---|---|
| All built | 0.33 | 0.33 | 0.98 | 2.22 | 2.61 | 27.12 |
| Reference | 0.33 | 0.33 | 1.14 | 2.37 | 3.02 | 11.44 |

**Figure 4.6.** Usage of SAX' distinct methods

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—*if* we measure coverage in terms of called methods.

We measured cumulative coverage for the methods of the SAX API to be 50.98 %. This relatively high coverage was surprising. There are 310 projects with SAX usage among the built projects of the corpus. Figure 4.6 shows the percentage-based coverage metrics for the methods of the SAX API for those SAX-using projects. We found that three of the projects with the highest coverage were in fact the previously discussed projects with JDOM clones in source 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 the SAX API within the DOM API which brings up coverage of the SAX API. Even if we removed those 3 projects, the cumulative coverage only drops down a little to 49.34 %.

We also found other reasonable reasons for relatively high coverage. There are projects that use inheritance and composition to define new handlers (e.g., `http://corpusreader.sourceforge.net/`) so that API methods may get called through 'super' or delegation. As the quartiles show in the figure, most projects use a small percentage of the SAX API. Most of the relevant methods are concerned with processing the parameters of the handlers. Many of the SAX projects use (largely) predefined handlers, e.g., for validation—thereby implying a very low coverage.

## Framework-like API usage

Finally, we introduce an analysis for framework-like API usage: What API types are typically implemented and extended, if any? Also, can we determine whether a given API is presumably more framework-like (less class library-like) than another API? What are the parts of an API that account for framework-like usage? In the context of API migration, proper framework-like usage is very challenging because it implies an 'inversion of control' in applications, which is very hard to come by with mapping rules [20].

| API | # projects | | | # methods | | # dist. methods | | # derived types | | # API types | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | impl | ext | any | impl | over | impl | over | int | cl | int | cl |
| Swing | 173 | 381 | 391 | 2,512 | 11,150 | 305 | 645 | 443 | 1,859 | 39 | 92 |
| AWT | 194 | 75 | 225 | 4,201 | 756 | 593 | 176 | 651 | 120 | 31 | 24 |
| Java Collections | 120 | 0 | 120 | 986 | 0 | 16 | 0 | 208 | 0 | 3 | 0 |
| SAX | 28 | 21 | 42 | 428 | 90 | 85 | 21 | 37 | 29 | 12 | 3 |
| JUnit | 3 | 38 | 40 | 4 | 344 | 4 | 19 | 3 | 46 | 2 | 2 |
| Core XML | 11 | 5 | 14 | 89 | 13 | 17 | 4 | 14 | 5 | 9 | 3 |
| SWT | 5 | 8 | 10 | 37 | 86 | 4 | 13 | 25 | 11 | 3 | 3 |
| log4j | 1 | 8 | 8 | 25 | 87 | 7 | 9 | 2 | 9 | 2 | 3 |
| Reflection | 7 | 0 | 7 | 10 | 0 | 1 | 0 | 7 | 0 | 1 | 0 |
| JMF | 4 | 2 | 6 | 8 | 6 | 6 | 3 | 4 | 3 | 3 | 3 |

**Table 4.4.** Top 10 of the APIs with framework-like usage (sorted by the sum of numbers of API-interface implementations and API-class extensions; see the first 3 columns)

More specifically, by *framework-like usage*, we mean any sort of idiomatic evidence for refining, configuring, and implementing API types within client code. In particular, we may measure i) extensions (i.e., client classes that extend API classes); ii) implementations (i.e., client classes that implement API interfaces); iii) overrides (i.e., client classes that subclass API classes and override inherited API methods). Obviously, there are facets that may be harder to identify generically. For instance, if a framework would involve plug-in or configuration support based on regular method calls, then such framework-like usage would be missed by i)—iii). There is again a way of defining framework-like usage in a cumulative way—very similar to coverage analysis. That is, for a given API, we may determine the set of API types that are ever involved in framework-like usage.

In reality, many APIs allow for both—class library-like and framework-like usage. For instance, the Core Java API DOM is essentially interface-based so that new providers can be implemented, but there are default implementations for the important use case of DOM as an in-memory XML API. In contrast, there are other APIs that are subject to framework-like usage more inevitably. For instance, the Core Java API Swing is often used in a way that the `JPanel` class is overridden.

In our corpus, 35 out of all 77 known APIs exercise a measurable facet of framework-like usage. Table 4.4 lists the top 10 of these APIs. In the table, we also show the numbers of API methods that are implemented or overridden throughout the corpus: we show both absolute numbers of implementations/overrides and the numbers of distinct methods. Further, we show the number of derived types in client code, and the number of API types ever exercised through framework-like usage.

Surprisingly, the table shows that there are only 7 APIs that are used in 10 or more projects in a framework-like usage manner. This clearly suggests that our corpus (of built projects) and our selection of APIs is trivial in terms of framework-like usage. Many APIs do not show up at all in the table—despite heavy usage in the corpus. For instance, DOM-like APIs like JDOM or XOM do not show up at all, which means that they are only used in a class library-like manner. The DOM API itself is subject to API-interface implementations in a number of projects. In Appendix A.2.1, we also break down the numbers of the table 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.

## 4.3 .NET framework

In this study, we follow the second research direction identified in the Section 4.1, namely:

- What are the ways of providing reusability? I.e., what is a classification (a profile) of potential reuse of an API. And dually, are those ways used equally in projects? I.e., what is the profile of the actual reuse of the API.

As we have discussed in Section 4.2.3 (pp. 102 and 104), there are two major styles of APIs: library-like vs. framework-like. The former suggests simple construction of objects and direct invocation of methods, while the latter implies any sort of refining, configuring, and implementing API types within client code. In other words, library-like APIs use the basic form of reuse, while framework-like APIs exercise more sophisticated forms. Therefore, we focus specifically on frameworks in our study.

We place our research in the domain of comprehension. Suppose you need to (better) understand the architecture of a platform such as the Java Standard Edition[6], the Microsoft .NET Framework[7], or another composite framework. There is no silver bullet for such *framework comprehension*, but a range of models may be useful in this context. We suggest the notion of *framework profile* which incorporates characteristics of potential and actual reuse of frameworks. The approach is applied to the Microsoft .NET Framework and a corpus of .NET projects in an empirical study.

Framework comprehension supports reverse and re-engineering activities. In quality assessment of designs [182], framework profiles help understanding frameworks in a manner complementary to architectural smells, patterns, or anti-patterns. Also, one can compare given projects with the framework profile. More specifically, in framework re-modularization, framework profiles help summarizing the status of modularization and motivating refactorings [54]. In API migration [21], framework profiles help assessing API replacement options with regard to, for example, different extensibility characteristics. Finally, framework profiles help in teaching OO architecture, design, and implementation [163].

### 4.3.1 Methodology

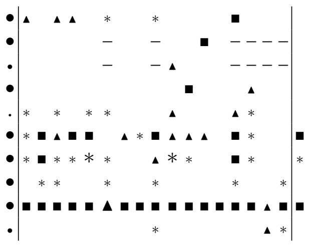
#### Research hypothesis

Platforms such as JSE or .NET leverage programming language concepts in a systematic manner to make those frameworks reusable (say, extensible, instantiatable, or configurable). It is challenging to understand the reuse characteristics of frameworks and actual reuse in projects at a high level of abstraction. *Software metrics on top of simple static and dynamic program analysis are useful to infer essential high-level reuse characteristics.*

#### Research questions

1. What are the interesting and helpful high-level characteristics of frameworks with regard to their potential and actual reuse?
2. To what extend can those characteristics be computed with simple metrics subject to simple static and dynamic program analysis?

[6] `http://www.oracle.com/us/javase`
[7] `http://www.microsoft.com/net/`



*Rows*: top-10 .NET namespaces, in terms of number of types.
*Middle block of columns*: actual reuse by project of the corpus.
*Leftmost column*: potential reuse in terms of specializability.
*Rightmost column*: summary of actual reuse.

**Figure 4.7.** Infographics for an excerpt of a framework profile for .NET

| .NET | Project | Repository | LOC | Description |
|---|---|---|---|---|
| 3.5 | Castle ActiveRecord | GitHub | 30,303 | Object-relational mapper |
| 4.0 | Castle Core Library | GitHub | 36,659 | Core library for the Castle framework |
| 3.5 | Castle MonoRail | GitHub | 58,121 | MVC Web framework |
| 4.0 | Castle Windsor | GitHub | 50,032 | Inversion of control container |
| 4.0 | Json.NET | Codeplex | 43,127 | JSON framework |
| 2.0 | log4net | Sourceforge | 27,799 | Logging framework |
| 2.0 | Lucene.Net | Apache.org | 158,519 | Search engine |
| 4.0 | Managed Extensibility Framework | Codeplex | 149,303 | Framework for extensible applications and components |
| 4.0 | Moq | GoogleCode | 17,430 | Mocking library |
| 2.0 | NAnt | Sourceforge | 56,529 | Build tool |
| 3.5 | NHibernate | Sourceforge | 330,374 | Object-relational mapper |
| 3.5 | NUnit | Launchpad | 85,439 | Unit testing framework |
| 4.0 | Patterns & Practices - Prism | Codeplex | 146,778 | Library to build flexible WPF and Silverlight applications |
| 3.5 | RhinoMocks | GitHub | 23,459 | Mocking framework |
| 2.0 | SharpZipLib | Sourceforge | 25,691 | Compression library |
| 2.0 | Spring.NET | GitHub | 183,772 | Framework for enterprise applications |
| 2.0 | xUnit.net | Codeplex | 23,366 | Unit testing framework |

**Table 4.5.** .NET projects in study's corpus (versions as of 19 June 2011)

## Research method

We applied an explorative approach such that a larger set of metrics of mainly structural properties was incrementally screened until a smaller set of key metrics and derived classifiers emerged. We use infographics (such as Figure 4.7) to visualize metrics, classifiers, and other characteristics of frameworks and projects that use them. The resulting claims are subject to validation by domain experts for the framework under study.

## Study subject

The subject of study consists of the Microsoft .NET Framework and a corpus of open-source .NET projects targeting different versions of .NET (2.0, 3.5, 4.0).

.NET (4.0) has 401 namespaces in total, but we group these namespaces reasonably, based on the tree-like organization of their compound names. For instance, all namespaces in the *System.Web* branch provide web-related functionality and can be viewed as a single namespace. In this manner, we obtained the manageable number of 69 namespaces; see Table 4.6.[8] In the rest of the chapter, we signify grouping by "*" as in *System.Web.**. Grouping is often used in discussions of .NET—also by Microsoft.[9]

Table 4.5 collects metadata about the corpus of the study. The following text summarizes the requirements for the corpus and the process of its accumulation; more information is available from the supplementary website.

One requirement is that the corpus is made up from well-known, widely-used and mature projects. We assume that such projects make good use of .NET.

Another requirement is that dynamic analysis must be feasible for the projects of the corpus. This requirement implies practically that we need projects with good available testsuites. The need for testsuites, in turn, implies practically that the corpus is made up from frameworks or libraries as opposed to, e.g., interactive tools. Admittedly, advanced test-data generation approaches could be used instead [181].

Yet another requirement is that the corpus is made up from open-source projects so that our results are more easily reproducible. Also, the instrumentation for static and dynamic analysis would be problematic for proprietary projects which usually commit to signed assemblies.

We searched CodePlex, SourceForge, GitHub, and Google Code applying the repository-provided ranking for popularity (preferably based on downloads). For the topmost approx. 30 projects of each repository we checked all the requirements, and in this manner we identified a diverse set of projects as shown in Table 4.5. These projects all use C# as implementation language. (In principle, our approach is prepared to deal with other .NET languages as well—since the analysis uses bytecode engineering.)

### 4.3.2 Reuse-related Metrics for Frameworks

We consider the as-implemented design of a framework—without considering any client code (i.e., 'projects'). We define reuse-related metrics for frameworks and screen them for .NET. We explain the metrics specifically in the form as they are needed for a *composite* framework (such as .NET) which consists of many *component* frameworks—to which we refer here as *namespaces* for better distinction. We usually consider metrics per namespace. Some of the metrics are specific to .NET's type system.

[8] We also excluded some namespaces that are fully marked as obsolete and an auxiliary namespace, *XamlGeneratedNamespace*, used only by the workflow designer tool.

[9] `http://msdn.microsoft.com/en-us/library/gg145045.aspx`

## Definition of metrics

The ‘overall potential for reuse’ is described by the following metrics:[10] **# Types**—the number of (visible, say reusable) types declared by a namespace; **# Methods**—the number of (visible, say reusable) methods declared by a namespace.

The types (say, type declarations) of a namespace break down into percentages as follows: **% Interfaces**, **% Classes**, **% Value types**, and **% Delegate types**. If a namespace has relatively few classes, then this may hint at intended, potential reuse that is different from classic, class-oriented OO programming. In the sequel, we refer to classes and interfaces as *OO types*. Further, **% Generic types** denotes the percentage of all types that are generic. Relatively many generic types hint at a framework for generic programming. Further, the classes of a namespace break down into percentages as follows; likewise for methods: **% Static classes**, **% Abstract classes**, and **% Concrete classes**.[11] Clearly, abstract classes and methods hint at potential reuse of the framework by specialization. Static classes hint at non-OO libraries and associated, different forms of reuse.

There are metrics for ‘specializability and sealedness’ of namespaces: **% Specializable classes**—the percentage of all classes that are either abstract or concrete but non-sealed, thereby excluding static and sealed classes; **% Sealed classes**—the percentage of all concrete classes that are sealed (final). Sealing explicitly limits reuse by specialization. The aforementioned metrics can also be taken to the method level. Further, we can incorporate interfaces into a metric for specializability: **% Specializable types**—the percentage of all OO types (i.e., all classes and interfaces) that are either specializable classes or interfaces—the latter being all specializable by definition. There are OO types that must be specialized before they can be reused in client code; we refer to them as orphan types subject to the following metrics: **% Orphan classes**—the percentage of all abstract classes that are never concretely implemented within the framework; **% Orphan interfaces**—the percentage of all interfaces that are never implemented within the framework; **% Orphan types**—the percentage of all abstract classes and interfaces that are either orphan classes or orphan interfaces.

‘Inter-namespace reuse’ is described by these metrics: **# Referenced namespaces**—the number of namespaces that are referenced by a given namespace; **# Referring namespaces**—the number of namespaces that are referring to a given

[10] In the case of .NET, non-private, non-internal types and methods are considered. Properties (in the .NET sense) are considered here as methods, which they are indeed at the byte-code level. All (visible) method declarations are counted separately. For instance, the initial declaration of a method as well as all overrides are counted separately. Overloads are counted separately, too.

[11] In .NET, value types include structs and enum types. Static classes are not considered concrete classes; neither are they considered abstract classes—regardless of encoding in MS IL; they are counted separately here. A delegate type is essentially a method type; they are counted separately here—regardless of the encoding in MS IL where delegate types derive from class *System.Delegate*.

namespace. Obviously, one can also define metrics for ‘inter-namespace specialization’; this is omitted here.

The earlier breakdown of type declarations can be matched by a similar breakdown of type references. In particular, type references due to method arguments give rise to **% Interface arguments**, **% Class arguments**, **% Value type arguments**, and **% Delegate arguments**. These metrics hint at certain forms of reuse. In particular, interface arguments give rise to interface polymorphism whereas delegate arguments give rise to closure-based parametrization.

Finally, there are metrics that relate to the degree of ‘specialization within a namespace’: **MAX size class tree** and **MAX size interface tree**—the size of the largest class or interface inheritance tree in terms of the node count only considering nodes from the given namespace. (We view a class as a root of an inheritance tree in the given namespace, if it derives from *System.Object* or a class in a different namespace; likewise for interfaces.) These are reuse-related metrics because, for example, they hint at polymorphism that can be leveraged for framework reuse in client code.

## Measurements for .NET

Table 4.6 lists the various metrics for all the .NET namespaces while sorting namespaces by *#Types*. We use an infographics such that most data is not displayed as numbers, but distribution-based visualization is used instead: ‘blank’ for zero values and bullets of increasing size, i.e., ‘·’, ‘•’, ‘●’, ‘⬤’, for values in the percentage intervals (0,25), [25,50), [50,75), [75,100) of the distribution. For each column, the cell(s) corresponding to the maximum value for the column display(s) the value instead of ‘⬤’. Medians as well as 25th and 75th percentiles for all columns are displayed at the bottom of the table.

| Namespace | # Types | # Methods | MAX size class tree | MAX size interface tree | # Referenced namespaces | # Referring namespaces | % Classes | % Interfaces | % Value types | % Delegate types | % Generic types | % Class arguments | % Interface arguments | % Value type arguments | % Delegate arguments | % Sealed classes | % Specializable classes | % Specializable types | % Orphan classes | % Orphan interfaces | % Orphan types |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| System.Web.* | 2327 | 29315 | ● | ● | 43 | . | • | • | . | ● |  | • | • | . | ● | . | • | • | • | • | . |
| System.Windows.* | ● | ● | ● | 82 | ● | . | • | . | . | ● | ● | . | . | . | ● | . | • | • | • | • | . |
| System.ServiceModel.* | ● | ● | ● | ● | 43 | . | ● | • | . |  | ● | ● | . | . | ● | • | . | . | ● | ● | ● |
| System.Windows.Forms.* | ● | ● | ● | ● | ● | • | . | . | ● | ● |  | . | . | ● | ● | . | • | • | • | • | . |
| System.Data.* | ● | ● | ● | ● | ● | . | • | • | • | • | ● | . | • | • | • | ● | . | . | ● | • | • |
| System.Activities.* | ● | ● | ● | ● | ● | . | ● | . | . | • | ● | ● | • | . | ● | ● | . | . | ● | ● | • |
| System.ComponentModel.* | ● | ● | ● | ● | ● | ● | . | ● | . | ● |  | • | ● | . | ● | . | • | • | ● | • | ● |
| System.Workflow.* | ● | ● | • | ● | ● | . | ● | • | . | • |  | ● | ● | . | ● | • | . | . | • | • | • |
| System.Xml.* | ● | ● | • | • | ● | ● | . | . | • | • |  | ● | . | . | . | . | • | • | ● | ● | • |
| System.Net.* | ● | ● | . | • | ● | • | . | . | • | ● |  | . | • | • | ● | . | • | • | ● | • | • |
| System.DirectoryServices.* | ● | ● | ● |  | ● | . | • |  | ● | • | ● | • | . | • | . | . | ● | ● |  |  |  |
| System | ● | ● | ● | ● | ● | 69 | . | . | • | ● | ● | . | ● | ● | . | . | • | • | • | • | . |
| System.Security.Cryptography.* | ● | ● | ● | • | ● | • | ● | . | • |  |  | . |  | ● |  | • | . | . |  | • |  |
| Microsoft.VisualBasic.* | ● | ● | 38 | . | ● | . | . | • | • | • | ● | . | . | . | 48 | • | . | . | . | ● | ● |
| System.Runtime.InteropServices.* | ● | • | . | . | . | ● | . | ● | ● |  |  | . | ● | ● |  | ● | . | . |  | ● | ● |
| Microsoft.JScript.* | ● | ● | ● | ● | ● | . | • | ● | . |  |  | • | . | . |  | ● | . | . | • | ● | ● |
| System.Drawing.* | ● | ● | • | . | . | • | . | . | ● | ● |  | . | • | ● | • | ● | . | . | • | • | . |
| System.Runtime.Remoting.* | ● | • | • | ● | ● | • | . | ● | . | • |  | • | 23 | . | • | • | . | • | • | • | ● |
| System.Configuration.* | • | • | ● | • | ● | • | • | • | . | • |  | ● | • | . | • | ● | . | . | . | ● | • |
| System.Diagnostics.* | • | • | • | . | • | ● | • | . | • | • |  | • | . | • | . | • | . | . |  | ● | • |
| System.IO.* | • | ● | ● | . | ● | ● | . | . | ● | ● |  | . | • | ● | • | . | • | . | • | ● | . |
| System.Reflection.* | • | ● | . | • | . | ● | . | . | ● | • |  | . |  | ● |  | ● | . | . | . |  | . |
| System.EnterpriseServices.* | • | . |  | ● | • | . | . | ● | • |  |  | ● | . | . |  | ● | . | . |  | • | • |
| System.CodeDom.* | • | • | ● | . | • | • | ● | . | . |  |  | ● |  | . | . | . | ● | ● | • |  | . |
| System.IdentityModel.* | • | • | • | . | ● | . | ● | . | . |  |  | ● | . | . | . | . | ● | ● | ● | • | • |
| Microsoft.Build.* | • | • | ● | ● | . | . | . | ● | . | • | ● | ● | ● | . | . | . | ● | ● |  |  |  |
| System.Management.* | • | • | • | . | . | . | • | . | . | ● |  | • | ● | . | ● | • | • | • | • |  | . |
| System.Threading.* | • | • | . | . | . | ● | . | . | • | ● | ● | . | . | ● | ● | . | • | • | • |  | . |
| System.Runtime.Serialization.* | • | • | . | • | • | ● | . | ● | . |  |  | ● | • | . | . | ● | . | . | ● | ● | ● |
| System.Security.AccessControl | • | . | • |  | . | • | . |  | ● |  | ● | . |  | ● |  | ● | . | . | • |  | . |
| System.Security.Permissions | • | • | . | ● | . | ● | • | . | ● |  |  | . | ● | ● |  | 86 | . | . |  |  |  |
| System.Runtime.CompilerServices | • | . | . | . | . | ● | ● | . | . |  | ● | ● | . | . | • | ● | . | . | • | ● | . |
| System.Linq.* | • | • | ● | • | . | • | • | • | . |  | ● | • | 23 | . | ● | • | . | . | • | • | • |
| System.AddIn.* | . | . |  | ● | . |  | . | ● | ● |  | ● | . | ● | • | • | ● | . | • |  | ● | ● |
| System.Xaml.* | . | • | • | • | • | . | . | ● | . |  | ● | ● | ● | . | . | . | ● | ● |  | ● | ● |
| System.Messaging.* | . | • | . | • | • | . | . | . | ● | • |  | . | • | ● | • | . | ● | ● |  |  |  |
| Microsoft.Win32.* | . | . | • |  | • | ● | . |  | • | ● |  | . | . | • | ● | • | . | . | ● |  | • |
| System.Security.Policy | . | . | ● | ● | • | • | • | ● | . |  |  | ● | . | . |  | ● | . | . |  | • | . |
| System.Globalization | . | • | ● |  | . | ● | • |  | ● |  |  | . |  | ● |  | . | ● | ● |  |  |  |
| Microsoft.VisualC.* | . | . |  | • | . |  | . | ● |  | ● | 100 | • | ● | . | . |  | 100 | 100 |  | ● | ● |
| System.Transactions.* | . | . | • | . | . | . | . | ● | • | ● |  | • | ● | . | ● | • | . | . |  | 100 | ● |
| System.Security | . | . |  | • | . | ● | . | ● | • |  |  | • | ● | . | . | ● | . | . |  |  |  |
| System.Collections.Generic | . | • |  | ● | . | ● | . | ● | . |  | ● | . | ● | • | ● | . | ● | ● | ● |  | ● |
| System.Runtime.DurableInstancing | . | . | • |  | . | . | ● |  | . |  | ● | . | ● | . | • | . | • | . | ● |  | . |
| System.Collections | . | . |  | ● | . | ● | . | ● | . |  |  | • | • | • |  | . | • | ● |  |  |  |
| System.Text | . | . | • |  | . | ● | ● |  | . |  |  | . |  | ● |  | ● | . | . | ● |  | • |
| System.Deployment.* | . | . | • |  | • | . | ● |  | . | ● |  | • | . | . | ● | . | ● | ● |  |  |  |
| System.Runtime.Caching.* | . | . | • |  | • | . | . | • | . | ● |  | ● | • | . | ● | • | . | . | ● | 100 | ● |
| System.ServiceProcess.* | . | . |  |  | • | . | . |  | ● |  |  | . | • | ● |  | . | ● | ● |  |  |  |
| System.Resources.* | . | . | • | • | • | . | • | ● | . |  |  | ● | • | • |  | • | • | • |  |  |  |
| System.Dynamic | . | . | ● | • | . | . | ● | • |  |  |  | 91 |  | . | • | • | ● | ● | 72 | ● | 70 |
| System.Security.Principal | . | . | • | • | . | • | . | • | ● |  |  | • |  | • |  | • | • | • |  |  |  |
| Microsoft.SqlServer.Server | . | • | • |  | . | . | • | • | • |  |  | . |  | ● |  | ● | . | . |  | 100 | • |
| System.Security.Authentication.* | . | . | • |  | • | . | . |  | ● |  |  | ● | ● | • |  | • | • | • | ● |  | ● |
| System.Collections.Specialized | . | • | • | • | . | ● | • | • | . | ● |  | • | • | • | • |  | 100 | 100 |  |  |  |
| System.Device.Location | . | . |  | • | . |  | • | • | ● |  | ● | • |  | • | ● | • | • | ● |  |  |  |
| System.Text.RegularExpressions | . | . | • |  | . | • | ● |  | . | ● |  | • | . | ● | • |  | 100 | 100 | ● |  | ● |
| Accessibility | . | . |  | • | . | . | . | 60 | ● |  |  | . | • | ● |  |  | 100 | 100 |  | ● | ● |
| System.Collections.Concurrent | . | • |  | ● | . | . | ● | ● |  |  | ● | . | • | ● | • |  | 100 | 100 | ● |  | ● |
| System.Runtime.Versioning | . | . |  |  | . | • | • |  | ● |  |  | • |  | • |  | ● | . | . |  |  |  |
| Microsoft.CSharp.* | . | . |  |  | • | • | • |  | ● |  |  | • | ● | • |  | • | • | • |  |  |  |
| System.Collections.ObjectModel | . | . |  |  | . | • | 100 |  |  |  | 100 | . | . | ● | ● |  | 100 | 100 | ● |  | ● |
| System.Runtime.ConstrainedExecution | . | . |  |  | . | . | . |  | ● |  |  |  |  |  |  | ● | . | . |  |  |  |
| System.Runtime | . | . |  |  | ● | ● | ● |  | • |  |  |  |  | 100 |  | ● | . | . |  |  |  |
| System.Timers | . | . |  |  | . | . | • |  |  | 25 |  | . | ● | . | ● |  | 100 | 100 |  |  |  |
| System.Media | . | . |  |  | . | . | 100 |  |  |  |  | . |  | • | ● | • | • | • |  |  |  |
| System.Runtime.Hosting | . | . |  |  | . | • | 100 |  |  |  |  | ● |  | . |  | • | • | . |  |  |  |
| System.Runtime.ExceptionServices | . | . |  |  | . |  | 100 |  |  |  |  |  |  |  |  | • | • | . |  |  |  |
| System.Numerics | . | • |  |  | . | . |  |  | 100 |  |  | . | • | ● |  |  |  |  |  |  |  |
| 75% | 190 | 1579 | 8 | 7 | 22 | 26 | 80 | 16 | 23 | 5 | 1 | 66 | 8 | 46 | 5 | 51 | 89 | 90 | 4 | 40 | 10 |
| Median | 60 | 543 | 3 | 2 | 17 | 9 | 72 | 6 | 13 | 0 | 0 | 54 | 4 | 34 | 2 | 33 | 67 | 69 | 1 | 0 | 4 |
| 25% | 18 | 175 | 0 | 0 | 11 | 3 | 60 | 0 | 6 | 0 | 0 | 37 | 1 | 19 | 0 | 10 | 46 | 50 | 0 | 0 | 0 |

Table 4.6. Infographics for reuse-related metrics for .NET

We total some measurements *over all .NET namespaces*:

| | | |
|---|---|---|
| # Types | = 12611 | |
| # OO types | = 10103 | |
| # Classes | = 9215 | |
| # Interfaces | = 888 | |
| # Specializable classes | = 5750 | (62.4 % of all classes) |
| # Specializable types | = 6638 | (65.7 % of all OO types) |

In accordance with our methodology we grew this set of metrics, and we used the infographics of Table 4.6 and further views (available in Appendix A.2.2) to develop intuitions about reuse-related characteristics of namespaces. The following classification only uses some of the metrics directly, but the other metrics are useful for understanding and validation.

## 4.3.3 Classification of Frameworks

In the following, we use the reuse-related metrics to define categories for reuse characteristics of frameworks—in fact, namespaces. See Figure 4.8 for the concise definition of the categories. See Table 4.7 for the application of the classification to a few .NET namespaces that serve as representatives in this section. The section is finished with considerations of validation.

### Derivation of the categories

Let us start with 'inter-namespace reuse'. An **application** namespace is characterized by the lack of other namespaces referring to it. That is, no reuse potential is realized for the given namespace within the composite framework. Instead of namespaces with zero referring namespaces, we may also consider namespaces with the most referring namespaces. These are called **core** namespaces for obvious reasons.

As the medians and other percentiles at the bottom of Table 4.6 indicate, inter-namespace usage is very common for .NET. (The appendix of the online version even shows substantial mutual dependencies.) There are these application namespaces. The *System.AddIn.** namespace provides a generic framework for framework plug-ins in the sense of client frameworks on top of .NET. The *Microsoft.VisualC.** namespace supports compilation and code generation for C++. The *System.Device.Location* namespace allows application developers to access the computer's location. The *System.Runtime.ExceptionServices* namespace supports advanced exception handling for applications.

Perhaps the most obvious representative of a core namespace is *System.Collections* as it provides collection types very much like a library for basic datatypes. Starting at the top of Table 4.6, the largest core namespace is *System.ComponentModel.** with its fundamental support for implementing the run-time and design-time behavior of components and controls. The next core namespace is *System.Xml.** with various APIs for XML processing.

Let us consider 'specializability'. We speak of an **open** namespace when the percentage of specializable types is 'exceptional'. We speak of a **closed** namespace

Namespace categories with regard to 'inter-namespace reuse':

- *application* if *#Referring namespaces* $= 0$.
- *core* if *#Referring namespaces* is 'exceptional'.

Namespace categories with regard to 'specializability':

- *open* if *% Specializable types* is 'exceptional'.
- *closed* if *% Sealed classes* is 'exceptional'.
- *incomplete* if *% Orphan types* is 'exceptional'.

Namespace categories with regard to 'class-inheritance trees':

- *branched* if *MAX size class tree* is 'exceptional'.
- *flat* if *MAX size class tree* $= 0$.

Namespace categories with regard to 'intensiveness':

- *interface-intensive* if *% Interface arguments* is 'exceptional'.
- *delegate-intensive* if *% Delegate arguments* is 'exceptional'.

A sub-category for delegate-intensive namespaces:

- *event-based* if *% Delegate types* is 'exceptional'.

Occurrences of 'exceptional' are essentially configurable. We assume though that "$x$ is 'exceptional' for a namespace" proxies for the statement that the metric $x$ for the given namespace is in the $[75, 100)$ percentage interval with regard to the distribution for metric $x$ over all namespaces.

**Figure 4.8.** Definition of (non-mutually exclusive) categories

when the percentage of sealed classes is 'exceptional'. It will be interesting to see whether open namespaces are subject to 'more' specialization in projects than non-open (or even closed) namespaces. In any case, it is helpful to understand which namespaces come wide open and which namespaces limit specialization explicitly. In this context, another category emerges. We speak of an **incomplete** namespace, when the percentage of orphan types is 'exceptional'.

Starting at the top of Table 4.6, the largest open namespace is *System.DirectoryServices.**; it models entities in a network (such as users and printers) and it supports common tasks (such as adding users and setting permissions). The next open namespace is *System.CodeDom.**; it models an abstract syntax of .NET languages. These namespaces provide rich inheritance hierarchies that are left open for specialization by other frameworks or client code. We mention that *System.DirectoryServices.** is not specialized within the .NET Framework itself while *System.CodeDom.** is specialized by several namespaces. Basic knowledge of .NET suggests that CodeDom is specialized by namespaces that host 'CodeDom providers' and regular projects are actually not very likely to contain additional providers.

The largest closed namespace is *System.Data.**; it supports data access and management for diverse sources—relational databases specifically. One may expect this

| Namespace | Application | Core | Open | Closed | Incomplete | Branched | Flat | Interface-intensive | Delegate-intensive | Event-based |
|---|---|---|---|---|---|---|---|---|---|---|
| System.Web.* | | | | | | ✓ | | | ✓ | ✓ |
| System.Data.* | | | | ✓ | | ✓ | | | | |
| System.Activities.* | | | | ✓ | | ✓ | | | ✓ | |
| System.ComponentModel.* | | ✓ | | | ✓ | ✓ | | ✓ | ✓ | ✓ |
| System.Xml.* | | ✓ | | | | | | | | |
| System.DirectoryServices.* | | | ✓ | | | ✓ | | | | |
| System.EnterpriseServices.* | | | | ✓ | | | ✓ | | | |
| System.CodeDom.* | | | ✓ | | | ✓ | | | | |
| System.Linq.* | | | | | | ✓ | | ✓ | ✓ | |
| System.AddIn.* | ✓ | | | ✓ | ✓ | | ✓ | ✓ | | |
| Microsoft.VisualC.* | ✓ | | ✓ | | ✓ | | ✓ | ✓ | | |
| System.Transactions.* | | | | | ✓ | | | ✓ | ✓ | ✓ |
| System.Collections | | ✓ | ✓ | | | | ✓ | | | |
| System.Runtime.Caching.* | | | | | ✓ | | | | ✓ | ✓ |
| System.Device.Location | ✓ | | ✓ | | | | ✓ | | ✓ | |
| System.Runtime.ExceptionServices | ✓ | | | | | | ✓ | | | |

**Table 4.7.** Classification of selected .NET namespaces. Full list see in Appendix A.2.2

namespace to be open because of the need to define a rich provider interface for diverse sources. However, many of the classes for data access and management do not depend on the data source, and hence they are sealed. Also, various providers, such as SQL Server, Oracle, ODBC, and OleDB, are included into the namespace and accordingly sealed. The next closed namespace is *System.Activities.**; it supports abstract syntax and presentation of activities in a workflow sense. One may expect this namespace to be open because of the common design to provide extensibility of syntaxes by inheritance. However, the abstract syntax at hand is effectively constrained to be non-extensible.

Orphan types are clearly very common in .NET; see again Table 4.6.[12] The assumption is here that the orphan types model domain-/application-centric concepts that cannot be implemented in the framework. Let us review those incomplete .NET namespaces with all their interfaces being orphans. The *System.Transactions.** namespace supports transaction and resource management; it is referenced by several other

[12] (Third-party) framework design guidelines for .NET [47] discourage orphan types; a framework designer is supposed to provide implementations (say, concrete classes) for all interfaces and abstract classes. Still orphan types exist in .NET—presumably because corresponding implementations would be illustrative rather than reusable and hence better suited for samples than for inclusion into the framework assemblies.

.NET namespaces—without though implementing any of its interfaces. The *System.Runtime.Caching.** namespace supports caching; this namespace is only used in the *System.Web.** namespace—without though implementing any of its interfaces.

Let us turn to categories related to 'class-inheritance trees'. There are **flat** namespaces without intra-namespace class inheritance. There are **branched** namespaces with 'exceptional' inheritance trees. Flat namespaces may be thought of as providing a facade for (some of) the referenced namespaces in the broad sense of the design pattern of that name. Branched namespaces stand out with a complex object model—complex in terms of tree size.

Starting at the top of Table 4.6, the largest flat namespace is *System.EnterpriseServices.**; it supports component programming with COM+. In particular, .NET objects can be provided with access to resource and transaction management of COM+. One can indeed think of *System.EnterpriseServices.** as a facade. There are many branched namespaces at the top of Table 4.6; the bigger a namespace, the more likely it contains some sizable tree among its forest of classes. We previously encountered a 'small' branched namespace *System.CodeDom.** with its object model for .NET-language abstract syntax.

Let us finally consider what we call 'intensiveness'. An **interface-intensive** namespace makes much use of interfaces for method arguments, thereby supporting reusability in terms of interface polymorphism. (This may be seen as a symptom of interface-oriented programming.) There are also **delegate-intensive** namespaces, which make much use of delegates for method arguments. Basic knowledge of .NET tells us that delegates are used in .NET for two major programming styles. That is, delegates may be used for either functional (OO) programming or event-based systems. These two styles cannot be separated easily—certainly not by means of simple metrics. There is a specific form of an **event-based** namespace that reveals itself through the delegate types that it declares.

A clearcut example of a namespace that is, in fact, both interface- and delegate-intensive is *System.Linq.**; it supports functional (OO) programming specifically for collections based on the *IEnumerable* interface and friends. We note that *System.Linq.** does not declare any delegate type because the fundamental function types are readily provided by the *System* namespace. There are several namespaces with 'exceptional' percentages of both delegate arguments and delegate-type declarations, thereby suggesting themselves as candidates of the aforementioned, specific form of event-based namespaces; see, for example, *System.Web.** right at the top of Table 4.6—the namespace provides web-related functionality and uses an event-based style, for example, to attach handlers to user-interface components.

Event-based programming does not necessarily involve designated delegate types. Standard delegate types for functions or designated interface types may be used as well. For instance, the *System.Device.Location* namespace uses special interface types to process updates to the device's location. Hence, more advanced static analysis would be needed to find evidence of event-based programming in the form of, for example, subscription protocols or state-based behavior. Further, a strict separation between functional (OO) and event-based programming is not universally meaningful. For instance, the use of asynchronous calls is arguably both functional and event-based. This is an issue with classifying the *System.Activities.** namespace, for example.

### Validation of categories

We performed validation to check that the computationally assigned categories (based on Figure 4.8) match with the expectations of domain experts. We discussed each assigned category in a manner that one researcher had to provide the confirmative argument for a category, and another researcher had to confirm—both researchers (in fact, authors) being knowledgable in .NET.

In this process, we decided to focus on search for false positives and neglect search for false negatives on the grounds of the argument that the metrics-based category definitions are designed to find 'striking' true positives only. Nevertheless, we offer an example of a false negative for better understanding. The *System.Data.** namespace should arguably be classified as a delegate-intensive namespace. In fact, the namespace leverages functional (OO) programming in the way that data providers are LINQ-enabled. However, the actual percentage of delegate usage does not meet the threshold of the category's definition.

## 4.3.4 Comparison of Potential and Actual Reuse

We consider the as-implemented usage of a framework. Our main interest is to infer how projects 'typically' use the framework. We define corresponding metrics and screen them for .NET and the corpus of .NET projects of this study.

### Definition of metrics

The metric **% Referenced OO types** denotes the percentage of all OO types of a given namespace (or the entire framework) that are actually referenced (say, reused) in a given project (or the entire corpus). The following metrics are defined 'relative' to the referenced OO types as opposed to all types.

In §4.3.2, we considered specializable types; the corresponding relative metric is **% Specializable types (rel.)**—the percentage of all referenced OO types that are specializable. Likewise, the metric **% Specialized types (rel.)** denotes the percentage of specializable, referenced types that were actually specialized in projects. Finally, the metric **% Late-bound types (rel.)** denotes the percentage of specializable, referenced types that were actually bound late in projects. We say that a framework type is bound late in a project, if there is a method call with the framework type as static receiver type and a project type as runtime receiver type. (Clearly, said project type directly or indirectly specializes said framework type.)

### Measurements for .NET

We summarize measurements for the corpus:

- 44 namespaces (out of 69) are *referenced*.
- 22 namespaces are *specialized*.
- 15 namespaces are *bound late*.
- 925 classes (10.0 % of all classes) are *referenced*.
- 105 interfaces (11.8 % of all interfaces) are *referenced*.
- 173 types (2.6 % of all specializable types) are *specialized*:
  - 107 classes (1.9 % of all specializable classes)
  - 66 interfaces (7.4 % of all interfaces)
    - 30 interfaces are inherited.
    - 66 interfaces are implemented.[13]
- 611 *static receiver types* are exercised.
- 142 types (2.1 % of all specializable types) are *bound late*:[14]
  - 116 classes (2.0 % of all specializable classes)[15]
  - 26 interfaces (2.9 % of all interfaces)

An infographics with details is shown in Table 4.8; the figure in the methodology section was a sketch of this table. We order namespaces again by the number of types and we include only those ever referenced by the corpus.

The middle block of columns displays actual reuse for all combinations of namespace and project while using the following indicators: '–' denotes infeasible reuse (in the sense that the namespace is not available for the framework version of the project); 'blank' denotes no reuse; '⁎' or '∗' denotes less or more referencing (without specialization); '▴' or '▲' denotes less or more specialization (without late binding); '▪' or '■' denotes less or more late binding. Here, 'less or more' refers to below versus above median non-zero percentages of referenced OO types, specialized types (rel.), and late-bound types (rel.). Hence, those cells show whether referencing, specialization, and late binding happen at all, and if so, to what extent (at a coarse-grained level: less versus more).

The columns on the left summarize potential reuse for each namespace in terms of the metrics *# Types* and *% Specializable types* from Table 4.6.

The columns on the right summarize actual reuse in terms of a 'dominator' (i.e., the dominating form of reuse) and the actual reuse metrics defined above. The dominator is determined as follows—without taking into account extent of reuse ('less or more'). If a namespace is not reused by more than half of all projects, then the dominator cell remains empty; see, e.g., *System.CodeDom.∗*. Otherwise, if 'referencing' is more frequent than 'specialization' and 'late binding' combined, then the dominator is '∗'; in the opposite case, the dominator is '▲' or '■'—whatever reuse form is more frequent. For instance, namespace *System* is used with late binding in most projects. Hence, actual reuse is summarized as '■'.

[13] Hence, all .NET interfaces serving as base type in interface inheritance in the corpus are also implemented in the corpus.

[14] Our analysis cannot find all forms of late binding, since we observe late binding based solely on the calls from client code to the framework, while it might also be the case that the framework calls into the client code through callbacks.

[15] The number of types bound late may indeed be greater than the number of specialized types because late binding relates to static receiver types; one project type may have several ancestors in the framework.

| Namespace | # Types | % Specializable types | ActiveRecord | CastleCore | MonoRail | Windsor | Json.NET | log4net | MEF | Moq | NAnt | NHibernate | NUnit | Prism | Rhino.Mocks | Spring.NET | xUnit | SharpZipLib | Lucene.Net | Dominator | % Referenced OO types | % Specializable types (rel.) | % Specialized types (rel.) | % Late-bound types (rel.) |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Framework | | | 3.5 | 4.0 | 3.5 | 4.0 | 4.0 | 2.0 | 4.0 | 4.0 | 2.0 | 3.5 | 3.5 | 4.0 | 3.5 | 2.0 | 2.0 | 2.0 | 2.0 | | | | | |
| System.Web.* | 2327 | • | ▴ | | ▴ | ▴ | | * | | | * | | | | | ▪ | | | | | · | • | ● | • |
| System.Windows.* | ● | • | | | | | | – | | | – | | | ▪ | | – | – | – | – | | · | • | • | ● |
| System.ServiceModel.* | ● | · | | | | | | – | | | – | ▴ | | | | – | – | – | – | | · | · | ● | |
| System.Windows.Forms.* | ● | ● | | | | | | | | | | | ▪ | | | | ▴ | | | | • | • | • | ● |
| System.Data.* | ● | · | * | | * | | * | * | | | | ▴ | | | | ▴ | * | | | | · | · | • | |
| System.ComponentModel.* | ● | • | * | ▪ | ▴ | ▪ | ▪ | | ▴ | * | ▪ | ▴ | ▴ | ▴ | | ▪ | * | | | ■ | • | • | ● | ● |
| System.Xml.* | ● | • | * | ▪ | * | * | ∗ | * | | | ▴ | ∗ | * | | | ▪ | * | | | ∗ | • | ● | • | • |
| System.Net.* | ● | • | | * | * | | | * | | | * | | | | | * | | | * | | • | • | | |
| System | ● | • | ▪ | ▪ | ▪ | ▪ | ▪ | ▲ | ▪ | ▪ | ▪ | ▪ | ▪ | ▪ | ▪ | ▪ | ▪ | ▴ | ▪ | ■ | ● | • | • | ● |
| System.Security.Cryptography.* | ● | • | | | | | | | | | * | | | | | | | ▴ | * | | · | • | • | |
| System.Runtime.InteropServices.* | ● | • | | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | ∗ | · | · | | |
| Microsoft.VisualBasic.* | ● | • | | | | | | | | | * | | | | | * | | | | | · | · | | |
| System.Drawing.* | • | · | | | | | | | | | | | ∗ | | | * | ∗ | | | | • | · | | |
| System.Runtime.Remoting.* | • | • | | * | | * | | * | | | * | * | ▪ | * | ▪ | * | ▴ | | | ∗ | • | • | • | • |
| System.Configuration.* | • | · | ▴ | * | ▴ | ▴ | | ▴ | | | ▴ | ▴ | * | ▪ | | ▴ | ▴ | | * | ▲ | · | · | ● | • |
| System.Diagnostics.* | • | • | * | ∗ | * | * | * | * | * | * | * | * | * | * | * | * | ▴ | | * | ∗ | · | · | • | |
| System.IO.* | • | • | * | ∗ | ▴ | * | * | ▪ | * | * | ▪ | ∗ | ▪ | ▪ | * | ∗ | ∗ | ▪ | ▪ | ∗ | • | ● | ● | ● |
| System.Reflection.* | • | · | ∗ | ∗ | ∗ | ∗ | ∗ | ∗ | ∗ | ∗ | ∗ | ∗ | ∗ | ∗ | ∗ | ∗ | ∗ | ∗ | ∗ | ∗ | ● | · | | |
| System.EnterpriseServices.* | • | · | | | | | | | | | | | | | | ∗ | | | | | · | · | | |
| System.CodeDom.* | • | ● | | | | | | | * | * | ∗ | * | * | * | | | * | | | | ● | ● | | |
| Microsoft.Build.* | • | ● | | | | | | | | | ▴ | | | | | | * | | | | • | · | ● | |
| System.Threading.* | • | • | * | * | * | * | * | ∗ | * | * | * | * | ∗ | * | * | ▴ | ∗ | * | * | ∗ | • | · | • | |
| System.Runtime.Serialization.* | • | · | * | ▲ | * | ▴ | ▴ | ▴ | ▴ | * | ▴ | ▲ | * | * | * | ▲ | * | * | ▴ | ▲ | • | · | 75 | |
| System.Security.Permissions | • | · | | * | | * | * | * | * | * | ∗ | * | * | * | * | ∗ | | | | ∗ | · | | | |
| System.Runtime.CompilerServices | • | · | ∗ | * | ∗ | ∗ | ∗ | * | ∗ | * | ∗ | ∗ | * | * | ∗ | ∗ | * | * | * | ∗ | • | · | | |
| System.Linq.* | • | • | * | * | * | ∗ | * | – | ∗ | ▪ | – | ▪ | | ∗ | * | – | – | – | – | ∗ | ● | · | ● | ● |
| System.Messaging.* | • | ● | | | | | | | | | | | | | | ∗ | | | | | • | 100 | | |
| Microsoft.Win32.* | • | • | | | | | | | | | * | | * | | | * | * | | | | · | · | | |
| System.Security.Policy | • | · | | | | | | | | | * | | ∗ | * | | * | * | | | | · | | | |
| System.Globalization | • | ● | * | * | ∗ | ∗ | ∗ | ∗ | * | * | ∗ | * | * | * | | ∗ | * | * | ∗ | ∗ | • | · | | |
| System.Transactions.* | • | • | | | | | | | | | | ▴ | | | | ∗ | ∗ | | | | ● | • | • | |
| System.Security | · | · | | ∗ | | ∗ | ∗ | * | ∗ | * | * | ∗ | * | | | ∗ | * | * | * | ∗ | ● | · | | |
| System.Collections.Generic | · | ● | ■ | ■ | ∗ | ■ | ■ | * | ■ | ▲ | * | ■ | ■ | ■ | ■ | ▲ | ■ | | ■ | ■ | ● | ● | ● | ● |
| System.Text | · | · | * | * | * | * | * | * | * | * | ∗ | * | * | * | * | ∗ | * | * | * | ∗ | · | · | | |
| System.Collections | · | ● | ■ | ▲ | ■ | ▲ | ▲ | ■ | ■ | ▲ | ■ | ■ | ■ | ▲ | ■ | ■ | ■ | ▲ | ■ | ■ | ● | • | ● | ● |
| System.ServiceProcess.* | · | ● | | | | | | | | | ∗ | | | | | | | | | | · | 100 | | |
| System.Resources.* | · | • | | ∗ | ∗ | | * | | * | ∗ | * | | * | ∗ | | ■ | * | | ∗ | ∗ | • | • | • | ● |
| System.Security.Principal | · | • | | | ∗ | | | ∗ | | | | | * | | | | ∗ | | | | ● | 100 | | |
| System.Collections.Specialized | · | 100 | | ▲ | ■ | ■ | | ∗ | | | ■ | ∗ | ∗ | ▲ | ∗ | ■ | | | ∗ | ■ | 86 | 100 | ● | ● |
| System.Text.RegularExpressions | · | 100 | ∗ | | ∗ | ∗ | ∗ | ∗ | | ∗ | ∗ | ∗ | ∗ | | ∗ | ∗ | ∗ | ∗ | ∗ | ∗ | ● | 100 | | |
| System.Runtime.Versioning | · | · | | ∗ | | ∗ | ∗ | | ∗ | ∗ | | | | ∗ | | | | | | | • | | | |
| System.Collections.ObjectModel | · | 100 | | ■ | ■ | ■ | ■ | | ■ | ∗ | | ■ | | ■ | ∗ | ∗ | ∗ | | ■ | ■ | ● | 100 | ● | 50 |
| Microsoft.CSharp.* | · | • | | | | | | | | | ∗ | ∗ | ∗ | | | | | | | | · | 100 | | |
| System.Timers | · | 100 | | | | | | | | | | | ∗ | | | | ∗ | | | | ● | 100 | | |
| # Referenced types | | | 137 | 301 | 245 | 229 | 277 | 229 | 201 | 174 | 375 | 374 | 437 | 213 | 135 | 604 | 308 | 113 | 193 | | | | | |
| # Specialized types | | | 16 | 39 | 28 | 26 | 27 | 20 | 18 | 13 | 26 | 39 | 31 | 29 | 10 | 73 | 19 | 11 | 26 | | | | | |
| # Late bound types | | | 6 | 7 | 9 | 10 | 11 | 3 | 6 | 4 | 10 | 12 | 20 | 11 | 8 | 22 | 5 | 2 | 8 | | | | | |
| 75 % | 235 | 89 | | | | | | | | | | | | | | | | | | | 33 | 92 | 33 | 8 |
| Median | 80 | 73 | | | | | | | | | | | | | | | | | | | 20 | 75 | 6 | 0 |
| 25 % | 36 | 54 | | | | | | | | | | | | | | | | | | | 12 | 50 | 0 | 0 |

**Table 4.8. Infographics for comparing potential and actual reuse for .NET**

**Discussion**

The corpus misses several .NET namespaces totally—including *all* application namespaces (see Table 4.7) and various namespaces related to user interfaces—the latter most likely due to our methodology; see §4.3.1.

The online version determines correlations between various metrics. We state one interesting correlation here: the percentage of referenced OO types is inversely correlated with the size of the namespace (in terms of the number of types). Hence, it may be possible to identify an 'essential core' for each of the largest namespaces.

Let us study the metrics by reviewing all those namespaces that are referenced but not specialized by the corpus. There are 21 namespaces like this and they are all specializable, in principle. Nine of these namespaces are in the upper half of the distribution for % Specializable types. (See, for example, namespaces *System.Globalization* and *System.Text.RegularExpressions* with 'exceptional' specializability.) The referenced OO types are only slightly less specializable. That is, eight namespaces are in the upper half of the distribution for % Specializable types *(rel.)*. Thus, low (high resp.) specialization is not predicted by low (high resp.) specializability in any obvious sense.

Most namespaces are actually referenced by enough projects to get assigned an actual reuse summary in the form of a dominator. This suggests that the projects of the corpus indeed share a 'profile' in an informal sense.

Let us compare potential reuse in terms of specializability with actual reuse in terms of the dominator. There are eight namespaces with dominator '▲' or '■'. Half of these namespaces contribute to the *System.Collections.** hierarchy and the associated specializability is 'exceptional'. However, specializability is 'non-exceptional' for the remaining cases; specializability is, in fact, in the percentage interval (0,25) for two cases; see namespaces *System.Configuration.** and *System.Runtime.Serialization.**. This observation further confirms that high specialization is not predicted by high specializability in any obvious sense.

## 4.4 Multi-dimensional Exploration

We have developed and applied various API usage analyses. In this section, we discuss who and how can benefit from them.

We identify abstract *exploration insights* as they are expected by API developers and project developers with regard to their overall intention to understand API usage. These expected insights rely on multiple dimensions of exploration, e.g., hierarchical organization of scopes and project- versus API-centric perspectives. Existing methods such as code completion and searching API documentation do not serve these insights.

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

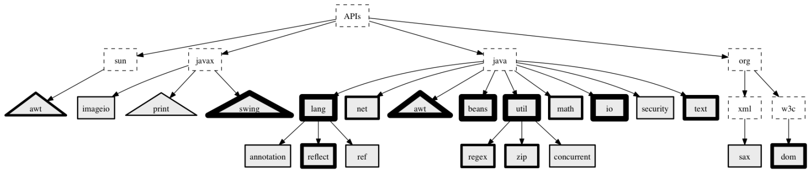
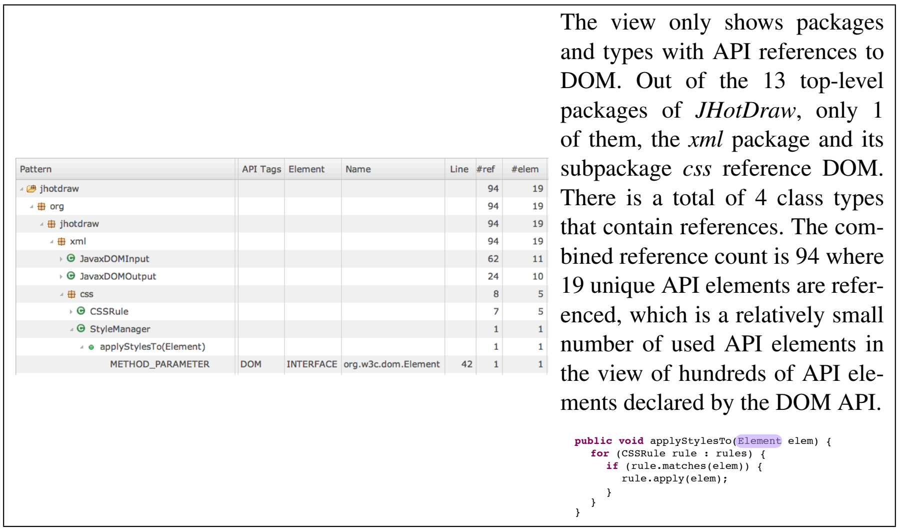


**Figure 4.9.** API usage in *JHotDraw* with scaling applied to numbers of API references



| 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);
      }
   }
}
```

**Figure 4.10.** The slice of *JHotDraw* with DOM usage

and records knowledge about APIs. Preparation of the QUAATLAS is described in detail in Chapter 6.

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 open-source, IDE-like, Web-enabled tool so that we also provide *tool support for exploration* that can be used by others for exploration experiments.

### 4.4.1 An Exploration Story

Let us sketch a simple yet likely story that an API users (i.e., project developers) may find themselves in. *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 heavily 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.

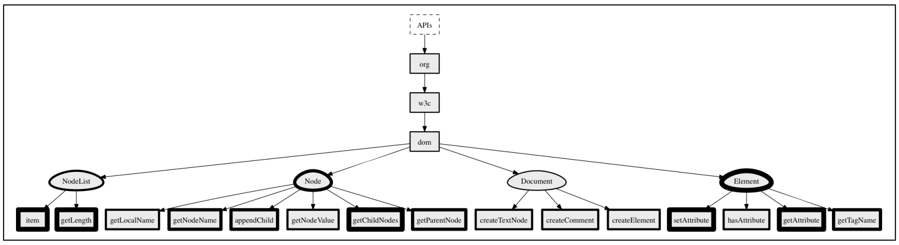


**Figure 4.11.** Minuscule view for *DOM* usage in *JHotDraw*: with leaves for methods, eggs for types, and the remaining nodes for packages.

Fig. 4.9 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. 4.9, two XML APIs show up: *DOM* and *SAX*. *Joanna* begins with an exploration of *DOM* usage. Fig. 4.11 summarizes *DOM* usage in *JHotDraw* as analyzed with EXAPUS. Encouragingly, *DOM*'s footprint in *JHotDraw* only covers a few types and methods.

A logical option for continuation of exploration is to examine the distribution of API usage across *JHotDraw*. In this manner, *Joanna* gets a sense of locality of API usage. The corresponding view is shown in Fig. 4.10 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.

## 4.4.2 Basic Concepts

We set up the basic concepts of this effort: APIs, API usage, and API-usage metrics. We also augment the basic notion of API with extra dimensions of abstraction—API domains and API facets—which are helpful in raising 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 effort 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 subpackages of *java.util*, but they are preferably considered separate APIs. In fact, the *java.util* API deserves further breakdown, giving rise to the notion of *sub-API* because the package serves de facto unrelated purposes, notably Java's collections and Java's event system, which can be quantified as subsets of the types in *java.util*. (This is not an uncommon situation.)

Clearly, APIs may exist in different versions. If these are major versions (e.g., JUnit 3 and 4), then they may be treated effectively as different APIs. In the case of minor versions (assuming qualified names of API elements have remained stable), they may be treated as the same API.

### *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 effort, 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*.

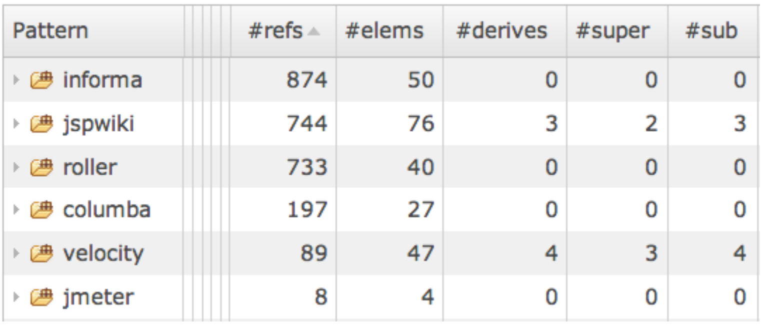
API domains are helpful in reporting API usage and quantifying API usage of interest in more abstract terms than the names of individual APIs, as will be illustrated in §4.4.3.

### *API facets*

An API may contain dozens or hundreds of types each of which has many method members in turn. Some APIs use subpackages to organize such API complexity, but those subpackages 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.

Accordingly, we propose leveraging a notion of API facets. Each API enjoys a manageable number of facets. In general, we may use arbitrary program analyses to attest use of a facet. We limit ourselves to a simple form of facets, which can be attested on the grounds of specific types or methods being used. Except for the notion of API usage patterns, we are not aware of any general, widely adopted attempt to break down APIs into facets, but the idea appears to merit further research. We have begun identifying API facets and tagging APIs appropriately. As an illustration, we briefly characterize 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.



| 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 |

**Figure 4.12.** *JDOM*'s API Dispersion in QUAATLAS (project-centric table).

**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.

We may also designate a facet to 'Nontrivial API' usage when it involves advanced 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 §4.4.3.

## 4.4.3 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*.

### 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.

### The API Dispersion insight

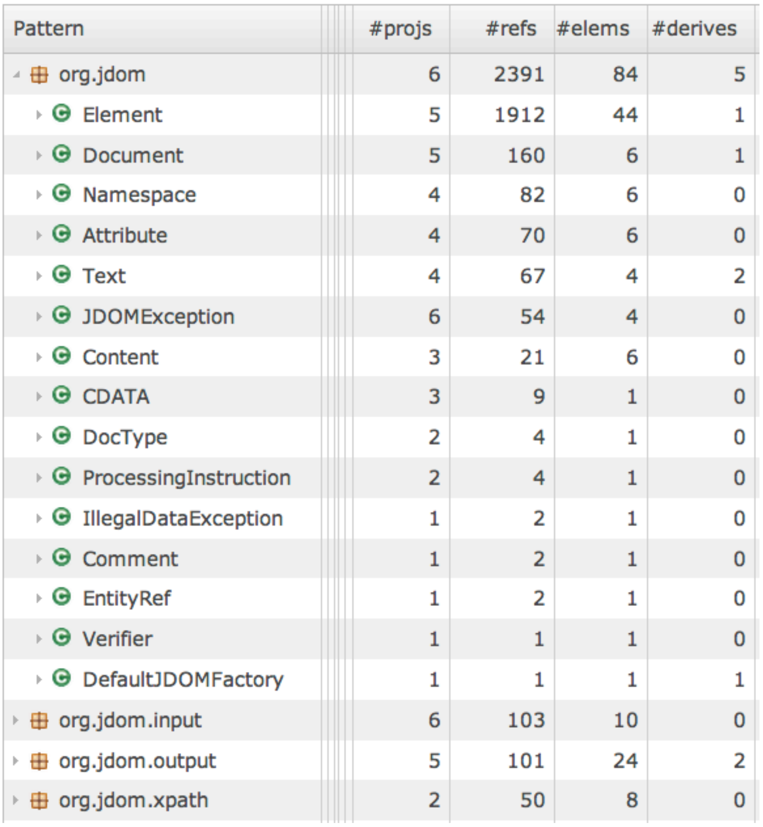
**Intent** – Understand an API's dispersion in a corpus by comparing API usage across the projects in the corpus.
**Stakeholder** – API developer.
**API usage** – One API.
**View** – The listing of projects with associated API-usage metrics for quantitative comparison and API facets for qualitative comparison.
**Illustration** – Fig. 4.12 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 [22, 20].

### 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. 4.10 in §4.4.1. This view was suitable for efficient exploration of project scopes that directly depend DOM.
**Intelligence** – The insight may help a developer to decide on the feasibility of an API migration, as we discussed in §4.4.1.

### 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.
**Illustration** – Remember the tree-based representation of the API footprint for *JHotDraw* as shown in Fig. 4.11 in §4.4.1. In a similar manner, while using a table-based representation, Fig. 4.13 summarizes *JDOM* usage across QUAATLAS. All *JDOM* packages are listed. The core package is heavily used and thus the listing is further



| 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 |

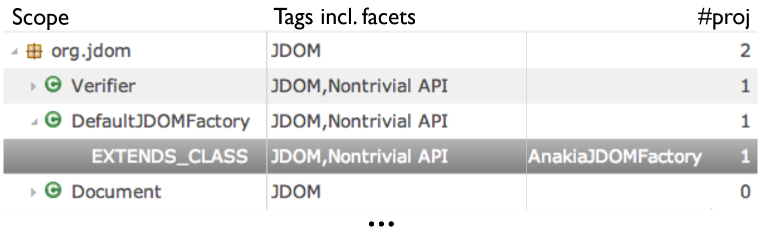
**Figure 4.13.** *JDOM*'s API Footprint in QUAATLAS (api-centric table).

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 [4, 3], 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 [22, 20].

### The **Sub-API Footprint** insight

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



| 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 |

...

**Figure 4.14.** ‘Non-trivial API’ usage for package *org.jdom* in QUAATLAS.

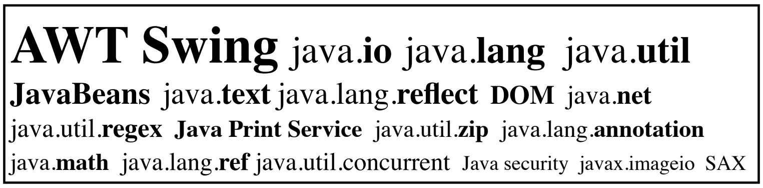


**Figure 4.15.** The API Cocktail of *JHotDraw* (cloud of API tags).

**Stakeholder** – API developer and, possibly, project developer.
**API usage** – One API.
**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. 4.14 illustrates ‘Non-trivial API’ usage for *JDOM*’s core package. The selection is concerned with a project type which extends the API type *Default-JDOMFactory* 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 [20]. 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 [172, 58].

### The API Cocktail insight

**Intent** – Understand what APIs are used together in larger project scopes.
**Stakeholder** – Project developer.
**API usage** – All APIs.

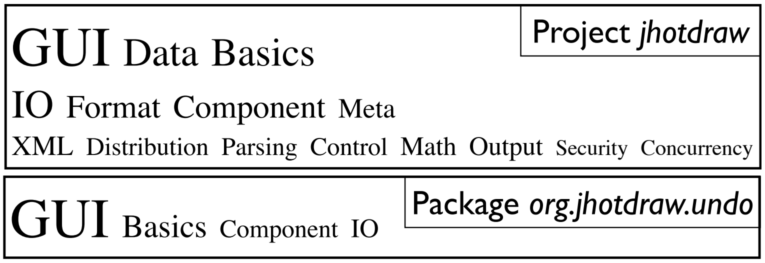


**Figure 4.16.** Cocktail of domains for *JHotDraw*.

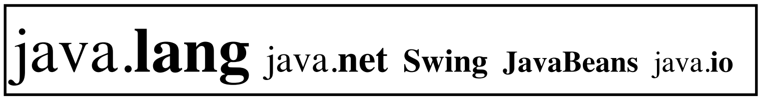


**Figure 4.17.** API Coupling for *JHotDraw*'s interface *org.jhotdraw.app.View*.

**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. 4.9 in §4.4.1. The same cocktail of 20 APIs is shown as a tag cloud in Fig. 4.15. 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". Chances are that API usage may cause some "software or API asbestos" [114]. 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.

## 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. 4.16 for illustration. It shows API domains for all of *JHotDraw* and also for its *undo* package. Thus, it presents the API cocktails of Fig. 4.15 in a more abstract manner.

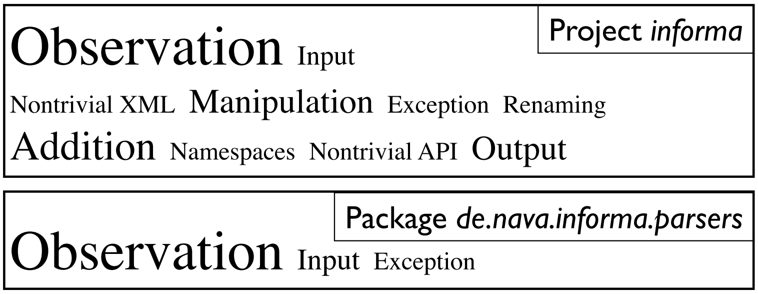

Project *informa*
Observation Input
Nontrivial XML Manipulation Exception Renaming
Addition Namespaces Nontrivial API Output
Package *de.nava.informa.parsers*
Observation Input Exception

**Figure 4.18.** *JDOM*'s API Profile in the *informa* project (cloud of facet tags).

### 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 §4.4.3 except APIs or domains are listed for smaller project scopes.
**Illustration** – Fig. 4.17 shows API Coupling for the interface *org.jhotdraw.app.View* from the *JHotDraw*'s *app* package[16]. 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 [60, 149] that is worth addressing. Alternatively, a dissection, as performed for the illustrative example, may help in understanding the design and reasonable API dependencies.

### The API Profile insight

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

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

**Stakeholder** – Project developer and, possibly, API developer.
**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. 4.18 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" [114], as discussed in §4.4.3. For instance, the *JDOM* facets 'Non-trivial API' and 'Non-trivial XML' and to a lesser extent also 'Namespaces' proxy for development challenges or idiosyncracies. 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. 4.18. Such a facet-based approach to understanding API-usage scenarios complements existing more code pattern-based approaches [98, 199]. API profiles also provide feedback to API developers with regard to 'usage in the wild', thereby guiding API evolution or documentation.

### 4.4.4 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.[17]

#### 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)]
```

[17] 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.

## 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, subpackages, 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. 4.11 shows only API elements that are actually used to summarize an API foot print concisely. In contrast, Fig. 4.14 shows all API types to better understand what API types possibly could exercise the 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 §4.4.1 and §4.4.3. 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
```

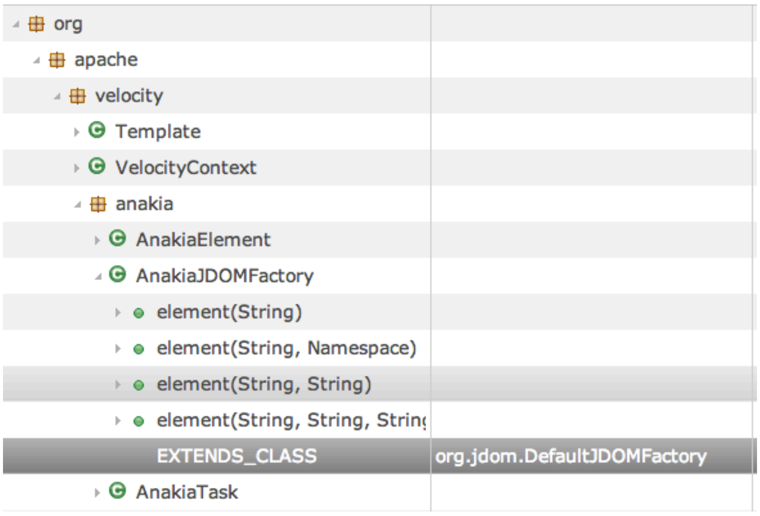
For instance, the #ref metric can be configured to only count references to API types as opposed to other API elements; the #derive metric can be configured to only count class-to-class extension as opposed to other forms of derivation.

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.

## 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. 4.19 shows a project-centric view which is fundamentally dual to the API-centric view of Fig. 4.14 in that the selected outgoing reference corresponds to the originally incoming reference selected in Fig. 4.14. 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



| | |
|---|---|
| org | |
| apache | |
| velocity | |
| Template | |
| VelocityContext | |
| anakia | |
| AnakiaElement | |
| AnakiaJDOMFactory | |
| element(String) | |
| element(String, Namespace) | |
| element(String, String) | |
| element(String, String, Strin | |
| EXTENDS_CLASS | org.jdom.DefaultJDOMFactory |
| AnakiaTask | |

**Figure 4.19.** The dual view for Fig. 4.14 (project-centric table).

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 using basic means of package exploration are available.

### 4.4.5 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 Section 3.2.2. 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 (cf. Section 4.4.4) 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. 4.9) and tables (e.g., Fig. 4.10). 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.

## 4.5 Threats to Validity

There are several limitations to the performed API usage analyses.

### External validity

We have analyzed a restricted set of projects. It means, the results of the studies might not be generalizable, though we consider the used corpora to be representative.

In the study of Java APIs, we restricted ourselves in at least three ways, namely, the source of picking the projects (SourceForge only), the version control system (SVN only) and the build tool (Apache Ant only). These choices were concessions to the primary goal of the present research milestone: to prove the feasibility of large-scale, automated, resolved AST-based API-usage analysis.

In the study of .NET framework, on the contrary, we selected the projects of presumably the best quality from several different online repositories. The requirement of sufficiently large test suits was induced by the chosen style of analysis—namely, dynamical analysis.

### Internal validity

In both studies, we use homegrown tools, which are though tested and manually validated during the research, are still a subject to possible technical errors.

In the case of .NET, there are also more subtle threats, due to the modelunderlying our research. First, while investigating potential and actual .NETreuse, we focus on type specialization—even though frameworks might be alsoconfigured via attributes (i.e., annotations) or XML files. This applies to a number of .NET namespaces. Second, we observe late binding based solely on the calls from client code to the framework, while it might also be the case thatthe framework calls into the client code through callbacks. Further, the analysis of late binding relies on the runtime data gathered from the testsuite execution. Coverage of method-call sites is incomplete; the tests do not cover 38.96 % of the method-call sites in the projects of the study.

In the study of Java APIs, we rely mostly on the results gained from the AST-based fact extractor gathered through a default build. This means, for example, that we miss sources that were not build in this manner. We also miss projects whose builds fail (and hence fact extraction is not completed).

## 4.6 Related Work

We identify several categories of related work.

### Analysis of Open-source Repositories

Our work relates to other work on analyzing open-source repositories. For instance, there has been work on the analysis of *download data*, *evolution data*, *metadata* (e.g., development status), or *user data* (e.g., the number of active developers), and simple *metrics* (e.g., LOC) for open source projects (e.g., on SourceForge) [95, 126, 83, 192]. In this context, we also refer to the project FLOSSmole `http://flossmole.org/` and the project FOSSology [77]. Usually, such large-scale efforts do not involve dynamic analysis, which we use in our studies. For instance, in [82], the adoption of design patterns is studied in open-source software, but documentation (commit messages) as opposed to parse trees are analyzed. In [125], a very large-scale code clone analysis is performed.

### API-usage Analysis

There are efforts that we would like to collectively label with *API-usage analysis*. The kinds of analysis in such related work are different from those considered in the present effort.

One important direction is concerned with the analysis of *API usage patterns*. In [137], data mining is used to determine API reuse patterns, and more specifically classes and methods that are typically used in *combination*. We also refer to [130] for a related approach. In [102], data mining is used, too, to determine frequently appearing ordered sets of function-call usages, taking into account their proximal control constructs (e.g., if-statements). In [196], data mining is used, too, to compute lists of frequent API usage patterns based on results returned from code-search engines; the underlying code does not need to be compilable. In [122], a machine learning-based approach is used to infer protocol specifications for API; see [191] for a related approach. In [10], inter-procedural, control-flow-sensitive static traces for C programs are used to mine API-usage patterns as partial orders from source code.

The closest related work to the study of Java APIs is [180]: API hotspots and coldspots (say, frequently or rarely used classes and methods) are determined. This work is again based on the analysis of data obtained from a code-search engine. Such frequency analysis can be compared to our efforts on coverage analysis (c.f., Sect. 4.2.3)—except that we are using resolved ASTs. Also, we are specifically interested in the large-scale, cumulative coverage. Further, we are interested in interpreting analysis results in terms of API characteristics, and with a view on API migration.

The closest related work to the study of .NET framework is [35, 158] insofar as alignment of static and dynamic receiver types is concerned. In particular, the work of [35] deals with the dynamic measurement of polymorphism in Java and interprets it from a reuse-oriented point of view. Bytecode is instrumented and runtime receiver types are determined by accessing the virtual machine's stack—similar to our approach. This work is not focused though on reuse of a composite framework.

## Type Specialization

In both studies (in the case of Java APIs we have identified the direction, and in the study of .NET framework we have advanced it), we pay attention to type specialization, including class and interface inheritance, interface implementation, overriding. Related work that studies related metrics does so without the objective of summarizing reuse characteristics at a high of level of abstraction. The work of [38] studies structural metrics of Java bytecode; some reuse-related measurements are covered, too, e.g., the number of types that inherit from external framework types, or the most implemented external interfaces. The work of [177, 176] focuses on metrics for inheritance and overriding for Java, and it shows, for example, that programmers extend user-defined types more often than external library or framework types. In those works, depth of inheritance trees is considered relevant whereas our metrics-based approach favored size of inheritance trees since we are interested in the number of types participating in specialization. The work of [162] analyzes instantiations of frameworks (Eclipse UI, JHotDraw, Struts), though for a purpose of detecting usage changes in the course of framework evolution. None of the aforementioned efforts involve dynamic analysis.

## Software Metrics

We define and apply metrics for exploring reuse characteristics and the alignment between potential and actual reuse. Elsewhere, metrics are typically used to understand maintainability [48] or quality of the code and design [131, 182, 185]. There is also a trend to analyze the distribution characteristics for metrics and the correlation between different metrics [39, 25]. In the context of OO programming, work on metrics typically focuses on Java; the work of [123] targets .NET with a few basic metrics without focus on reuse.

## Exploration of Projects

There are several conceptual styles of project comprehension. An example of interactive, human-involving effort can be found in work of Brühlmann et al. [33], where experts annotate project parts to capture human knowledge. They further use the emerged meta-model to analyze features, architecture, and design flaws of the project.

Query-driven comprehension can proceed through user-defined queries that identify code of interest, as in the work of Mens and Kellens [135] or De Roover et al. [53], where a comprehensive tool suite facilitates defining and exploring query results. Alwis and Murphy in their work [51] 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 [121]. Visualizations can be more involved, as in the work of Wettel et al. [193], where a a city metaphor is used to represent a 3D structure of projects based on the value of metrics.

Our tool, EXAPUS, 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 the typical questions like identified by Alwis and Murphy [51]. Tag clouds, tables, and trees accompanied by metrics provide basic and familiar visual aid in exploration.

## Exploration of APIs

### Measuring 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 [138] or to identify and predict API usage trends over time [139]. 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 [91]. Eisenberg et al. use font scaling w.r.t. popularity of API elements to help navigate through its structure [58, 59]. Ma et al. investigate coverage of Java Standard API to identify which parts are ignored by the API users [128]. In our work, we suggest more advanced metrics indicating API usage; their distribution is integrated in the table and graph views of our tool, providing sorting and scaling.

### Understanding usage

Robillard and DeLine discovered in their field study on API learning obstacles that API users prefer to learn from patterns of related calls rather than illustrations of individual methods [156]. And, indeed, many existing efforts are exercising information about API usage to help developers use APIs. E.g., Nasehi and Maurer show that API units tests can be used as usage examples [146]. Zhong et al. cluster API calls and mine patterns to recommend useful code snippets to API users [199]. Bruch et al. develop intelligent code completion that narrows down the possible suggestions to those API elements that are actually relevant [32]. Our tool, EXAPUS, 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 possible exploration activities to perform.

## 4.7 Conclusion

We have performed an exploratory study of API usage, assessing the aspects of API usage in two different ways: from the point of view of a software language and from the point of view of reuse characteristics.

In the study on Java APIs, we have demonstrated a scalable approach to AST-based API-usage analysis for a large-scale corpus of open-source projects. Our implementation allows us to answer questions about usage of many well-known Java APIs. We have demonstrated this capability, for example, with specific aspects of XML programming APIs and generic aspects of framework-like API usage.

In the study of .NET framework, we presented a new approach to understanding reuse characteristics of composite frameworks. We applied the approach in an empirical study to .NET and a suitable corpus of .NET projects. The reuse characteristics include metrics of potential reuse (such as the percentage of specializable types), categories related to reuse (such as open or closed namespaces), and metrics of actual reuse (such as the percentage of specialized types). These metrics and the classification add up to what we call a framework profile. Infographics can be used to provide different views on framework profiles.

We summarized the developed intuitions into a catalogue of exploratory activities. We described these insights along with the abstract model of API views, identifying the possible exploratory scenarios and involved stakeholders. We have built the tool, EXAPUS, enabling the mentioned explorations.

# Part III

# Corpus Engineering

# 5 Literature Survey of Empirical Software Engineering

In this chapter, we describe literature surveys that we carry out in order to understand the existing usage of empirical evidence in Software Engineering. We collect and analyze published papers, extracting signs and characteristics of used empirical evidence. We discover that more than 80% of papers in Software Engineering research use software projects as empirical evidence. The discovered existing demand in the contemporary research motivates our subsequent effort (presented in the next chapter) to provide matching supply in our area of expertise.

## Road-map of the chapter

- Section 5.1 provides brief motivation for the study.
- Section 5.2 describes the two preceding pilot surveys.
- Section 5.3 contains methodology and results of the main survey.
- Section 5.4 discusses threats to validity for this empirical study.
- Section 5.5 discusses related work.
- Section 5.6 concludes the chapter.

## Reproducibility

We provide additional data (lists of used papers, results of the coding, etc.) on the supplementary websites[1,2].

## Related publications

Research presented in this chapter underwent the peer-reviewing procedure and was published in the proceedings of International Conference on Software Language Engineering in 2011 [2]. The main part of the chapter (the final survey) is the work under submission [7].

[1] `http://toknow.sourceforge.net/`
[2] `http://softlang.uni-koblenz.de/empsurvey/`

## 5.1 Introduction

Empirical research is usually perceived as taking one of the established forms with well-defined protocol of the study and applied techniques: controlled and quasi-experiments, exploratory and confirmatory case studies, survey, ethnography, and action research [167, 170]. In a broader sense that we consider in our thesis, empirical research includes any research based on collected evidence (quoted from [167], emphasis ours): "Empirical research seeks to explore, describe, predict, and explain natural, social, or cognitive phenomena by using evidence based on observation or experience. It involves obtaining and interpreting evidence by, e.g., experimentation, systematic observation, interviews or surveys, or *by the careful examination of documents or artifacts*."

And, indeed, Software Engineering research often needs corpora (sets of projects/systems) to test a hypothesis, to validate a tool, or to illustrate an approach. Software corpora have been, for instance, used to evaluate tools for design pattern detection [65], API method recommendation [87], and refactoring automation [141]—just to mention a few areas and papers.

In general, since Software Engineering is a practical area, it is logical to expect that most of the SE research is evidence-based, i.e., empirical de facto, and in the present effort, we submit to show that. We believe that a bottom-up approach of defining forms of research as well as discovering methodology by observing existing research complements the prominent top-down approach, when a methodology is derived from theoretical considerations or by borrowing from other sciences (medicine, sociology, psychology).

To this end, we are interested in harvesting information from the published papers in the Software Engineering area, driven by the general question

*What is the nature of empirical evidence used in Software Engineering?*

Our intention is to detect and assess the existing demand in corpora in the contemporary research. We refine our question through two pilots studies to the following questions:

I How often do Software Engineering papers use *corpora*—collections of empirical evidence?
II What is the nature and characteristics of the used corpora?
III Does common contents occur in the used corpora?

After a literature survey of the contemporary Software Engineering research, we provide answers to these questions by developing and applying a coding scheme to the collected papers.

## 5.2 Pilot Studies

In this section, we describe the two pilot studies which helped us to explore available methods of literature surveys as well as the topic of our interest, the usage of empirical evidence in Software Engineering. (For the brief account of considered methods, their suitability for our research, and evolution of our understanding of the topic, see Part I, Prerequisites, Section 2.5.)

### 5.2.1 Survey on Empirical Language Analysis

Historically, we were interested in understanding usage of empirical evidence in Software Language Engineering only. For that, we restricted ourselves to the area of empirical language analysis and used a corresponding coding scheme à la content analysis [117]. In this subsection, we introduce the paper collection, the research questions of the study, and its results. The study is reported in the same manner as it was originally reported: unaware of the follow-up studies and development of our problem understanding.

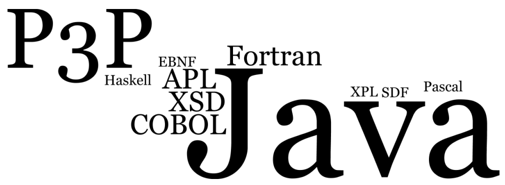


**Figure 5.1.** Tag cloud for the language distribution for the underlying paper collection.

#### Paper collection

We have started with a hand-picked set of papers that we were aware of and considered to belong to Software Language Engineering. Following their appropriate references, we have accumulated 52 papers on empirical analysis of software languages. As an illustration, Fig. 5.1 shows the language distribution for the full collection. For reasons of maturity of metadata, we have eventually used a *selective collection* of 17 papers.

#### Research questions

- Each paper in the collection involves a corpus of a chosen software language. What are the characteristics of those corpora?
- Each empirical analysis can be expected to serve some objective. What are those objectives for the collection of papers?
- Each empirical analysis can be expected to leverage some actual (typically automated) analyses on the corpus. What are those analyses for the collection of papers?

## Terminology

In this first pilot study, we were exploring the terminology and used the following definitions. The term (software) *corpus* refers to a collection of *items* that are expressed in the language at hand. (These items may be valid or invalid elements of the language in a formal sense.) Items originate from possibly several *sources*. The term *source* refers to different kinds of physical or virtual sources that contribute items. For instance, a corpus may leverage an open-source repository as a source to make available, or to retrieve the items—based on an appropriate search strategy. A paper may provide a *corpus description* that identifies sources and explains the derivation of the actual corpus (say, item set) from the sources.

## Corpus characteristics

Fig. 5.2 provides metadata that we inferred for the corpora of the selective paper collection.[3] We capture the following characteristics of the software corpora: the software language of the corpus, numbers of sources and items (with a suitable unit), the online *accessibility* of the sources (on a scale of *none*, *partial*, and *full*), and the *reproducibility* of the corpus (on a scale of *none*, *approximate*, *precise*, and *trivial*). We say that reproducibility is *trivial*, if the corpus is available online—in one piece; reproducibility is *precise*, if the sources and the corpus description suffice to reproduce the corpus precisely by essentially executing the corpus description. Otherwise, we apply a judgment call, and use the tags *approximate* or *none*. For instance, the inability to reproduce a ranking list of a past web search *may* be compensated for by a new web search, and hence, reproducibility can be retained at an approximate level. In future work, we would like to go beyond the characteristics that we have sketched here.

## Objectives of the papers

Based on our analysis of the paper collection, we propose the following list of objectives for empirical language analysis; see Fig. 5.3 for corresponding metadata for the selective paper collection.

### *Language adoption*

The objective is to determine whether the language is used, and with what frequency. Typically, some scope applies. For instance, we may limit the scope geographically, or on the time-line.

### *Language habits*

The objective is to understand the usage of the language in terms of syntactically or semantically defined terms. For instance, we may study the coverage of the language's diverse constructs, or any other, well-defined metrics for that matter. This objective may be addressed with substantial measurements and statistical analysis.

[3] A cell with content "?" means that the relevant data could not be determined.

| | language | sources | items | unit | accessibility | reproducibility |
|---|---|---|---|---|---|---|
| [16] | SDF | 8 | 27 | grammars | partial | approximate |
| [24] | Java | 17 | 56 | projects | full | precise |
| [34] | COBOL | 1 | 50 | programs | none | none |
| [37] | Java | 1 | 1132 | JAR files | full | approximate |
| [40] | Pascal | 1 | 264 | programs | none | none |
| [44] | P3P | 3 | ? | policies | full | approximate |
| [71] | Java | 4 | 14 | projects | full | precise |
| [81] | Haskell | 1 | 68000 | compilations | none | approximate |
| [82] | Java | 1 | 988 | projects | full | approximate |
| [85] | Java | ? | 9 | packages | full | approximate |
| [108] | Java | 2 | 2 | programs | full | approximate |
| [115] | Fortran | 7 | 440 | programs | none | none |
| [120] | XSD | 2 | 63 | schemas | partial | none |
| [1] | P3P | 1 | 3227 | policies | full | trivial |
| [154] | P3P | 1 | 2287 | policies | full | approximate |
| [160] | APL | 6 | 32 | workspaces | none | none |
| [187] | XSD | 9 | 9 | schemas | full | approximate |

**Figure 5.2.** Corpus characteristics for selective paper collection

### *Language taming*

The objective is to impose extra structure on language usage so that habits can be categorized in new ways. For instance, we may equip the language with patterns or metrics that are newly introduced or adopted from other languages. In some cases, the empirical effort towards addressing the objective of language taming may also qualify as effort that attests to the objective of language habits.

### *User feedback*

The objective is to compile data of any kind that helps the language user to better understand or improve programs. For instance, we may carry out an analysis to support the proposal of a new pattern that should help with using the language more effectively. This objective could be seen as a more specific kind of language taming.

### *Language evolution*

The objective is to gather input for design work on the next version of the language. For instance, we may try to detect indications for missing constructs. Or detect obsolete language features that could be deprecated in the future versions.

### *User behavior*

The objective is to understand the usage of the language and its tools in a way that involves users or user experiences directly—thereby going beyond the narrow notion of corpus consisting only of “programs”. For instance, we may analyze instances of compiler invocations with regard to problems of getting programs to compile eventually.

***Implementor feedback***

The objective is to understand parameters of language usage that help language implementors to improve compilers and other language tools. For instance, we may carry out an analysis to suggest compiler optimizations.

| | [16] | [24] | [34] | [37] | [40] | [44] | [71] | [81] | [82] | [85] | [108] | [115] | [120] | [1] | [154] | [160] | [187] |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| language adoption | | | | | | • | | | • | | | | | | • | | |
| language habits | | • | | • | • | • | | | • | | | | | • | • | • | |
| language taming | • | | | | | | • | | | • | • | | • | | | | • |
| user feedback | | | | | | | | | | • | | | | | | | |
| language evolution | | | | • | | | • | | | | | | | | | | |
| user behavior | | | | | | | | • | | | | | | | | | |
| implementor feedback | | | • | • | • | | | | | | | • | | | | | |

**Figure 5.3.** Objectives of the selected publications

Without going into detail here, the available data caters for various observations. For instance, we realize that research on language adoption is generally not exercised for programming languages. It appears that online communications but not scientific publications are concerned with such adoption.[4,5,6]

## Analyses of the papers

Based on our (preliminary) analysis of the paper collection, we have come up with a simple hierarchical classification of (typically automated) analyses that are leveraged in the empirical research projects; see Fig. 5.4 for the classification; see Fig. 5.5 for corresponding metadata for the selective paper collection.

The presented classification focuses on prominent forms of static and dynamic analysis. In our paper collection, static analysis is considerably more common, and there is a substantial variety of different analyses. We also indicate two additional dimensions for analyses. An analysis is concerned with *evolution*, when different versions of items, sources, or languages are considered. The dimension of *clustering* implies grouping by geographical characteristics. By no means, our classification

[4] The TIOBE Index of language popularity: `http://www.tiobe.com/tpci.htm`

[5] Another web site on language popularity: `http://langpop.com/`

[6] Language Popularity Index tool: `http://lang-index.sourceforge.net/`

| | |
|---|---|
| Static analysis | Source code or other static entities are analyzed. |
| Validity | The validity of items in terms of syntax or type system is analyzed. |
| Metrics | Metrics are analyzed. |
| Size | The size of items is analyzed, e.g., in terms of lines of code. |
| Complexity | The complexity of items is analyzed, e.g., the McCabe complexity. |
| Structural properties | Example: the depth of inheritance hierarchy in OO programs. |
| Coverage | The coverage of language constructs is analyzed. |
| Styles | The usage of coding styles is analyzed. |
| Patterns | The usage of patterns, e.g., design patterns, is analyzed. |
| Cloning | Cloning across items of the corpus is analyzed. |
| Bugs | The items are analyzed w.r.t. bugs that go beyond syntax and type errors. |
| Dynamic analysis | Actual program runs are analyzed. |
| Profiles | Execution frequencies of methods, for example, are analyzed. |
| Traces | Execution traces of method calls, for example, are analyzed. |
| Dimensions of analysis | Orthogonal dimensions applicable to analyses. |
| Evolution | An analysis is carried out comparatively for multiple versions. |
| Clustering | The corpus is clustered by metadata such as country, team size, or others. |

**Figure 5.4.** Classification of analyses

| | [16] | [24] | [34] | [37] | [40] | [44] | [71] | [81] | [82] | [85] | [108] | [115] | [120] | [1] | [154] | [160] | [187] |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| validity | | | | | | ● | | ● | | | | | | ● | ● | | |
| metrics | ● | ● | | ● | ● | | | ● | ● | ● | | | ● | ● | | ● | ● |
| coverage | | | ● | ● | ● | ● | ● | | | | | ● | ● | ● | | ● | |
| styles | | | | ● | | | ● | | | ● | | | ● | ● | | | |
| patterns | | | | | | | ● | | ● | | | | | | | | |
| cloning | | | | | | | | | | | ● | | | ● | | | |
| bugs | | | | | | ● | | | | | | | | ● | ● | | |
| profiles | | | ● | | | | | ● | | | | ● | | | | | |
| traces | | | | | | | | | | | | | | | | | |
| evolution | | | | | | ● | | ● | | | | | ● | | | | |
| clustering | | | | | | ● | | | ● | | | | | | ● | | |

**Figure 5.5.** Analyses of the selected publications

scheme is complete. For instance, we currently miss characteristics regarding data analysis (e.g., in terms of the involved statistical methods), and the presentation of research results (e.g., in terms of the leveraged tables, charts, etc.).

### 5.2.2 Survey on Corpora Usage

During the first pilot study, we have realized that focusing only on Software Language Engineering is an unnecessary restriction, that the usage of empirical evidence is inherent to many areas of Software Engineering, especially those of practical kind, explicitly using software in their analyses: program comprehension, maintenance, reverse engineering, and re-engineering. We also have developed a better understanding of the classification scheme of the used empirical evidence. This led to the second pilot study that is described in this subsection. The study is reported in the same manner as it was originally reported: unaware of the follow-up study and development of our problem understanding.

**Table 5.1.** Surveyed conferences

| Year | Conf. | Papers | "No" | % | "Yes" | % | "Gray" | |
|---|---|---|---|---|---|---|---|---|
| | | | | | | | (i) | (ii) |
| 2012 | CSMR | 30 | 9 | 30 | 17 | 56.7 | 3 | 1 |
| 2012 | ICPC | 23 | 11 | 47.8 | 8 | 34.8 | 1 | 3 |
| 2012 | ICSE | 87 | 25 | 28.7 | 39 | 44.8 | 6 | 17 |
| 2011 | ICSM | 36 | 6 | 16.7 | 25 | 69.4 | 1 | 4 |
| 2011 | SCAM | 16 | 4 | 25 | 6 | 37.5 | 4 | 2 |
| 2011 | WCRE | 47 | 10 | 21.3 | 21 | 44.7 | 5 | 11 |
| | Total | 239 | 65 | 27.2 | 116 | 48.5 | 20 | 38 |

## Overview

To understand the current demand in corpora among researchers in Software Engineering, we undertook a survey of six relevant conferences: CSMR, ICPC, ICSE, ICSM, SCAM, and WCRE. The survey basis was respectively the latest installment of each conference available from the DBLP bibliography service[7]. We examined the abstracts of the papers from the main research tracks[8], looking for the following signs of corpus usage:

- The *unit* of the corpus is clearly mentioned: system, project, program, etc.
- The *number of units* in the corpus is clearly mentioned. (We also accepted the quantifier "several" in 5 cases.)

Based on this data, we assigned each paper to one of the following categories:

- "No": There are no signs of corpus usage.
- "Yes": There are clear signs of corpus usage, both unit and number of units are mentioned. E.g., "We compared <metrics> [...] by applying them over six open source software systems"

[7] The DBLP Computer Science Bibliography, `http://dblp.uni-trier.de/`

[8] Excluding from our survey industry and early research tracks, invited papers, keynotes, tool descriptions, posters and suchlike.

- "Gray area", if either:
    i. only the unit of the corpus is mentioned. E.g., "and evaluated it on large-scale C/C++ programs",
    ii. or a (controlled) experiment, case study, empirical evaluation, assessment, etc. is mentioned where a corpus has probably been used. E.g., "and further validated with two case studies".

The results of this classification are shown in Table 5.1: almost half of all papers clearly indicate corpus usage; additionally, about one fifth of the papers are in the grey area.

**Software corpora**

The following analysis was carried out on the papers with clear evidence of corpus usage. Furthermore, we focus on the six most frequently used corpus units: system, open-source project (or program), open-source software (or system), application, project, and program. These represent the largest homogenous group and are used in 82 papers, which is 70.7 % of the papers in the "Yes" category. Terminologically, we unify these units in the following under the umbrella term of a *system*.

We examined the full texts of those 82 papers, in order to understand the used corpora in detail. In particular, we were looking for answers to the following questions:

- System code form: source code or binaries?
- What are the software languages of the corpus?
- Dependencies: are systems buildable/runnable?
- Has an established corpus been used?
- What is a typical size of a corpus?
- What systems are frequently used across publications?

The results were as follows.

**Code form.** Looking for an explicit mention of code form in the text of the papers, we discovered that in 63 cases corpora consisted of source code only; in 5 cases, they consisted of bytecode/binaries. From the rest of the papers, we could not extract a clear statement on this account.

**Languages.** We found that corpora are mostly monolingual, with Java and C/C++ being the most popular languages. In 52 cases, Java was mentioned as the only language of the corpus; in 13 cases, C/C++ was mentioned as the only language of the corpus; in 9 cases the corpus was said to contain both units written in Java and in C-like languages (C, C++, C#).

**Dependencies.** We counted towards "resolved dependencies" those cases when it was explicitly mentioned that: i) systems were run; ii) systems were built/compiled; iii) libraries were present; iv) the tool(s) used required resolved dependencies (e.g., Eclipse TPTP). In 31 cases corpora were classified as containing needed code dependencies.

**Established corpora.** In 8 papers, we found explicit mention of one of the following established corpora: DaCapo [29], Qualitas [175], SIR [56], Sourcerer [124], SPEC CPU and SPEC JVM [9]. In 6 other papers, the following online repositories were mentioned as the source of corpus units: Android Market, GitHub, Google Code, SourceForge, ShareJar.[9]

**Corpus size.** Most corpora (57 cases) contain less than ten units. The median of the size distribution is 4, and the most frequent size of a corpus is 2 units.

**Frequently used systems.** We collected the names of used systems as mentioned by the papers and checked if they are provided by our illustrating established corpus, Qualitas. In Table 5.2, we group systems by their frequency of usage (Freq): For each group, we list how many systems belong to it (# Sys); how many of them are found in Qualitas (# In Q.); names of the systems present in Qualitas; and names of the systems not found in Qualitas. E.g., the systems **lucene**, **rhino**, and **xalan** are used in four different corpora, but **rhino** is not present in the Qualitas corpus, while **lucene** and **xalan** are included.

**Table 5.2.** Frequently used systems

| Freq | # Sys | # In Q. | In Qualitas | Not in Qualitas |
|---|---|---|---|---|
| 8 | 1 | 1 | jedit | — |
| 7 | 2 | 2 | argouml, eclipse | — |
| 5 | 3 | 3 | ant, jhotdraw, pmd | — |
| 4 | 3 | 2 | lucene, xalan | rhino |
| 3 | 8 | 4 | aspectj, hibernate, hsqldb, jfreechart | fop, jabref, jface, jython |
| 2 | 20 | 10 | antlr, htmlunit, itext, jmeter, pooka, rssowl, tomcat, vuze, weka, xerces | bloat, chart, compare, debug.core, exvantage, freemind, jdt.core, lexi, quickuml, zxing |

## Discussion

Our survey shows that corpus usage is popular in Software Engineering: half of the papers contain clear signs of it. Most of the used corpora consist of systems in source code form with Java as the most popular language. The survey also shows that established corpora are not widely used, and that home-grown corpora usually are of moderate size and consist of hand-picked systems. The available data suggests that an established corpus (resp. its subset) could have theoretically been used in many cases instead of a home-grown one. In particular, comparison shows that Qualitas, for instance, contains the most popular systems.

Our hypothesis as to why researchers do not embrace established corpora is that they consider engineering a home-grown corpus an easier task than adopting an established one. This may be partly due to the small number of systems used in some

[9] `https://play.google.com/`, `https://github.com/`, `http://code.google.com/`, `http://sourceforge.net/`, `http://www.sharejar.com/`

papers, and partly due to the authors' requirements not being met by the established corpora in their current form.

During our own attempts of adopting established corpora, the most prominent unmet requirement was the absence of unresolved dependencies. The survey shows that more than a third of potential corpus users also have such a requirement. Thus, we believe that adoption of established corpora is significantly simplified by resolving dependencies.

## 5.3 A Survey on Empirical Software Engineering

After the two pilot studies, we have developed sufficient knowledge of the literature survey methods as well as the topic of our interest, the usage of empirical evidence in Software Engineering. This section presents the final and the main survey that was carried out by slightly adjusting our methodology compared to the second pilot study (adjusting the list of conferences, removing inclusion/exclusion criteria, expanding the coding scheme).

### 5.3.1 Methodology

**Table 5.3.** Conferences used in the survey

| Year | Conference | # papers | |
|---|---|---|---|
| | | total | long |
| 2012 | CSMR | 30 | 30 |
| 2012 | ESEM | 43 | 24 |
| 2012 | ICPC | 23 | 21 |
| 2011 | ICSM | 36 | 36 |
| 2012 | MSR | 29 | 18 |
| 2011 | SCAM | 19 | 19 |
| 2011 | WCRE | 47 | 27 |
| | Total | 227 | 175 |

This survey is particularly concerned with (collections of) empirical evidence. Thus, the following questions guide the research:

I How often do Software Engineering papers use *corpora*—collections of empirical evidence?
II What is the nature and characteristics of the used corpora?
III Does common contents occur in the used corpora?

For that, we collected the papers from the latest edition of seven SE conferences: CSMR, ESEM, ICPC, ICSM, MSR, SCAM, and WCRE (see Table 5.3 for details). We used DBLP[10] pages of conferences to identify long papers and downloaded them from digital libraries. We then proceeded to read the papers to perform coding. From a previously done, smaller and more specific literature survey [2] and a pilot study for the present survey, we had some basic understanding of the parts of the scheme to emerge. During the first pass of coding, we started with the empty scheme and completed it eventually to arrive at the current scheme, as described below. During the second pass, we compared profiles of coded papers against the latest version of the scheme, we went through the papers again and filled in the missing details.

We put the collected information in several groups:

**Corpora.** We captured what was used as study objects (e.g., projects), what were their characteristics (e.g., language, open- vs. closed-source, code form), what were the requirements to the study objects, did they come from a specific source (e.g., established dataset or online repository), were they observed over a time (e.g., versions or revisions), what was the nature of preparation of the corpus.

**Forms of empirical research.** During coding, several structural forms emerged that we used for capturing information conveyed in papers: experiments, questionnaires, literature surveys, and comparisons.

**Tools.** We collected mentions of existing tools (e.g., Eclipse, R, Weka) that were used as well as of introduced tools that were introduced in the papers.

**Structural signs of rigorousness/quality.** We paid attention to the following aspects of the study presentation: Do authors use research questions? null hypotheses? Is there a section on definitions and terms? Is validation mentioned? Is there a "Threats to validity" section? Are threats addressed in any structured way?

**Self-classification.** For each paper we captured what words authors use to describe their effort: e.g., case study, experiment.

**Reproducibility.** We tried to understand in each case, if a study can be reproduced. We paid attention to the following signs: Are all details provided for a possible study replication (i.e., versions of used projects, time periods, etc.)? Do authors provide any material used in the paper, e.g., on a supplementary website? Altogether, would it be possible to reproduce the study?

**Assessment.** Finally, we characterized the process of coding: how easy it was to extract information and how confident we are in the result.

We used Python and Bash scripts, Google Refine tool[11], and R project[12] to process the data. We provide online the list of the papers and results of coding[13].

### 5.3.2 Results

In this section, we present the results of our study. We group them similarly to the description provided in Section 5.3.1: details about detected corpora, emerged forms

[10] The DBLP Computer Science Bibliography, `http://dblp.uni-trier.de/`
[11] `http://code.google.com/p/google-refine/`
[12] `http://www.r-project.org/`
[13] `http://softlang.uni-koblenz.de/empsurvey`

of empirical research, used or introduced tools, signs of rigorousness/quality of research, reproducibility of the studies, and, finally, assessment of our effort. When we use the phrase *"on the average"*, we imply the median of the appropriate distribution.

Next to the numbers, we provide framed highlights.

We use formula "X out of Y papers" to provide feeling for the numbers. E.g., "one out of three papers" means that in every three surveyed papers there is one that has the discussed characteristic.

We also provide conference-wise percentage of found characteristics. The table below illustrates the format on an artificial example: conferences are listed from left to right as the percentage increases. Percentage is always given relative to the total number of the long papers in the conference.

**Artificial example**

| CSMR | ESEM | ICPC | ICSM | MSR | SCAM | WCRE |
|---|---|---|---|---|---|---|
| 1 % | 2 % | 3 % | 4 % | 5 % | 6 % | 7 % |

## Corpora

### Usage

Altogether, we have found 198 corpora in 165 papers (28 papers contain more than one corpus).

We decided that more than one corpus was used, when we met at least two of the following motivations mentioned in the paper when describing the purpose of collected empirical evidence: for benchmark or oracle (6 corpora), for training (6 corpora), for evaluation (5 corpora), for investigation (5 corpora), for testing (4 corpora), for investigating quality like accuracy or scalability (4 corpora).

We have found that 168 corpora (used in 145 papers), consist of projects (systems, software); in other cases, corpora contain another kind of study object: image, trace, feature, web log, etc. Till the end of the current subsection (5.3.2), we restrict ourselves to the project-based corpora.

Almost all papers use a corpus of some sort. One out of six papers has more than one corpus. Most of the corpora consist of projects.

**Project-based corpora usage**

| ESEM | ICPC | WCRE | SCAM | CSMR | MSR | ICSM |
|---|---|---|---|---|---|---|
| 58 % | 81 % | 81 % | 84 % | 87 % | 89 % | 94 % |

**Contents** We identified the following common characteristics of project-based corpora.

*Size.* Half of the corpora, 99 cases, have three or less projects (of them, 45 corpora consist of only one project). There are 24 corpora that contain more than 10 projects. We detected large corpora (with more than 100 projects) in 8 papers—one of them introducing an established dataset itself.

*Languages.* Most of the corpora are monolingual (147 cases); most of the remaining ones are bilingual (19 cases). As for the software language, 106 corpora contain projects written in Java, while C-like languages are used in 50 corpora (in C-like languages we include C, C++, C#).

*Code form.* In 125 cases, corpora consist of source code; in 15 cases—of binaries. In the rest of the cases, code of the projects is not used, something else is in focus (developers, requirements, etc.)

*Access.* In 128 cases, corpora consist only of open-source projects; in 12 cases, corpora consist only of projects not available publicly (e.g., industrial software); in 9 cases, corpora are self-written. The remaining cases mix access forms.

**Table 5.4.** Used projects

| Project | # corp |
|---|---|
| JHotDraw | 15 |
| JEdit | 12 |
| Ant | 11 |
| ArgoUML | 11 |
| Eclipse | 11 |
| Firefox | 10 |
| Vuze/Azureus | 8 |
| Linux kernel | 6 |
| Lucene | 6 |
| Mozilla | 6 |
| Hibernate | 5 |

*Projects.* We collected names of the used projects as they are provided by the papers (modulo merging of names like Vuze/Azureus[14]). Table 5.4 lists projects frequently used in the corpora. Eclipse is a complex project, and some corpora make use of its sub-parts, considering them as projects on their own (e.g., JDT Search, PDE Build)—counting such cases, there are altogether 22 papers making use of Eclipse.

*Units.* We captured when some unit related to the project was in the focus of the study: a bug report or a UML class diagram—namely, we would capture the fact when such unit was used to give quantitative information (e.g., in a table presenting number of bug reports in the project under investigation). The most popular units turned out to be bug reports, they are used in 21 corpora; defects (faults, failures) are used in 16 corpora; tests—in 10; traces—in 5.

An average project-based corpus consists of source code of three open-source projects, written in Java. Eclipse or its sub-parts is used in one out of eight papers using project-based corpora. The projects used in at least five papers are JHotDraw, JEdit, Ant, ArgoUML, Firefox, Vuze/Azerus, Linux kernel, Lucene, Mozilla, and Hibernate. Within the corpus, bug reports, defects, tests, and traces can be in the focus of the study.

[14] The project changed its name in 2008.

**Popular projects**

| WCRE | SCAM | ESEM | ICSM | CSMR | ICPC | MSR |
|---|---|---|---|---|---|---|
| 11 % | 11 % | 13 % | 14 % | 23 % | 24 % | 28 % |
| Eclipse | Lynx | Eclipse | ArgoUML | Eclipse | JEdit | Firefox |
| JEdit | Minix | | | | | Eclipse |

*Sources.* Some papers clearly state the source of their corpora. Table 5.6 lists the most popular types of sources and their distribution across conferences.

**Table 5.5.** Online repositories and established datasets

| Repository | # papers |
|---|---|
| SourceForge[1] | 6 |
| Apache.org[2] | 3 |
| GitHub[3] | 3 |
| Android Market[4] | 2 |
| CodePlex[5] | 2 |

| Ref Dataset | # papers |
|---|---|
| [56] SIR | 3 |
| [144] MSR challenge | 2 |
| [80] P-MARt | 2 |
| [151] PROMISE | 2 |
| [175] Qualitas | 2 |

[1] `http://sourceforge.net/`
[2] `http://projects.apache.org/`
[3] `https://github.com/`
[4] Now known as Google Play, `https://play.google.com/store`
[5] `http://www.codeplex.com/`

Online repositories used in more than one paper are listed in Table 5.5. The rest of detected online repositories are used in only one paper each: BlackBerry App World[15], Google Code[16], Launchpad[17], ShareJar[18].

Established datasets used in more than one paper are listed in Table 5.5. Some of the other datasets that used only in one paper each: Bug prediction dataset [49], CHICKEN Scheme benchmarks [19], CoCoMe[20], DaCapo [29], FLOSSMetrics[21], iBUGS[22], SMG2000 benchmark[23], SourcererDB [124], TEFSE challenge[24].

[15] `http://appworld.blackberry.com/webstore`
[16] `http://code.google.com/`
[17] `https://launchpad.net/`
[18] `http://www.sharejar.com/`
[19] `https://github.com/mario-goulart/chicken-benchmarks`
[20] `http://agrausch.informatik.uni-kl.de/CoCoME`
[21] `http://libresoft.es/research/projects/flossmetrics`
[22] `http://www.st.cs.uni-saarland.de/ibugs/`
[23] `https://asc.llnl.gov/computing_resources/purple/archive/benchmarks/smg/`
[24] `http://www.cs.wm.edu/semeru/tefse2011/Challenge.htm`

**Table 5.6.** Sources of corpora

| Type | # papers | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | Total | CSMR | ESEM | ICPC | ICSM | MSR | SCAM | WCRE |
| Established dataset | 20 | 5 | 0 | 2 | 6 | 5 | 0 | 2 |
| Previous work | 13 | 2 | 3 | 2 | 1 | 1 | 2 | 2 |
| Online repository | 12 | 3 | 0 | 1 | 2 | 2 | 1 | 3 |
| Total | 43 | 9 | 3 | 5 | 9 | 7 | 3 | 7 |
| Percentage | 25 | 30 | 13 | 24 | 25 | 39 | 16 | 26 |

One out of four project-based corpora uses an established dataset, previous work, or online repository as a source of the projects. There is no common frequently used dataset or repository. Only SourceForge shows moderately frequent usage.

**Usage of corpora sources**

| ESEM | SCAM | ICPC | ICSM | WCRE | CSMR | MSR |
|---|---|---|---|---|---|---|
| 13 % | 16 % | 24 % | 25 % | 26 % | 30 % | 39 % |

**Evolution**

We encountered 52 papers that use evolution of the projects in their research, meaning that they operate on several versions, releases, etc. To describe the evolution measure, the following terms were used: “version” (21 times), “revision” (11), “commit” (10), “release” (11).

On the average, papers mentioning commits use 3,292 commits; papers with revisions—18,870 revisions; with versions—10 versions; with releases—10 releases.

There are 46 papers that mention a time span of their study. In 36 cases, the unit of the time span is a year and on the average such papers are concerned with a 8-year span.

We found 23 papers to mention what version control system was involved in the study. CVS is mentioned 11 times, SVN—11 times, Git and Mercurial—4 and 2 times respectively.

One out of three papers with project-based corpora uses evolution aspect in its research. In half of the cases, large-scale evolution is involved: several thousands commits/revisions or ten versions/releases of projects—often spanning several years of a project’s lifetime.

**Evolution usage**

| ICPC | SCAM | ESEM | CSMR | ICSM | WCRE | MSR |
|---|---|---|---|---|---|---|
| 14 % | 16 % | 21 % | 33 % | 33 % | 33 % | 56 % |

**Requirements**

Papers often mention requirements to the used empirical evidence, usually dictated by the research goals,—we collected these requirements to the corpora, explicit as well as implicit. For instance, an implicit requirement for a bug tracking system is inferred if the paper uses bug reports of the projects under investigation. The most popular direction of requirements is the presence of some 'ecosystem' (found in 37 papers): existence of bug tracking systems, mailing lists, documentation (e.g., user manuals). Another popular requirement, found in 25 papers, has to do with the size of the projects: small, sufficient, large, or of particular size (as specific as "medium of the sizes of the ten most popular Sourceforge projects"), or the need of diversity of sizes. In 23 papers, it was stated that the used projects were chosen because they were used in previous work (of the same or other authors). Language-related requirements were present in 22 papers for a specific language or for the diversity of languages in a corpus. In 14 papers, the choice of projects was attributed to either diversity of application domains or to a specific domain. Some aspect of the used projects was mentioned as essential in 14 papers: active or wide-spread usage, popularity, well-known and established software. Other popular requirements include presence of development history (15 papers), dependencies (11 papers), or tests (10 papers).

One out of five papers requires the projects of its corpus to have an ecosystem: a bug tracker, or a mailing list, or some kind of documentation. Other requirements focus on the size and language of the projects, application domain, development history, etc.

**Popular requirements**

| SCAM | MSR | ICSM | ICPC | ESEM | WCRE | CSMR |
|---|---|---|---|---|---|---|
| 11 % | 28 % | 14 % | 24 % | 13 % | 11 % | 23 % |
| domain | ecosys | size | size | ecosys | ecosys | ecosys |
| lang | prev.work | | | | | |
| size | | | | | | |

**Tuning**

We captured what kind of action is applied to a corpus during research. In 20 papers, sources or binaries were modified by instrumentation, faults/clones injection, adjusting identifiers, etc. In 15 papers, tests needed to be run against the corpus either to verify made modifications or to collect the data. In 10 papers, the corpora had to be executed in order to perform the needed analysis or to collect data. In 6 papers, some filtering of the contents of the corpus was needed to, e.g., identify the main source code/main part of the project.

We have detected few common actions applied to corpora during research: source code/binaries modification; execution of the tests on the corpus or of the corpus itself; filtering of the corpus contents. Altogether, one out of four papers contains signs of one of these actions.

**Popular actions**

| ESEM | MSR | SCAM | ICPC | ICSM | WCRE | CSMR |
|---|---|---|---|---|---|---|
| 8 % | 11 % | 11 % | 14 % | 19 % | 19 % | 20 % |
| tests | run | modif. | modif. | tests | modif. | modif. |

We captured manual effort that went into creation of a corpus, e.g., when a paper mentions setting up environments and providing needed libraries in order to execute the corpus. For that, we graded each corpus on the following scale. *None*: no manual effort mentioned (120 corpora); *some*: some manual effort mentioned, e.g., manual detection of design patterns in source code (33 corpora); and *all* means that the corpus is self-written (10 corpora).

One out of four project-based corpora requires some manual effort.

**Manual effort**

| CSMR | ESEM | ICSM | SCAM | ICPC | MSR | WCRE |
|---|---|---|---|---|---|---|
| 13 % | 13 % | 22 % | 26 % | 33 % | 33 % | 37 % |

## Emerged forms

We did not use any theoretical definition for what is considered to be a questionnaire or an experiment. The developed definitions are structural, composed of the characteristics that emerged from the papers, as they were discussed and structurally supported by the authors.

**Experiment**

We have identified 22 experiments in 19 papers. Except for two, they all involve human subjects. On the average, an experiment has 16 participants. The maximum number of participants is 128, the minimum is 2, first and third quartiles are 5 and 34 respectively. In 21 cases, an experiment uses a corpus (in 17 cases, a project-based one); 20 questionnaires are used in 10 experiments.

In two-thirds of the experiments, participants come from one population, the remaining experiments draw participants from two or three populations. The most common source of participants is students; sometimes distinguished by their level—graduate, undergraduate, Bachelor, Master, and PhD students. In one-third of the cases, professionals are involved (full-time developers, experts, industry practitioners, etc.). In half of the cases, participants form the one and only group in the experiment. When there is more than one group (usually, two—with a couple of exceptions of 4 and 5 groups), the group is representing a treatment (a task), or an experience level, or a gender. On the average, an experiment has 4 tasks and lasts for an hour (with a few exceptions when an experiment takes several weeks or even a month).

In 6 cases, it is mentioned that an experiment had a pilot study. In 6 cases, it is mentioned that participants of the experiment were offered compensation: monetary or another kind of incentive (e.g., a box of candy).

The main requirement for the participants is their experience: basic knowledge of used technology, or language, or IDE. As for the tasks, they are expected to be of a certain size (e.g., a method body to fit on one page), or of certain contents (e.g., contain "if" statements). The usual requirement for an experiment also is either that the tested tool or used code is unfamiliar to the participants, or on the contrary that the background is familiar (e.g., well-known design patterns).

One out of ten papers contains an experiment. The majority of the experiments use project-based corpora; experiments often use questionnaires, usually two per experiment. An average experiment involves 16 students, often in two groups (by the received treatment or experience level); it consists of four tasks and lasts for an hour. One out of four experiments suggests some compensation to its participants; one out of four experiments is preceded by a pilot study.

ICPC and ESEM are the main source of experiments involving professionals.

**Experiments**

| MSR | SCAM | CSMR | WCRE | ICSM | ESEM | ICPC |
|---|---|---|---|---|---|---|
| 0 % | 0 % | 3 % | 7 % | 8 % | 21 % | 38 % |

**Questionnaire**

Altogether, we have found 36 questionnaires in 24 papers. As mentioned, 20 questionnaires are used in experiments—to distinguish, we will refer to them as experiment-related and the other 16 we will qualify as experiment-unrelated.

Sizewise, there is no particular difference between experiment-related and -unrelated questionnaires. On the average, both have 20 questions grouped in one section. In 6 cases, an experiment-unrelated questionnaire has a corpus.

While experiment-related questionnaires have the same participants as the experiments they relate to (i.e., involve mostly students), experiment-unrelated questionnaires involve professionals (testers, managers, experts, consultants, software engineers) as participants in two-thirds of the cases. On the average, an experiment-unrelated questionnaire has 12 participants. When it was possible (6 cases), we calculated how many participants took part in the experiment-unrelated questionnaire compared to the initial number of invited participants. On the average, 19 % take part in the end, in the worst case the ratio can be as low as 5 %.

While experiment-related questionnaires have the same requirements regarding the participants as the experiments they relate to, experiment-unrelated questionnaires have requirements concerned with the participants' experience (e.g., Java experience) or expertise (specific area of experience such as clone detection or web development).

When related to experiments, questionnaires are often performed before (referred to as "pretest" in 6 cases) and after the experiment (referred to as "posttest" in 9 cases).

In 5 cases, an experiment-unrelated questionnaire was preceded by a pilot study.

More than half of the detected questionnaires are used in experiments—often as pretest and postttest questionnaires. The other half, experiment-unrelated questionnaires, are found in one out of twelve papers. Sizewise, on the average there is no difference between experiment-related and -unrelated questionnaires. Experiment-unrelated questionnaires usually involve professionals as participants—in contrast to experiment-related questionnaires that mostly use students. Typical requirements for participants in experiment-unrelated questionnaires have to do with experience or expertise. One out of three experiment-unrelated questionnaires are preceded by a pilot study.

**Experiment-unrelated questionnaires**

| MSR | CSMR | SCAM | WCRE | ICSM | ICPC | ESEM |
|---|---|---|---|---|---|---|
| 0 % | 3 % | 5 % | 7 % | 8 % | 19 % | 25 % |

**Literature survey**

We have found 6 literature surveys in 5 papers. Except for one, they provide extensive details on how the survey was conducted. In particular, the used methodology is clearly stated: four times it is said to be a "systematic literature review" and once a "quasi systematic literature review". In three cases, the systematic literature review was done following the guidelines by Kitchenham [109].

The papers are initially collected either by searching digital libraries or from the proceedings of specific conferences and journals. Among used digital libraries are EI Compendex, Google Scholar, ISI, and Scopus—the latter was used in two papers. As for the conferences and journals, there is no intersection between the lists of names—except for ICSE, which was used in two papers.

On the average, a literature survey starts with 2161 papers, its final set contains 35 papers, meaning that on the average only 1.6 % papers are taken into account in the end. The percentage can be as high as 39 % and as low as 0 %.

Requirements for papers to be included into the survey are usually related to the scope of the investigated research. Other requirements are concerned with the paper itself: available online, written in English, a long paper, with empirical validation.

After all the papers are collected, they are filtered based on the titles and abstracts, which are examined manually by the researchers (in one case, also conclusions were taken into account; in another case, full text of the papers was searched for keywords). Then the full text of each paper is read and the final decision is made as to whether to consider the paper relevant.

Literature surveys are quite rare: only one out of 35 papers contains it. On the average, a literature survey starts with few thousand papers to be filtered down to few dozens papers that will be analyzed. Usually, the first round of filtering is based on the title and abstract, then the full text of the papers is considered. There is not enough information to conclude about frequently used digital libraries or conferences/journals. Half of the surveys were following guidelines of systematic literature reviews by Kitchenham [109].

**Literature surveys**

| ICSM | MSR | SCAM | WCRE | CSMR | ICPC | ESEM |
|---|---|---|---|---|---|---|
| 0 % | 0 % | 0 % | 0 % | 3 % | 5 % | 13 % |

**Comparisons**

During the coding phase of the survey, we noticed the recurring motif of comparisons in the papers. While we did not assess the scope nor the goal, we have coded the basic information: what is the nature of the subjects being compared (tools, techniques), how many subjects are compared, and is one of them introduced in the paper.

We have found comparisons in 56 papers. Almost all of them (except for 5 papers), use project-based corpora. Half of the time, a comparison includes the technique, approach, or tool that was introduced in the study—with the apparent reason to evaluate the proposed technique, approach, or tool. On the average, in such evaluation was used one other technique, approach, or tool. In the other cases, were compared metrics, tools, algorithms, designs, etc. For such comparisons, on the average, the group of compared entities was of size 3.

One out of three papers compares tools, techniques, approaches, metrics, etc.—half of the time, to evaluate what was introduced in the study. On the average, such evaluation involves one other entity. In the other half of the cases, the average number of compared entities is 3.

**Comparisons**

| ICPC | MSR | SCAM | WCRE | ESEM | ICSM | CSMR |
|---|---|---|---|---|---|---|
| 19 % | 22 % | 26 % | 30 % | 33 % | 36 % | 47 % |

## Tools

We have found 46 papers to introduce a tool (where we were able to capture this fact only if the name of the tool was mentioned or it was clearly stated that “a prototype”

**Table 5.7.** Existing tools used in the papers

| Tool | # papers | Tool | # papers |
|---|---|---|---|
| Eclipse[1] | 25 | MALLET[7] | 4 |
| R project[2] | 16 | ChangeDistiller[8] | 3 |
| CCFinder[3] | 6 | CodeSurfer[9] | 3 |
| Understand[4] | 6 | Evolizer[10] | 3 |
| Weka[5] | 6 | RapidMiner[11] | 3 |
| ConQAT[6] | 4 | RECODER[12] | 3 |

[1] `http://eclipse.org`
[2] `http://www.r-project.org/`
[3] `http://www.ccfinder.net/`
[4] `http://www.scitools.com/`
[5] `http://www.cs.waikato.ac.nz/ml/weka/`
[6] `https://www.conqat.org/`
[7] `http://mallet.cs.umass.edu/`
[8] `http://www.ifi.uzh.ch/seal/research/tools/changeDistiller.htm`
[9] `http://www.grammatech.com/products/codesurfer/overview.html`
[10] `http://www.ifi.uzh.ch/seal/research/tools/evolizer.html`
[11] `http://rapid-i.com/content/view/181/190/`
[12] `http://sourceforge.net/projects/recoder/`

is implemented). In 46 more papers, we detected that additional, helper tooling for the current purpose of the study is implemented (parsers, analyzers, and so on).

When names of existing tools were explicitly mentioned to be used, we collected the names. We have found that in 126 cases, a paper makes use of existing tools. On the average, a paper uses 2 tools; the captured maximum is 6. The frequently used tools are listed in Table 5.7. We counted towards Eclipse usage also cases when a paper used an existing tool that we know to be an Eclipse plug-in. For brevity, we omit names of 19 more tools that were used in two papers.

One out of four papers introduces a new tool; another one out of four papers uses some home-grown tooling. Almost three out of four papers use existing tools.

The most popular standard tool, Eclipse—an IDE and a platform for plug-in development—is used in one out of seven papers. Other popular tools cater for source code analysis, clone detection, evolution analysis, data mining, statistics, quality analysis, document classification.

**Home-grown tooling**

| ICPC | ESEM | CSMR | ICSM | MSR | WCRE | SCAM |
|---|---|---|---|---|---|---|
| 5 % | 17 % | 20 % | 33 % | 33 % | 33 % | 42 % |

**Introduced tools**

| ESEM | MSR | ICPC | SCAM | CSMR | WCRE | ICSM |
|---|---|---|---|---|---|---|
| 4 % | 11 % | 19 % | 21 % | 30 % | 37 % | 44 % |

## Structural signs of rigorousness/quality

We do not aim to assess the quality or rigorousness of the studies. We capture presence of some of the aspects that are taken into account when assessing rigorousness/quality of research (cf., [97])—in that, we restrict ourselves only to the structural aspects.

**Study presentation aspects**

A clear set of definitions for the terms used in the paper is found in 25 papers. Research questions are adopted in 83 papers. In 22 papers, an approach "Goal-Question-Metric" [23] is used. Explicit mention of null hypothesis or hypotheses is found in 23 papers. Section "Threats to validity" is present in 111 papers; of them, 75 discuss threats using classification described, e.g., in [195]: threats to external (mentioned in 73 papers), internal (59 papers), construct (53 papers), and conclusion (26 papers) validity.

If to consider combinations of these signs (definitions, research questions, hypotheses, and threats), the most popular one is the absence of all of them: demonstrated by 42 papers. The second most popular combination is presence of research questions and threats to validity: found in 34 papers. The third most popular—usage of only threats to validity—found in 29 papers. Together, these three combinations describe 60 % of the papers.

Half of the papers use research questions to structure their study. One out of seven papers use "Goal-Question-Metric" approach and/or formulate (null) hypotheses to structure their research. One out of seven papers provides an explicit set of definitions of the terms used in the study. Threats to validity are discussed in three out of five papers.

The following three combinations of structural signs describe at least half of the papers in each conference, except for WCRE, where only 44 % of papers are covered by these combinations.

**No structural signs**

| ICPC | MSR | WCRE | ESEM | ICSM | CSMR | SCAM |
|---|---|---|---|---|---|---|
| 14 % | 17 % | 19 % | 21 % | 22 % | 27 % | 53 % |

**Both research questions and threats to validity**

| ICSM | WCRE | SCAM | CSMR | MSR | ICPC | ESEM |
|---|---|---|---|---|---|---|
| 8 % | 11 % | 16 % | 20 % | 22 % | 24 % | 42 % |

**Only threats to validity**

| ESEM | MSR | SCAM | WCRE | CSMR | ICPC | ICSM |
|---|---|---|---|---|---|---|
| 4 % | 11 % | 11 % | 15 % | 17 % | 19 % | 31 % |

**Validation**

We captured the mentions of performed validation of done research. We have found evidence of some kind of validation in 88 papers. In 50 cases, validation was manually performed: either the results are small enough, or a sufficient subset is checked. In 27 cases, validation was done against existing or prepared results: actual data (when evaluating predictions), data from previous work, or an oracle/gold standard. In 8 cases, cross-validation was used.

**Self-classification**

**Table 5.8.** Self classification

| Type | # |
|---|---|
| case study | 48 |
| experiment | 44 |
| empirical study | 22 |
| evaluation | 14 |
| exploratory study | 6 |
| ... | ... |

We collected explicit self-classifications from the papers; from the sentences like "we have conducted a case study" we would conclude that the current paper is a case study. Some of the self-classifications were very detailed and precise, e.g., "a pre/post-test quasi experiment", in such cases we reduced the type to a simpler version, e.g., an experiment. We would also count terms like "experimental assessment" or "experimental study" towards the experiment type. As seen from Table 5.8, most often authors use terms such as "case study" and "experiment" to describe their research. In some cases, papers contain more than one self-classification (24 cases). In 36 papers, we could not detect any self-classification.

Four out of five papers provide self-classification, but it might be vague. The most popular term, 'case study,' may be misused. Cf., "There is much confusion in the SE literature over what constitutes a case study. The term is often used to mean a worked example. As an empirical method, a case study is something very different." [170]. Cf., "... our sample indicated a large misuse of the term case study." [198]

**Self-classification**

| SCAM | MSR | CSMR | WCRE | ICPC | ESEM | ICSM |
|---|---|---|---|---|---|---|
| 37 % | 61 % | 80 % | 81 % | 86 % | 88 % | 94 % |

## Reproducibility

We looked for signs of additionally provided data for a replication of the study. Since it is usually done via the Internet, we searched the papers for (the stems of) the following keywords: “available,” “download,” “upload,” “reproduce,” “replicate,” “host,”, “URL,” “website,” “http,” “html”. In such manner, we have found links in 61 papers. In 6 cases, we could not find any mentioned material, tools or data,—links led to a general page or to a homepage, which we searched thoroughly but without success. In 3 more cases, we have found replication material on the website after some searching.

One out of three papers additionally provides online some data from the study, though not always to be found.

**Additional data provided**

| SCAM | ICSM | CSMR | MSR | WCRE | ESEM | ICPC |
|---|---|---|---|---|---|---|
| 26 % | 31 % | 33 % | 33 % | 33 % | 38 % | 48 % |

As to the nature of the provided data, in 25 cases, an introduced tool or tooling used in the research is provided. In 15 cases, the used corpus—in full or partially—is provided; the complete description of the corpus (list of used projects with their versions and/or links) is provided by 6 more papers. Raw data are available for 14 papers; the same number of papers provide final or/and additional results of the study.

When the corpus is not provided by the paper, but the names of the used projects are mentioned, the main aspect of being able to reproduce the corpus is knowing which versions of the projects were used. We noticed that in 21 papers versions of the used projects are not provided. In 67 papers, versions of the projects are mentioned explicitly; in 26 more cases, it is possible to reconstruct the version from the mentioned time periods that the study spans.

Altogether, we judged 29 papers to be reproducible, meaning that either all components were provided by the authors or we concluded that the paper contains enough details to collect exactly the same corpus and the same tools. We did *not* judge if it is possible to follow the provided instructions, specific to the reported research—as, for instance, was done in the work by González-Barahona and Robles [78].

We also note that 8 papers mention that they are doing a replication in their study, of them 3 papers with self-replication.

We judged one out of six papers to be reproducible with respect to the used corpus and tools. We did *not* assess whether enough details were provided to re-conduct the research itself.

**Judged to be reproducible**

| ICSM | WCRE | SCAM | ICPC | ESEM | CSMR | MSR |
|---|---|---|---|---|---|---|
| 3 % | 4 % | 16 % | 19 % | 25 % | 27 % | 33 % |

### Assessment

Though usually information we extracted from the papers was scattered across different sections, half of the papers had tables (listing projects, their names, versions, used releases, and similar information) that helped us during the coding phase of the papers. We captured our confidence in the end result of the coded profile of each paper. For that, we used the following scale: high, moderate, and low levels of confidence. The results are as follows: high—81 papers, moderate—78 papers, low—16 papers.

We have low confidence in one out of eleven papers that we have coded. In the rest, half of the time we are moderately confident and half of the time—highly confident in the results.

**High confidence**

| WCRE | SCAM | ICSM | ICPC | ESEM | CSMR | MSR |
|---|---|---|---|---|---|---|
| 15 % | 26 % | 42 % | 52 % | 54 % | 60 % | 78 % |

**Moderate confidence**

| MSR | CSMR | ESEM | ICPC | ICSM | SCAM | WCRE |
|---|---|---|---|---|---|---|
| 17 % | 33 % | 33 % | 43 % | 53 % | 58 % | 67 % |

**Low confidence**

| ICPC | ICSM | MSR | CSMR | ESEM | SCAM | WCRE |
|---|---|---|---|---|---|---|
| 0 % | 6 % | 6 % | 7 % | 13 % | 16 % | 19 % |

## 5.4 Threats to Validity

### Choice of the papers

We did not use journal articles—while they might provide more information or be of higher quality, we wanted to capture the state of the common research, of which we believe conference proceedings to be more representative[25]. We have chosen conferences with proceedings of similar and reasonable size (see Table 5.3): so that not to skew the general results by one larger conference and so that to include all the papers but still be able to process them within reasonable period of time. Specifically, we excluded the ICSE conference, which had 87 long papers in the proceedings of 2012 edition. Altogether, this means our results might not be generalizable, but we believe them to be representative enough.

[25] While we do not aim to define what “common research” is, we find indirect justification for such position in the work that compares scientific impact of conference and journal publications (see, for example, [152] and [66])

### Choice of the period

Since we perform a snapshot study, it might be that some of the discovered numbers are a coincidental spike. Possible future work, in order to provide more details and deeper understanding, will have to be a longitudinal study—along the same lines as we have investigated the health of Software Engineering conferences in [8], observing 11 conferences over a period of more than 10 years.

### Data extraction

The coding phase of the papers consisted of the manual effort with occasional search by specific keywords (mentioned in the appropriate subsections of Section 5.3.2). In 5 cases, papers were OCR-scanned (i.e., non-searchable).

**Human factor.** Coding was done by one researcher, but the results of the first pass were cross-validated during the second pass as well as during the aggregation phase. When in doubt, the researcher constantly referred back to the surveyed papers to double-check.

**Scheme.** We do not claim our coding scheme to be complete or advanced. We captured basic data related to the used empirical evidence, often either obvious or structurally supported. Therefore, we might miss sophisticated or under-specified forms of empirical research.

## 5.5 Related Work

We summarize related work in Table 5.9. We compare our work to other literature surveys of SE research.

There are key differences between our survey and previous work. The cited surveys start with a predefined schema, while we allow our schema to emerge. Further, the cited surveys focus on SE research in some way, while we are interested in whatever empirical evidence is used to facilitate SE research. For instance, Kitchenham and Sjøberg et al. are interested in specific kinds of studies, systematic reviews, and controlled experiments respectively; Glass et al. surveyed a sample of all papers, without filtering by types, but their intention was to capture in what areas SE research is done and how.

Below we compare the findings of related work where they overlap with ours. As a general remark, we believe that our findings quantitatively differ from previous findings because of several factors: i) the dependence on the choice of venues: even conferences in our study differ considerably; ii) passed time: there is at least a five-year gap, during which popularity of empirical research and of its particular forms might have grown; iii) the cited papers use mostly journals: this may increase the aforementioned gap because of the longer process for journal publications; iv) snapshot versus longitudinal approach: we take into account all papers of the latest proceedings while the cited papers focus on a sample across several years.

The closest work to ours is by Zannier et al.: they measured quantity and quality of empirical evaluation in ICSE papers over the years. Our work provides a snapshot study aiming to represent SE research broadly across conferences. Zannier et al. when assigning types to the papers, could confirm the self-classification of half of the studies. Which agrees with our observation that self-classification is rather weak among SE papers. They also observe the extremely low usage of hypotheses (only one paper) and absence of replications. We do find some adoption of null hypotheses and replications.

**Table 5.9.** Related work

| Name | Ref | Year | # used j | # used c | Period | # papers total | # papers sel. | # papers rel. | Focus | Coding schema |
|---|---|---|---|---|---|---|---|---|---|---|
| Glass et al. | [76] | 2002 | 6 | 0 | 1995–1999 | — | 369 | 369 | Characteristics of SE research | Topics, research approaches and methods, theoretical basis, level of analysis |
| Sjøberg et al. | [168] | 2005 | 9 | 3 | 1993–2002 | 5453 | 103 | 103 | Controlled experiments | Extent, topic, subjects, task and environment, replication, internal and external validity |
| Zannier et al. | [198] | 2006 | 0 | 1 | 1975–2005 | 1227 | 63 | 44 | Empirical evaluation: quantity and soundness | Study type, sampling type, target and used population, evaluaton type, proper use of analysis, usage of hypotheses |
| Kitchenham et al. | [111] | 2009 | 10 | 3 | 2004–2007 | 2506 | 33 | 19 | Systematic reviews | Inclusion and exclusion criteria, coverage, quality/validity assessment, description of the basic data |
| Our study | | 2013 | 0 | 7 | 2011/2012 | 227 | 175 | 175 | Empirical evidence | Emerged classification |

Legend: *j* and *c* stand for *journals* and *conferences*; *sel.* and *rel.* stand for *selected* and *relevant*.

According to their classification, Glass et al. have found 1.1 % papers to contain literature reviews and 3 % papers to present "laboratory experiment (human subjects)." We also discover that number of literature surveys and experiments is low, but relatively it increased 2-3 times.

Kitchenham et al. considered to be systematic literature reviews only 0.75% of surveyed papers. We have found literature surveys in 5 papers, one of which did not contain a clear methodology—a requirement to be met by Kitchenham's inclusion criteria—leaving 4 papers. Thus, our percentage of detected literature surveys is 2.3 %

According to Sjøberg et al.'s study, only 2 % of the papers contain experiments, while we discover 10 % surveyed papers to contain an experiment. On the average, Sjøberg et al. detected an experiment to involve 30 participants—in 72.6 % cases only students, in 18.6 % cases only professionals, and in 8 % cases mixed groups. We have found that on the average an experiment involves 16 participants—in 57 % cases only students, in 14 % cases only professionals, and in 29 % cases mixed groups.

## 5.6 Conclusion

We have presented a literature survey on the current state of empirical research in Software Engineering.

**Answers to the questions**

Coming back to the initial questions that motivated our research (see Section 5.1), we suggest the following answers:

I The overwhelming majority of Software Engineering papers use corpora—collections of empirical evidence.
II The majority of the corpora consist of projects and can be characterized by size, code form, software language, used evolution measures, and others.
III We have detected some recurring projects with low frequency though. (See discussion below.)

**"Holy grail" vs. common ground**

Though projects are used in the majority of SE papers and characteristics of the used corpora are quite typical (number of used projects, language, etc.), the usage of established datasets is low. We suggest two possible reasons. First, adoption may be low only yet: among detected datasets being used (see Table 5.5), the oldest dataset, SIR, was introduced in 2005, the youngest, Qualitas—in 2010. Second, researchers may prefer to collect and prepare their corpora themselves, because there might not be a "holy grail" among corpora to suit all possible needs. Partially, this last guess is supported by the fact that even on the level of projects no clear favorite was detected among SE papers. On the other hand, we find that three out of seven conferences have favorite projects, when considered separately—projects that are used by a quarter of the papers within these conferences. This leads to a refined version of the third question in our study: When it is possible to detect commonly used projects within a conference, would it be useful to provide a curated version of them? This is a topic for future work.

### Top-down vs. bottom-up introduction of methodology

While there is need for adoption of advanced and theoretically specified forms of empirical research, we believe that there is a certain amount of de facto empirical research that has formed historically. Our position is that by understanding its characteristics, its quality can be improved or at least assessed.

# 6 Corpus (Re-)Engineering

The previous chapter assesses the existing demand of corpora in empirical Software Engineering research: we have discovered that more than 80% of papers use software projects as empirical evidence. This motivates us to provide matching supply in our area of expertise. In this chapter, we identify obstacles to corpus adoption based on our own experience (see Chapter 4) and develop a method for comprehending and improving corpus content, producing a complete, automatically buildable corpus, with extended metadata. We apply the method to the Qualitas corpus [175], whose adoption is thereby simplified.

**Road-map of the chapter**

- Section 6.1 provides a brief motivation for the study.
- Section 6.2 describes the methodology of corpus (re-)engineering.
- Section 6.3 contains results of applying the methodology to Qualitas.
- Section 6.4 discusses threats to validity for this empirical study.
- Section 6.5 discusses related work.
- Section 6.6 concludes the chapter.

**Reproducibility**

We provide the refined version of Qualitas on the supplementary website[1].

**Related publications**

Research presented in this chapter underwent the peer-reviewing procedure and was published in the proceedings of International Conference on Program Comprehension in 2013 [6]. This chapter reports an extended and detailed version compared to the publication.

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

## 6.1 Introduction

Empirical software research often needs corpora to test a hypothesis, to validate a tool, or to illustrate an approach. Software corpora have been, for instance, used to evaluate tools for design pattern detection [65], API method recommendation [87], and refactoring automation [141]—just to mention a few areas and papers. All such efforts face the choice as to whether to use a more or less home-grown corpus, which is the case for the papers just referenced, or instead an established corpus, such as the Qualitas curated Java code collection [175] comprising over 100 systems written in Java.

As we have demonstrated in Chapter 5, there is a common need in corpora in the contemporary Software Engineering research. Based on our experience, described in Chapter 4, in the present effort we identify obstacles of corpus adoption and corresponding means of corpus (re-)engineering, thereby improving corpora and simplifying their adoption. More specificallly, we develop a method for corpus (re-)engineering, which we ultimately demonstrate and validate by their application to Qualitas.

### 6.1.1 Benefits of Using an Established Corpus

Creating and using established corpora of experiment subjects offers significant benefits to the research field as a whole. Drawing the experiment subjects from an established corpus increases comparability of approaches as well as reproducibility of research. Both of these goals are highly valued in empirical sciences.

There is also hope that availability of established corpora would entice researchers to validate their results on a larger number of experiment subjects, though this outcome is of course subject to individual resource constraints. An established corpus offers furthermore an opportunity to exercise quality control, e.g., by reaching a consensus that a collection is, in some sense, representative, free from bias or artifacts.

Apart from the benefits to the field, there are also practical reasons and benefits for the individual researcher to use an established corpus. A well-maintained corpus reduces the effort in obtaining, organizing, preparing, and maintaining experiment subjects. An established corpus provides a stable point of reference to the experiment subjects used while limiting the documentation-related burden for the researcher. A corpus may readily provide additional metadata, which can be useful for particular research applications.

Of course, in practice, using an established corpus is hardly an all-or-nothing affair. While the payoff is highest when a large number of experiment subjects is used, it is already present for research on a small subject populace. Conversely, improving an established corpus benefits many users at once. Nonetheless, the method that we propose is an aid to everybody who is interested in setting up a well-engineered corpus.

### 6.1.2 Obstacles to Corpus Adoption

Guided by our experiences in using an established corpus for own research, we address two obstacles for corpus adoption. As our pilot study suggests (see Section 5.2.2), others could also benefit from the improvements.

Obstacle I is 'unresolved dependencies'. Being able to build and/or run the system is crucial in many research questions, e.g., when studying API usage [173, 92, 148], for code instrumentation [99, 165], or dynamic analysis [150, 69]. A corpus may include compiled code/binaries as provided by system developers, but this is often insufficient due to missing dependencies such as libraries.

Obstacle II is 'insufficient metadata'. System distributions often contain disparate code content such as unit tests, demos and test cases, and third-party libraries. Failing to discern between these categories in empirical studies may lead to skewed results. Indeed, the metadata of established corpora does not allow one, e.g., to automatically identify and isolate test cases [141], the system core [52], or, conversely, third-party libraries [174].

## 6.2 A Method for Corpus (Re-)Engineering

We will now present a method to produce a corpus that i) includes all necessary dependencies, ii) can be built and analyzed automatically, in a uniform way, and iii) contains extended and precise metadata about systems and their components. We submit that the cornerstone of this method for corpus (re-)engineering is that the build process for systems in the corpus is managed in a scalable and traceable way such that all dependencies are resolved and metadata about the system scope is collected.

The method is defined for Java programs, but can be analogously applied to other similar languages.

### 6.2.1 Underlying Concepts

A *system* is a set of files. We assume a system is available in two forms. First, the *source-code form* consists of Java files, libraries, build scripts, documentation, and so on, and we refer to the totality of included Java files as *source code*. Second, the *binary form* usually consists of compiled source code, resources, etc., packed into JARs; it may also optionally contain a number of library JARs. (For example, the Qualitas corpus readily serves both forms.)

A Java file may contain one or more *Java types* (classes, interfaces, enums, annotations) that are distinguished by their *qualified names*. A *build* is a process, during which source code is compiled and packed into *built JARs*, which can be distributed as part of the binary system form. We use the term *root* to refer to any source directory from which the hierarchy of directories with Java packages descends. Basically, one obtains a root by removing from the full path to a source file the suffix corresponding to the Java package[2].

We use two classifications of Java types to describe system boundaries. The first one consists of a set of package prefixes containing *system* types, as suggested by the authors of Qualitas. System types are part of the system in question—as opposed to, say, third-party types. The second classification is more fine-grained: Java types are classified as *core*, *test*, or *demo*, reflecting the type's function within the system. We will later see how the second classification may be computed by a heuristic.

### 6.2.2 Method

Consider the following, partially automated pseudocode:

```
1. input : $corpus, systemCandidateList$
2. output : $corpus$
3. for each $name$ in $systemCandidateList$ :
4.   $(p_{src}, p_{bin}) = obtainSystem(name)$;
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 $validate(p')$ :
15.      $corpus = corpus + p'$;
16.      $factExtraction(p')$;
```

The input is a (possibly empty) *corpus* to be extended and a list of candidate systems, *systemCandidateList*, to be added to it. The output is the corpus populated with refined systems.

We assume that a system can be obtained both in its source-code and binary forms (line 4): e.g., by downloading them from the system's website. During an exploratory build (line 5), the nature of the system is manually investigated by an expert. We detect how the system 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

[2] E.g., for a file `systemA/src/org/foo/bar/Example.java` that contains a type with qualified name `org.foo.bar.Example`, the root is `systemA/src/`

JARs in the binary distribution form of the system. 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 system for completeness. Once the exploratory build is successful, we are able to automatically build the system (line 6), if necessary after applying patches. (Builds are always done from the initial state of the source code form of the system.)

After the build, for all files found in the system (line 7), we collect their full file path and name, creation and modification times. 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* according to classifications presented in Section 6.2.1.

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 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. Then, (line 11) we refine the system's source code form $p'_{src}$ to consist only of the compiled Java types together with the necessary library JARs. The binary form $p'_{bin}$ is refined (line 12) to consist of the built JARs.

The refined system $p'$ (line 13) is validated (line 14) by rebuilding the system in a sandbox, outside its specific setup, while only using the files that have been identified by the method. (In practice, we use an Eclipse workspace with automatically generated configuration files (i.e., `.system` and `.classpath`).) If we were able to correctly filter out source code and detect libraries, the system successfully builds in this manner. We can add the refined system to the corpus (line 15) and run fact extraction on it (line 16).

This pseudocode is, of course, an idealized description of the process. In practice, we would execute line 4 only once per system; 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. As additional means of validation, during the step on line 10, we use visual aids to understand the completeness of builds (see Section 6.3.5 for an example).

## 6.3 Reengineering Qualitas

In this section we provide results of applying our method to the Qualitas corpus.

### 6.3.1 Details of Qualitas Content

The Qualitas curated Java code collection [175] for each included system provides both source code and binary form as they are distributed by developers. No additional effort is made to include the dependencies necessary to build/run the systems, and these are indeed often missing. The employed build mechanisms vary from system to system.

Qualitas offers the following metadata for each system: found Java types, their location in source code and binaries, metrics like LOC (Lines of Code), NCLOC (Non-comment, Non-blank Lines of Code), etc. Qualitas also provides a space-separated list of prefixes of packages of Java types: types covered by these prefixes are considered to be within system scope (or simply *system*).

We used the 20101126r release of Qualitas, which contains 106 systems. Table 6.1 shows descriptive statistics about the corpus: for each metric, minimum and maximum, first and third quartiles, median and mean are given. The table shows how many files and different extensions are there in the corpus; how many are Java files, how many of them belong to the system according to the Qualitas classification, and how many respective roots are present. The table also lists how many different Java packages are detected and how many system prefixes Qualitas metadata provides. Finally, the table shows how many JAR(s) are found and how many different build systems are detected.

**Table 6.1.** Qualitas 20101126r descriptive statistics

| | Min. | 1st Qu. | Median | Mean | 3rd Qu. | Max |
|---|---|---|---|---|---|---|
| # files | 77 | 500 | 1,102 | 2,794 | 2,845 | 66,550 |
| # extensions | 1 | 13 | 21 | 29.2 | 35 | 346 |
| # Java files | 44 | 200 | 564 | 1,453 | 1,192 | 32,550 |
| # sysJava files | 38 | 198 | 540 | 1,341 | 1,192 | 29,180 |
| # roots | 1 | 2 | 5 | 29.79 | 15 | 1,499 |
| # sysRoots | 1 | 1 | 3 | 21.27 | 8 | 1,130 |
| # packages | 2 | 21 | 37 | 130.9 | 92 | 3,620 |
| # prefixes | 1 | 1 | 1 | 1.648 | 2 | 20 |
| # JARs | 1 | 1 | 7 | 38.22 | 20 | 1,822 |
| # ANT scripts | 1 | 1 | 1 | 14.43 | 2.75 | 1,020 |
| # Maven 1.x scripts | 1 | 1 | 1 | 12.08 | 1 | 1,035 |
| # Maven 2.x scripts | 1 | 1 | 1 | 8.811 | 1 | 341 |
| # Makefile scripts | 1 | 1 | 1 | 2.679 | 1 | 135 |

### 6.3.2 Exploratory Builds

The Qualitas corpus contains 106 systems. We were able to build 86 systems, of them we had to patch 54 systems to make them build. We limited our effort per system during exploratory builds, and in some complex cases, systems could not be built within the time constraints.

We applied the following kinds of build patches (the number of patches applied of each kind is given in parentheses):

- **addJar** (197): add a JAR
- **makeDir** (98): create a necessary directory
- **addFile** (76): add a non-Java file
- **batchBuild** (34): execute several build files or targets
- **moveDir** (30): move a directory
- **patchBuildFile** (22): patch an existing build file
- **changeJDK** (8): use a specific Java version
- **addSettings** (8): add a file with properties/settings
- **patchSettings** (7): patch a file with properties/settings
- **moveFile** (6): move a non-Java file
- **addBuildFile** (6): add a build file
- **patchSrc** (5): patch a Java file
- **changeBuildV** (5): use specific build system version
- **chmod** (1): make a file executable

The most common patch operation is **addJar**: almost 200 JARs were added to 30 systems to make them build (not counting libraries that Maven downloads automatically). The next two most frequent patches, **makeDir** and **addFile** are aimed to create the file layout as the build script expects it to be. While creating new directories is needed for 20 systems, only 4 systems are responsible for the sum of added files (missing resources).

Almost half of the patched systems (25 out of 54) needs only one type of fixing, the most popular being patches of build files (6 systems), missing library JARs (5 systems), and usage of specific Java version (4 systems). The number of systems requiring 2, 3, 4, 5, and 6 kinds of patches was 9, 10, 6, 2, and 2, respectively.

We used ANT to build the majority of systems. Some systems support more than one build system (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.

In the rest of the study, we operate only on systems that we were able to build.

### 6.3.3 Identifying Core Files and Types

We heuristically classify files (and, by extension, types) into core, test, and demo by checking whether some part of the file path starts or ends with any of the following terms:

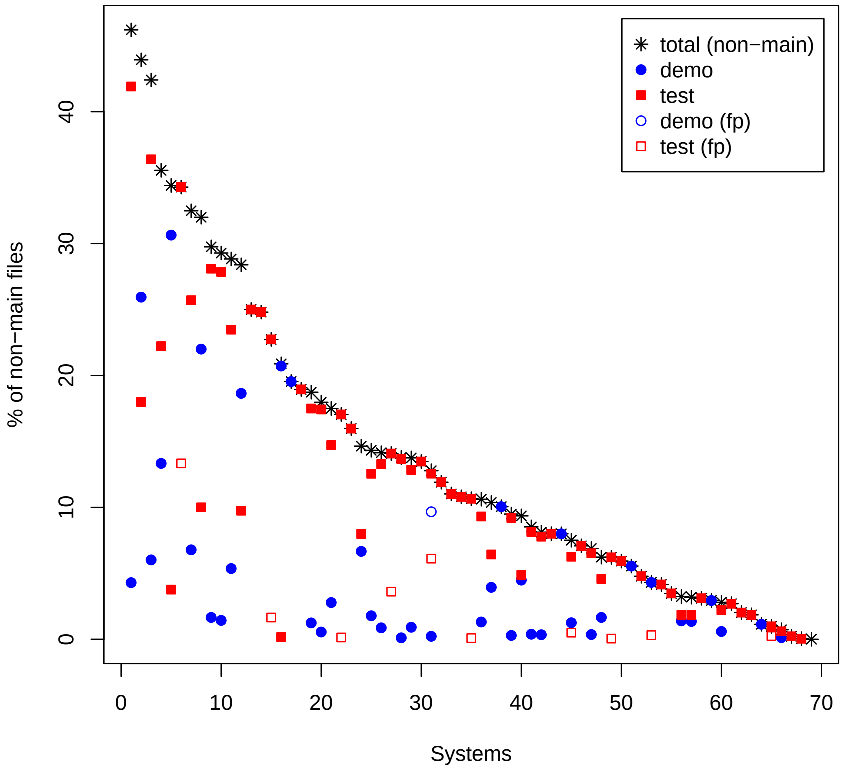
- Category *test*: test, tests, testcase, testcases, testsuite, testsuites, testing, junit[3]

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

- Category *demo*: demo, demos, tutorial, tutorials, example, examples, sample, samples

For the *test* category, we additionally check if the file name ends with either "Test," "TestCase," or "TestSuite." By default, source code is considered to be *core*.

In Fig. 6.1 we show systems that contain code other than core (there are 69 of them); they are ordered by the percentage of non-core files. The table below the



| % | Min. | 1st Qu. | Median | Mean | 3rd Qu. | Max |
|---|---|---|---|---|---|---|
| total | 0.04 | 5.36 | 10.73 | 14.14 | 19.09 | 46.2 |
| 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 |

**Figure 6.1.** Distribution of non-core code

figure contains descriptive statistics about percentage of non-core files per system. We observe that an average system contains 10.73 % non-core files. Usually those files are tests.

During validation phase (see §6.3.6), we discovered that our classification produced several false positives: we show them as non-filled symbols in Fig. 6.1.

### 6.3.4 Identifying System Namespaces

We reuse the classification of code that Qualitas considers part of the system (given as a list of package prefixes). Since our classification complements the one of Qualitas, we were interested in understanding the difference. For that, we collect in Table 6.2 the prefixes that appear in the core files but that are not covered by the Qualitas' list of system prefixes. We list those prefixes along with some additional information: number of Java files with packages corresponding to such prefix, what percent of all Java files in a system they constitute, how many of these files contribute to built JARs, and for comparison, how many of these files contribute to the binaries provided by Qualitas. Finally, we document, whether we consider—in deviation from Qualitas—those package prefixes belonging to the system ("in" the system) and why. Entries in the table are ordered by their impact on a system, i.e., percentage of affected Java files. We cut the table at threshold limit of 5 % under which we did not make an intelligent decision—considering the impact weak—but en masse excluded those prefixes from being system. (The cut tail of the table contains 65 prefixes from 32 systems with a median of "% of all" being 0.55.) As we see, Qualitas metadata provides good coverage of system code and does not leave out much.

For each prefix in the table, we investigate the role of the respective packages in the system, and in few cases we make a decision (marked with '✓') to classify the prefix as system, where we feel strongly that it is appropriate to do so. For instance, we additionally count as system two prefixes of system **jsXe**, an XML editor, which contain the editor's features for different views of XML files (tree-like versus textual). We also make decisions (marked with '?') to include prefixes, where opinions could differ. For instance, we count as system code certain extensions or contributions to systems.

There are two interesting observations about the table.

*Note on default packages*

We assume that the code in the default package is never part of the system. Such code violates naming conventions of Java and can create clashes and ambiguities in the name space of types. (Since Qualitas uses a space-separated list for storing the prefixes, there is even no possibility to classify the default package as system.) But default packages are not rare.

Consider the table entry for **webmail** system, a web mail server: out of 24 files within the default package—19 are plugins. They reside in the directory `webmail/src/net/wastl/webmail/plugins/`, which is part of the package hierarchy. `net.wastl.webmail` is a system prefix. Together this suggests that either the default package is an omission, or developers use mixed semantics for directories.

**Table 6.2.** Prefixes of core code that are not listed by Qualitas metadata (cut at 5 %)

| System | Prefix | # files | % of all | % in | covered by | | isIn | Comment |
|---|---|---|---|---|---|---|---|---|
| | | | | built JARs | built JARs | binaries | | |
| fitjava | eg.* | 21 | 30.88 | 100.00 | 21 | 21 | | Example(s) |
| mvnforum | net.myvietnam.* | 174 | 24.96 | 100.00 | 174 | 174 | | Example(s) |
| webmail | <*default*> | 24 | 21.62 | 0.00 | 0 | 0 | | Default package |
| jsXe | gnu.regexp | 27 | 20.61 | 100.00 | 27 | 0 | | 3rd party sources |
| jext | <*default*> | 101 | 18.17 | 0.00 | 0 | 0 | | Default package |
| jsXe | treeview.* | 20 | 15.27 | 100.00 | 20 | 0 | ✓ | Feature(s) |
| gt2 | net.opengis.* | 801 | 14.95 | 100.00 | 801 | 801 | ? | Extension(s) |
| jFin_DateMath | <*default*> | 8 | 12.70 | 0.00 | 0 | 0 | | Default package |
| azureus | org.bouncycastle.* | 384 | 11.83 | 100.00 | 384 | 384 | | 3rd party sources |
| jext | org.gjt.sp.jedit | 61 | 10.97 | 100.00 | 61 | 61 | | 3rd party sources |
| mvnforum | org.mvnforum.* | 75 | 10.76 | 0.00 | 0 | 0 | ? | Contribution(s) |
| itext | com.itextpdf.rups.* | 46 | 10.29 | 0.00 | 0 | 0 | ? | Extension(s) |
| jsXe | org.syntax.* | 13 | 9.92 | 100.00 | 13 | 0 | | 3rd party sources |
| jrefactory | <*default*> | 127 | 9.58 | 0.00 | 0 | 0 | | Default package |
| fitlibraryforfitnesse | fitbook.* | 64 | 8.66 | 100.00 | 64 | 64 | | Example(s) |
| jsXe | sourceview.* | 11 | 8.40 | 100.00 | 11 | 0 | ✓ | Feature(s) |
| fitlibraryforfitnesse | fit.* | 62 | 8.39 | 100.00 | 62 | 62 | | 3rd party sources |
| mvnforum | com.mvnsoft.* | 54 | 7.75 | 98.15 | 53 | 53 | ? | Contribution(s) |
| roller | org.apache.jsp.* | 35 | 5.71 | 0.00 | 0 | 0 | | Internals of app.server |
| jhotdraw | net.n3.nanoxml | 23 | 5.10 | 100.00 | 23 | 23 | | 3rd party sources |
| compiere | org.apache.ecs.* | 126 | 5.06 | 100.00 | 126 | 126 | | 3rd party sources |

Another example is **jext** system, a programmer's text editor. It has 101 files within the default package, of them 99 constitute 11 different plugins. Yet, **jext** has one more plugin, which uses non-default packages and is therefore included as system.

*Note on third party sources*

We see that it is usual practice among systems to include large amounts of 3rd-party source code. We do not count it as system code, but without clone detection, one cannot be sure that 3rd-party code was not modified/adjusted and had not become an integral part of the system.

### 6.3.5 Builds

In order to understand the result of a build, we measure the percentage of Java types (system and core) found in built JARs. In Table 6.3 we list systems with incomplete builds, sorted by percentage of types not found in built JARs. Along with basic information—number of all Java types, percentage of them non-compiled—the table includes the number of roots and packages completely or partially missing in built JARs. We also show a comparison with coverage of Java types by binaries included in Qualitas, see column "Compare to Q": "=" means the same coverage and ">" means that our build result misses more types than the binaries included in Qualitas. We cut the table at 5 % of non-packed types, the tail of the table contains 21 more systems with median of percentage of non-packed types equal to 1.05 %.

An asterisk next to the system name means that types not found in built JARs were nonetheless compiled (values in columns 3 and 4 differ). Careful reading of build scripts revealed that this is due to white- or blacklisting of files or directories done in ANT scripts.

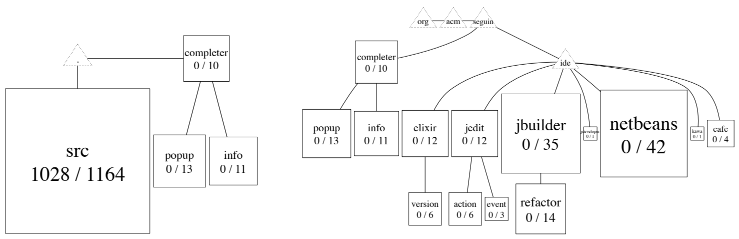
We tried to identify the reason for missing types. Based on several investigated cases, we concluded that completely missing packages or roots are an indication of omitted build goals. These packages or roots then typically correspond to modular code, like an optional feature. We automatically identify such cases.

For instance, the **james** system provides libraries for Internet mail communication. All non-built types are in the root hierarchy that starts with a directory `proposals/` and the build script does not compile it.[4] A similar situation is with the **checkstyle** system providing support for coding standards: all its non-built sources are in the root hierarchy starting with the directory `contrib/`, which is also not compiled by the build script.

[4] One partially compiled root upon closer inspection revealed to contain a type with the same qualified name as a type from `src/` directory, thus being counted as compiled.

**Table 6.3.** Incomplete builds (cut at 5 %)

| System | types | | | Compare to Q | # non-packed roots | | # non-packed packages | | Reason |
|---|---|---|---|---|---|---|---|---|---|
| | # total | % non-compiled | % non-packed | | completely | partially | completely | partially | |
| mvnforum | 510 | 36.08 | 36.08 | = | 3 | 0 | 11 | 0 | Feature |
| trove* | 32 | 0.00 | 34.38 | = | 0 | 1 | 2 | 0 | Feature |
| james | 419 | 34.13 | 34.13 | = | 1 | 4 | 4 | 3 | ??? |
| checkstyle | 359 | 25.63 | 25.63 | = | 2 | 0 | 6 | 0 | Feature |
| log4j* | 242 | 11.16 | 23.14 | > | 8 | 1 | 4 | 4 | ??? |
| gt2 | 5234 | 17.90 | 17.90 | > | 34 | 8 | 128 | 25 | ??? |
| jrefactory | 1199 | 14.18 | 14.18 | > | 3 | 1 | 14 | 0 | Feature |
| itext | 447 | 10.29 | 10.29 | = | 1 | 0 | 10 | 0 | Feature |
| jung | 358 | 9.78 | 9.78 | > | 2 | 0 | 6 | 0 | Feature |
| hsqldb* | 428 | 3.27 | 9.58 | > | 0 | 1 | 3 | 1 | ??? |
| nakedobjects | 2176 | 8.96 | 8.96 | = | 9 | 0 | 33 | 3 | Feature |
| jext | 365 | 8.77 | 8.77 | > | 1 | 0 | 3 | 0 | Feature |
| derby* | 1779 | 1.41 | 7.31 | = | 3 | 5 | 17 | 27 | ??? |
| jag | 130 | 6.15 | 6.15 | = | 1 | 0 | 1 | 1 | Feature |
| proguard | 562 | 5.87 | 5.87 | > | 0 | 2 | 0 | 10 | ??? |



(a) Root forest (b) Package forest

**Figure 6.2.** Non-built types of **jrefactory**: triangles represent empty packages/directories; squares show how many types are compiled in a directory/package.

Another example is the **jrefactory** refactoring tool. Fig. 6.2 presents the forests of the non-built roots and packages. The root forest shows that there are several completely non-built directories, while some code wasn't built in the main root, `src/`. Package forest shows that a large part of the non-built packages has a common prefix that apparently contains the code for integration with IDEs. Inspection of the build script reveals conditional builds: if certain classes of different IDEs are not found in the classpath, the corresponding parts of the code are excluded from compilation. The build script operates only on `src/` which explains non-built types outside it.

In such manner we inspected the incomplete builds to decide whether to accept the result or to attempt to increase the build coverage.

### 6.3.6 Validation

To validate our effort, we automatically converted the built and refined systems into Eclipse projects, based on the data collected during the build. Successful compilation of a system in Eclipse shows that the classpath and the filtering of source code are correct. In such manner, we successfully validated 79 systems (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** system, different parts of the code are compiled with different versions of a library JAR. While in an ANT script it is possible to change the classpath during compilation, in Eclipse, the classpath is a static global setting per system.

In particular, we validated our classification of files: if the exported code would not compile due to its dependencies, we would revise the classification of core code (see empty squares and circles in Fig. 6.1). In those few cases when a system'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 `<system>-sys.jar`. Since some systems use 3rd-party source code, we also ship those compiled classes in a library JAR with the name `<system>-nonSys.jar`

### 6.3.7 Automated Fact Extraction

To facilitate fact extraction for the corpus user, we offer (on the paper's website) a basic fact-extractor based on RECODER [89]. The extractor runs automatically on the whole corpus and checks for absence of syntax errors and unresolved references. It can be easily extended.

## 6.4 Threats to Validity

From the theoretical point of view, there are two threats: to construct validity and to content validity.

For instance, our decisions on what consistutes a system, its core part, are subject to the threat to construct validity, because, arguably, tests can be considered to be part of the system. We justify our decisions by taking an operational approach to definition—a system is what runs in production, which usually excludes tests.

The selection of the projects is subject to the threat to content validity: is the subset representative? We consider the Qualitas corpus to be a well-thought selection from the group of Java projects that are usually characterized as "well-developed" or "real-world" applications. Therefore, it does not cover the whole spectrum of Java projects (for instance, student projects or throw-away prototypes are not included), however, the covered part is usually the main focus of the practical, applied, research.

Technically, we can identify the following threats:

- The exploratory builds are a manual process: In each case we decided, which build scripts to execute, in which order, etc.
- The heuristic underlying our code purpose classification may give false positives (e.g., when the application domain of a system is itself testing) or false negatives (e.g., by relying on the particular file naming convention).
- The accuracy of the data in the refined corpus depends on the correctness of our toolchain. The toolchain was tested to eliminate bugs.

## 6.5 Related Work

*Corpora*

The related work on corpus engineering is summarized in Table 6.4. We list main reference for each effort, date of latest release/update, size of the corpus and languages of systems in it; what is the main purpose of the effort and in column Acc. we note, whether the corpus is accessible freely. The bibliographical reference (the first column) also include the web links where the corpora can be found.

The DaCapo project facilitates benchmarking: it provides a harness to measure running of the corpus. The systems in the binary form are provided within the DaCapo JAR while sources of the systems come with the tool's source code. The tool is freely available for download.

**Table 6.4.** Corpus (re-)engineering efforts

| Ref. | Name | Acc. | Date | Size | Lang. | Src | Bin | Depend. | Purpose |
|---|---|---|---|---|---|---|---|---|---|
| [29] | DaCapo | + | 12/2009 | 14 | Java | + | + | + | Tool for Java benchmarking |
| [77] | FOSSology | ± | 10/2012 | 16 | C/C++ | + | | | Framework for analyzing licenses |
| [175] | Qualitas | – | 04/2012 | 111 | Java | + | + | | Collection for empirical studies of code artifacts |
| [56] | SIR | – | 05/2012 | 73 | Java, C, C#, C++ | + | | + | Infrastructure for controlled experimentation with testing techniques |
| [9] | SPEC | $ | 08/2012 | 17 | Java, C, Fortran, ? | + | | + | Collection of realistic, standardized performance tests |
| [124] | Sourcerer | ± | 04/2010 | 18,826 | Java | + | | ± | Infrastructure for large-scale source code analysis |
| [55] | TraceLab | + | 07/2012 | 5 | Java | ± | | ± | Framework for feature location research |

The FOSSology project focuses on analyzing licenses in source code, providing its tools freely—one can download and use them on one's systems locally. There is a demo repository available separately[5].

Qualitas has two main distribution forms: recent version release contains 111 systems and evolution release contains 486 versions of 14 systems that has at least 10 versions per system. There is also a complete release containing 661 versions of 111 systems. Download links are available upon request.

The Software-artifact Infrastructure Repository (SIR) provides infrastructure to support controlled experimentation with testing and regression testing techniques. Access to the tools and the corpus is available after registration with statement of intent. Systems in the corpus contain real or seeded faults.

Standard Performance Evaluation Corporation is a non-profit organisation providing SPECmarks, standardized suites of source code, for performance testing. There are 17 benchmarks available for a non-profit/educational price ranging from $50 to $900.

The Sourcerer project provides infrastructure for large-scale source code analysis as well as a prepared repository of more than 18,000 systems (of them non-empty 13,241 systems). The full corpus is available upon request with statement of intent.

[5] `https://fossology.ist.unomaha.edu/`

From the description of the process, given in [124] and on the website[6], it is not clear whether it contains resolved dependencies.

TraceLab is a framework for creating, running and sharing experiments using a visual modeling environment. Along with the tool, authors provide datasets containing issues, associated functionality (methods), queries and execution traces. This information was extracted from issue tracking systems, version repositories, and code itself. According to the description of the process[7], compilable source code of the systems was used during creation of datasets, but datasets themselves do not seem to contain it.

There are few other corpora like TraceLab that can be seen as meta-corpora, providing metadata about the systems, but not the systems themselves. The FLOSSmole project [93] focuses on metadata about systems from different online repositories, such as Sourceforge, GNU Savannah, Google Code, GitHub, and others. Metadata includes a name of a system, description, programming languages used, number of developers, date of creation, etc. All the data is freely available as textual files and SQL dumps. The PROMISE repository [151] contains datasets to facilitate research on defect and effort prediction: extracted bug reports, modification requests and their impact on the code, as well as values of various source code metrics. All data in the repository is freely accessible.

*Build systems*

Research in the area of build systems is diverse. Complexity of build systems and their evolution, as well as correlation of these characteristics to the same of source code has been studied for Make, ANT, and Maven build systems [11, 133]. The Make system was also studied for the variability model it provides [127] and its correctness [145]. There are studies on the overhead of maintaining build systems and its reasons [118, 134]. We are not aware of any effort on understanding what it takes to make systems build: types of errors, missing libraries, needed patches, etc. Though studying the building process is not a central research question of our effort, we see our work as contributing towards this area of research.

## 6.6 Conclusion

We presented an approach to (re-)engineering a corpus of software systems. The approach transforms a raw source and binary distribution into a uniform and consolidated representation of each system. Both source-code- and byte-code-based fact extraction can be run automatically on the resulting corpus.

The need for the approach was derived from a literature survey on empirical software engineering efforts.

[6] `http://sourcerer.ics.uci.edu/tutorial.html`
[7] `http://www.cs.wm.edu/semeru/data/benchmarks/`

We applied the reengineering approach to the Qualitas corpus, thus validating its effectiveness as well as potential utility in developing reliable curated code collections and building confidence in these collections with their users and developers. The resulting corpus is available to the public together with a sample fact extraction mechanism.

There are several directions for future work:

- An intelligent and reproducible technique of automatic attempts to build systems with automatic classification of errors in order to identify patches.
- A refinement of the heuristic for identification of core files based on observations that we made, e.g.: discrepancy between Java packages and file path (usually a sign of non-core files); imports in test files (indicating usage of a testing framework).
- A clone detection analysis of included 3rd-party source code which determines whether such code was modified and integrated into the host system.
- A deeper analysis of builds for improved confidence in the builds. For instance, How clean is the state that we start from? What modifications occur during the build? What is the nature of differences between compiled classes and built JARs?

# Part IV

# Conclusion

# 7 Conclusion

In this chapter, we summarize the outcome of the work presented in the thesis. We revisit the contributions briefly stated in Introduction, Section 1.4, and substantiate them based on the reported studies. We also outline open challenges and future work.

## 7.1 Summary

Let us revisit the research goals of the thesis posed in Introduction, Section 1.2.3 and link them back to the studies reported in the thesis. We submitted in our work to:

*1. Develop and apply techniques to empirically analyze actual usage of languages.*

Part II of the thesis reports two extensive empirical studies. In a deep and systematic manner, we have analyzed usage of a privacy language among websites on the Internet and usage of APIs among open-source projects. Despite the significant differences in the analyzed languages (application domains, language design), we have shown that the approach for such analysis is essentially the same.

In the study of P3P language, we have utilized a broad and diverse range of techniques, combining basic measures of language usage (such as footprint and coverage) with Software Engineering methods (validity analysis, clone detection), for the purpose of revealing the status of language adoption.

In the study of APIs, we have applied the same basic measures of language usage (such as footprint and coverage) to capture and classify features of APIs that allow profiling and characterizing the projects using APIs. Based on this dual relationship between APIs and projects, we have developed a catalogue of exploration insights and implemented a tool for navigation of the code base for the benefit of API consumers as well as API designers.

We also submitted in our work to:

*2. Understand the usage of empirical evidence in Software Engineering research.*

Part III of the thesis describes our effort on establishing the current state of research in Software Engineering with respect to the usage of empirical evidence; it also operationally summarizes our experience in corpus engineering.

We have performed two pilot studies and the main literature survey, to work out the appropriate methodology for the task of analyzing published research with respect to used empirical evidence. We have developed and adjusted the classification scheme to capture that usage and summarize it.

We have found that use of project-based corpora is frequent in the practically oriented areas of Software Engineering such as program comprehension, maintenance, reverse engineering, and re-engineering. At the same time, we have found that there is no explicit methodology for the common task of collecting a corpus as well as no apparent adoption of standardized corpora in the research community. Based on our experience gained during language studies, we have suggested a method of corpus (re-)engineering and demonstrated its application on an existing collection of Java programs.

## 7.2 Future Work

Below we identify several lines of work stemming from the current thesis that we see both interesting and practically appealing.

In the area of language usage, we see the following possible continuations:

- Active feedback: Turning the findings of language usage analysis into the basis for close collaboration with language designers: working together on the next version of a language (e.g., in the privacy domain) or a new version of a library (e.g., particular Java API).
- Usage profiling: Detecting types of language usage (e.g., with the help of machine learning or statistical inference); linking them with particular user needs based either on metadata about the application area or via user studies.
- Replication studies: Applying the identified principles in studies of languages relatively similar to the analyzed in this thesis (e.g., usage of the SQL language or Haskell libraries) and relatively distant from them (e.g., languages of real-time systems like car engine control system or medical systems).

In the area of corpus engineering, we see the following possible continuations:

- Seeking explanations: One of the most intriguing open questions left for us is why standardized corpora are not adopted by the community. Testing hypotheses by qualitative studies (e.g., questionnaires) to find out whether researchers are aware of such corpora or by experiments to assess whether the overhead involved in corpora adoption is impractical in the end.
- Involving community: Obliging authors to use a suggested corpus in their submissions to a certain track of a conference—similar to Mining Challenge hosted by the International Working Conference on Mining Software Repositories—but in other areas of Software Engineering, such as program comprehension, maintenance, reverse engineering, and re-engineering.

Generally, in the area of Empirical Software (Language) Engineering, we strongly believe that the time has come to inspect the existing practices and detect common places in research that are being addressed in ad hoc manner and require explicit methodology. We also believe that the missing methodology can be and should be distilled by the inductive approach from the individual solutions of a repetitive problem—thus leading to research-driven research.

# Own Publications

1. Ralf Lämmel and Ekaterina Pek. Vivisection of a non-executable, domain-specific language - Understanding (the usage of) the P3P language. In *Proceedings of the 18th International Conference on Program Comprehension (ICPC)*, pages 104–113. IEEE Computer Society, 2010.
2. Jean-Marie Favre, Dragan Gašević, Ralf Lämmel, and Ekaterina Pek. Empirical language analysis in software linguistics. In Brian A. Malloy, Steffen Staab, and Mark van den Brand, editors, *Proceedings of the 3rd International Conference on Software Language Engineering (SLE)*, volume 6563 of *Lecture Notes in Computer Science*, pages 316–326. Springer, 2011.
3. Ralf Lämmel, Rufus Linke, Ekaterina Pek, and Andrei Varanovich. A framework profile of .NET. In Martin Pinzger, Denys Poshyvanyk, and Jim Buckley, editors, *Proceedings of the 18th Working Conference on Reverse Engineering (WCRE)*, pages 141–150. IEEE Computer Society, 2011.
4. Ralf Lämmel, Ekaterina Pek, and Jürgen Starek. Large-scale, AST-based API-usage analysis of open-source Java projects. In William C. Chu, W. Eric Wong, Mathew J. Palakal, and Chih-Cheng Hung, editors, *Proceedings of the 26th Symposium on Applied Computing (SAC)*, pages 1317–1324. ACM, 2011.
5. Ralf Lämmel and Ekaterina Pek. Understanding privacy policies - A study in empirical analysis of language usage. *Empirical Software Engineering*, 18(2):310–374, 2013.
6. Coen De Roover, Ralf Lämmel, and Ekaterina Pek. Multi-dimensional exploration of API usage. In *Proceedings of the 21th International Conference on Program Comprehension (ICPC)*, 2013.
7. Ekaterina Pek and Ralf Lämmel. A literature survey on empirical software engineering research. 10 pages. Work under submission.
8. Bogdan Vasilescu, Alexander Serebrenik, Tom Mens, Mark van den Brand, and Ekaterina Pek. How healthy are Software Engineering conferences? *Science of Computer Programming*, 2013. Pending minor revision.

# References

9. Standard Performance Evaluation Corporation benchmark suites.
10. Mithun Acharya, Tao Xie, Jian Pei, and Jun Xu. Mining API patterns as partial orders from source code: from usage scenarios to specifications. In *ESEC-FSE '07: Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 25–34. ACM, 2007.
11. Bram Adams, Herman Tromp, Kris De Schutter, and Wolfgang De Meuter. Design recovery and maintenance of build systems. In *ICSM*, pages 114–123, 2007.
12. Steve Adolph, Wendy Hall, and Philippe Kruchten. A methodological leg to stand on: Lessons learned using grounded theory to study software development. In *CASCON*, pages 13:166–13:178, 2008.
13. Rakesh Agrawal, Paul Bird, Tyrone Grandison, Jerry Kiernan, Scott Logan, and Walid Rjaibi. Extending relational database systems to automatically enforce privacy policies. In *ICDE '05: Proceedings of the 21st International Conference on Data Engineering*, pages 1013–1022. IEEE Computer Society, 2005.
14. Rakesh Agrawal, Jerry Kiernan, Ramakrishnan Srikant, and Yirong Xu. An XPath-based preference language for P3P. In *Proceedings of WWW 2003*, pages 629–639. ACM, 2003.
15. Rakesh Agrawal, Jerry Kiernan, Ramakrishnan Srikant, and Yirong Xu. XPref: a preference language for P3P. *Computer Networks*, 48(5):809–827, 2005.
16. Tiago Alves and Joost Visser. Metrication of SDF grammars. Technical Report DI-PURe-05.05.01, Universidade do Minho, 2005.
17. Paul Ashley. Enforcement of a P3P privacy policy. In *Proceedings of the 2nd Australian Information Security Management Conference, Securing the Future*, pages 11–26. School of Computer and Information Science, Edith Cowan University, Western Australia, 2004.
18. Magdalena Balazinska, Ettore Merlo, Michel Dagenais, Bruno Laguë, and Kostas Kontogiannis. Advanced Clone-Analysis to Support Object-Oriented System Refactoring. In *Proceedings of the Seventh Working Conference on Reverse Engineering (WCRE'00)*, pages 98–107. IEEE Computer Society, 2000.
19. Margaret Burnett Barbara G. Ryder, Mary Lou Soffa. The impact of software engineering research on modern progamming languages. *ACM Transactions on Software Engineering and Methodology*, 14(4):431–477, October 2005.

20. Thiago Tonelli Bartolomei, Krzysztof Czarnecki, and Ralf Lämmel. Swing to SWT and back: Patterns for API migration by wrapping. In *ICSM*, pages 1–10, 2010.
21. Thiago Tonelli Bartolomei, Krzysztof Czarnecki, and Ralf Lämmel. Swing to SWT and back: Patterns for API migration by wrapping. In *26th IEEE International Conference on Software Maintenance (ICSM 2010)*, pages 1–10. IEEE, 2010.
22. Thiago Tonelli Bartolomei, Krzysztof Czarnecki, Ralf Lämmel, and Tijs van der Storm. Study of an API migration for two XML APIs. In *SLE*, pages 42–61, 2010.
23. Victor Basili, Gianluigi Caldiera, and H. Dieter Rombach. The goal question metric approach, 1994.
24. Gareth Baxter, Marcus Frean, James Noble, Mark Rickerby, Hayden Smith, Matt Visser, Hayden Melton, and Ewan Tempero. Understanding the shape of Java software. In *Proceedings of OOPSLA 2006*, pages 397–412. ACM, 2006.
25. Gareth Baxter, Marcus R. Frean, James Noble, Mark Rickerby, Hayden Smith, Matt Visser, Hayden Melton, and Ewan D. Tempero. Understanding the shape of java software. In *OOPSLA*, pages 397–412, 2006.
26. Ira D. Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant'Anna, and Lorraine Bier. Clone detection using abstract syntax trees. In *Proceedings of ICSM 1998*, page 368. IEEE Computer Society, 1998.
27. Lise Jensen Bertil Ekdahl. The difficulty in communicating with computers. In *Interactive Convergence: Critical Issues in Multimedia*, chapter 2. Inter-Disciplinary Press, 2005.
28. Gavin M. Bierman, Erik Meijer, and Wolfram Schulte. The Essence of Data Access in C*omega*. In *ECOOP 2005 - Object-Oriented Programming, 19th European Conference, Proceedings*, volume 3586 of *LNCS*, pages 287–311. Springer, 2005.
29. S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA*, pages 169–190, 2006. `http://dacapobench.org/`.
30. E. T. Bonelli and J. Sinclair. Corpora. In Keith Brown, editor, *Encyclopedia of language and linguistics*. Elsevier Science, 2006.
31. Lee Breslau, Pei Cao, Li Fan, Graham Phillips, and Scott Shenker. Web caching and Zipf-like distributions: Evidence and implications. In *INFOCOM*, pages 126–134, 1999.
32. Marcel Bruch, Martin Monperrus, and Mira Mezini. Learning from examples to improve code completion systems. In *ESEC/SIGSOFT FSE*, pages 213–222, 2009.
33. Andrea Brühlmann, Tudor Gîrba, Orla Greevy, and Oscar Nierstrasz. Enriching reverse engineering with annotations. In *MoDELS*, pages 660–674, 2008.
34. R. J. Chevance and T. Heidet. Static profile and dynamic behavior of COBOL programs. *SIGPLAN Notices*, 13(4):44–57, 1978.
35. Kelvin Choi and Ewan D. Tempero. Dynamic measurement of polymorphism. In *Proceedings of the Thirtieth Australasian Computer Science Conference (ACSC2007)*, volume 62 of *CRPIT*, pages 211–220. Australian Computer Society, 2007.
36. Aaron Clauset, Cosma Rohilla Shalizi, and Mark E. J. Newman. Power-law distributions in empirical data. *SIAM Review*, 51(4):661–703, 2009.
37. Christian S. Collberg, Ginger Myles, and Michael Stepp. An empirical study of Java bytecode programs. Technical Report TR04-11, University of Arizona, 2004.
38. Christian S. Collberg, Ginger Myles, and Michael Stepp. An empirical study of Java bytecode programs. *Software, Practice & Experience*, 37(6):581–641, 2007.

39. Giulio Concas, Michele Marchesi, Alessandro Murgia, Sandro Pinna, and Roberto Tonelli. Assessing traditional and new metrics for object-oriented systems. In *Proceedings of the 2010 ICSE Workshop on Emerging Trends in Software Metrics, WETSoM '10*, pages 24–31. ACM, 2010.
40. Robert P. Cook and Insup Lee. A contextual analysis of Pascal programs. *Software, Practice & Experience*, 12(2):195–203, 1982.
41. Harris M. Cooper. *The integrative research review: A systematic approach*, volume 2 of *Applied social research methods series*. Sage, 1984.
42. Lorrie Faith Cranor. *Web Privacy with P3P*. O'Reilly & Associates, 2002.
43. Lorrie Faith Cranor. RE: COMMENTS ON FEDERAL TRADE COMMISSION PRELIMINARY STAFF REPORT. Protecting Consumer Privacy in an Era of Rapid Change: A Proposed Framework for Businesses and Policymakers, 2011. Available online at `http://www.ftc.gov/os/comments/privacyreportframework/00453-58003.pdf`.
44. Lorrie Faith Cranor, Serge Egelman, Steve Sheng, Aleecia M. McDonald, and Abdur Chowdhury. P3P deployment on websites. *Electronic Commerce Research and Applications*, 7(3):274–293, 2008.
45. David Crystal. *The Cambridge Encyclopedia of Language, Second Edition*. Cambridge University Press, 2005.
46. Hamish Cunningham. A definition and short history of language engineering. *Journal of Natural Language Engineering*, 5:1–16, 1999.
47. Krzysztof Cwalina and Brad Abrams. *Framework design guidelines. Conventions, idioms, and patterns for reusable .NET libraries*. Addison-Wesley, 2009.
48. Melis Dagpinar and Jens H. Jahnke. Predicting maintainability with object-oriented metrics - An empirical comparison. In *Proceedings of the 10th Working Conference on Reverse Engineering, WCRE '03*, pages 155–164. IEEE, 2003.
49. Marco D'Ambros, Michele Lanza, and Romain Robbes. An extensive comparison of bug prediction approaches. In *MSR*, pages 31–41, 2010. `http://bug.inf.usi.ch/`.
50. Suresh Jagannathan David Gelernter. *Programming Linguistics*. MIT Press, 1990.
51. Brian de Alwis and Gail C. Murphy. Answering conceptual queries with Ferret. In *ICSE*, pages 21–30, 2008.
52. Antonio Soares de Azevedo Terceiro, Manoel G. Mendonça, Christina Chavez, and Daniela S. Cruzes. Understanding structural complexity evolution: A quantitative analysis. In *CSMR*, pages 85–94, 2012.
53. Coen De Roover, Carlos Noguera, Andy Kellens, and Viviane Jonckers. The SOUL tool suite for querying programs in symbiosis with Eclipse. In *PPPJ*, pages 71–80, 2011.
54. Jens Dietrich, Catherine McCartin, Ewan D. Tempero, and Syed M. Ali Shah. Barriers to modularity - an empirical study to assess the potential for modularisation of java programs. In *6th International Conference on the Quality of Software Architectures, QoSA 2010, Proceedings*, volume 6093 of *LNCS*, pages 135–150. Springer, 2010.
55. Bogdan Dit, Evan Moritz, and Denys Poshyvanyk. A TraceLab-based solution for creating, conducting, and sharing feature location experiments. In *ICPC*, pages 203–208, 2012.
56. Hyunsook Do, Sebastian Elbaum, and Gregg Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empir. Softw. Eng.*, 10(4):405–435, 2005. `http://sir.unl.edu/`.
57. Serge Egelman, Lorrie Faith Cranor, and Abdur Chowdhury. An analysis of P3P-enabled Web sites among top-20 search results. In Mark S. Fox and Bruce Spencer, editors, *ICEC*, volume 156 of *ACM International Conference Proceeding Series*, pages 197–207. ACM, 2006.

58. Daniel S. Eisenberg, Jeffrey Stylos, Andrew Faulring, and Brad A. Myers. Using association metrics to help users navigate API documentation. In *VL/HCC*, pages 23–30, 2010.
59. Daniel S. Eisenberg, Jeffrey Stylos, and Brad A. Myers. Apatite: A new interface for exploring APIs. In *CHI*, pages 1331–1334, 2010.
60. Eva Van Emden and Leon Moonen. Java quality assurance by detecting code smells. In *WCRE*, pages 97–108, 2002.
61. Raimar Falke, Pierre Frenzel, and Rainer Koschke. Empirical evaluation of clone detection using syntax suffix trees. *Empirical Softw. Engg.*, 13:601–643, December 2008.
62. Jean-Marie Favre. Languages evolve too! changing the software time scale. In IEEE, editor, *8th Interntational Workshop on Principles of Software Evolution, IWPSE*, September 2005.
63. Norman E. Fenton and Shari Lawrence Pfleeger. *Software Metrics: A Rigorous and Practical Approach*. PWS Publishing Co., 2nd edition, 1996.
64. Kathi Fisler, Shriram Krishnamurthi, and Daniel J. Dougherty. Embracing policy engineering. In *Proceedings of the FSE/SDP workshop on Future of software engineering research*, FoSER '10, pages 109–110. ACM, 2010.
65. Francesca Arcelli Fontana, Andrea Caracciolo, and Marco Zanoni. DPB: A benchmark for design pattern detection tools. In *CSMR*, pages 235–244, 2012.
66. Jill Freyne, Lorcan Coyle, Barry Smyth, and Padraig Cunningham. Relative status of journal and conference publications in computer science. *Commun. ACM*, 53(11):124–132, November 2010.
67. E. Fudge. Glossematics. In Keith Brown, editor, *Encyclopedia of language and linguistics*. Elsevier Science, 2006.
68. Hristo Georgiev. *Language Engineering*. Continuum, 2007.
69. Malcom Gethers, Bogdan Dit, Huzefa H. Kagdi, and Denys Poshyvanyk. Integrated impact analysis for managing software changes. In *ICSE*, pages 430–440, 2012.
70. Kambiz Ghazinour and Ken Barker. Capturing P3P semantics using an enforceable lattice-based structure. In *Proceedings of the 4th International Workshop on Privacy and Anonymity in the Information Society*, PAIS '11, pages 4:1–4:6. ACM, 2011.
71. Joseph (Yossi) Gil and Itay Maman. Micro patterns in Java code. In *Proceedings of OOPSLA 2005*, pages 97–116. ACM, 2005.
72. Barney G. Glaser. *Doing Quantitative Grounded Theory*. Sociology Press, 2008.
73. Barney G. Glaser and Anselm L. Strauss. *Awareness of Dying*. Aldine Pub., 1965.
74. Barney G. Glaser and Anselm L. Strauss. *The Discovery of Grounded Theory: Strategies for Qualitative Research*. Aldine Pub., 1967.
75. Gene V. Glass, Barry McGaw, and Mary Lee Smith. *Meta-analysis in social research*. Sage, Beverly Hills, CA, 1981.
76. Robert L. Glass, Iris Vessey, and Venkataraman Ramesh. Research in software engineering: An analysis of the literature. *Inform. Software Tech.*, 44(8):491–506, 2002.
77. Robert Gobeille. The FOSSology project. In *MSR '08: Proceedings of the 2008 international working conference on Mining software repositories*, pages 47–50, New York, NY, USA, 2008. ACM.
78. Jesús M. González-Barahona and Gregorio Robles. On the reproducibility of empirical software engineering studies based on data retrieved from development repositories. *Empir. Softw. Eng.*, 17(1-2):75–89, 2012.
79. J.B. Goodenough. The comparison of programming languages: A linguistic approach. *ACM/CSC-ER*, 1968.

80. Yann-Gaël Guéhéneuc. PMARt: Pattern-like micro architecture repository. In *EuroPLoP Focus Group on Pattern Repositories*, 2007. `http://www.ptidej.net/download/pmart/`.
81. Jurriaan Hage and Peter van Keeken. Neon: A library for language usage analysis. In *Software Language Engineering, First International Conference, SLE 2008*, volume 5452 of *LNCS*, pages 35–53. Springer, 2009.
82. Michael Hahsler. A quantitative study of the adoption of design patterns by open source software developers. In *Free/Open Source Software Development, IGP*, pages 103–123, 2004.
83. Michael Hahsler and Stefan Koch. Discussion of a large-scale open source data collection methodology. *Hawaii International Conference on System Sciences*, 7:197b, 2005.
84. A. Hardie and T. McEnery. Statistics. In Keith Brown, editor, *Encyclopedia of language and linguistics*. Elsevier Science, 2006.
85. Edwin Hautus. Improving Java Software Through Package Structure Analysis. In *Proceedings of the 6th IASTED International Conference Software Engineering and Applications*, 2002.
86. Katia Hayati and Martín Abadi. Language-based enforcement of privacy policies. In *Privacy Enhancing Technologies, 4th International Workshop, PET 2004, Revised Selected Papers*, volume 3424 of *LNCS*, pages 302–313. Springer, 2005.
87. Lars Heinemann, Veronika Bauer, Markus Herrmannsdoerfer, and Benjamin Hummel. Identifier-based context-dependent API method recommendation. In *CSMR*, pages 31–40, 2012.
88. D. Herman. Narrative: Cognitive approaches. In Keith Brown, editor, *Encyclopedia of language and linguistics*. Elsevier Science, 2006.
89. Dirk Heuzeroth, Uwe Aßmann, Mircea Trifu, and Volker Kuttruff. The COMPOST, COMPASS, Inject/J and RECODER tool suite for invasive software composition: Invasive composition with COMPASS aspect-oriented connectors. In *GTTSE*, pages 357–377, 2005.
90. Giles Hogben, Tom Jackson, and Marc Wilikens. A fully compliant research implementation of the P3P standard for privacy protection: Experiences and recommendations. In *Proceedings of ESORICS 2002*, volume 2502 of *LNCS*, pages 104–125. Springer, 2002.
91. Reid Holmes and Robert J. Walker. Informing Eclipse API production and consumption. In *OOPSLA*, pages 70–74, 2007.
92. Daqing Hou and David M. Pletcher. An evaluation of the strategies of sorting, filtering, and grouping API methods for code completion. In *ICSM*, pages 233–242, 2011.
93. J. Howison, M. Conklin, and K. Crowston. FLOSSmole: A collaborative repository for FLOSS research data and analyses. *IJITWE*, pages 17–26, 2006.
94. S. Hunston. Corpus linguistics. In Keith Brown, editor, *Encyclopedia of language and linguistics*. Elsevier Science, 2006.
95. F. Hunt and P. Johnson. On the Pareto distribution of sourceforge projects. In *Proceedings Open Source Software Development Workshop*, pages 122–129, 2002.
96. John M. Daughtry III, Umer Farooq, Jeffrey Stylos, and Brad A. Myers. API usability: CHI'2009 special interest group meeting. In *Proceedings of the 27th International Conference on Human Factors in Computing Systems, CHI 2009, Extended Abstracts Volume*, pages 2771–2774. ACM, 2009.
97. Martin Ivarsson and Tony Gorschek. A method for evaluating rigor and industrial relevance of technology evaluations. *Empir. Softw. Eng.*, 16(3):365–395, 2011.
98. Juanjuan Jiang, Johannes Koskinen, Anna Ruokonen, and Tarja Systä. Constructing usage scenarios for API redocumentation. In *ICPC*, pages 259–264, 2007.

99. Wei Jin and Alessandro Orso. BugRedux: Reproducing field failures for in-house debugging. In *ICSE*, pages 474–484, 2012.
100. Guy L. Steele Jr. Growing a Language. *Higher-Order and Symbolic Computation*, 12(3):221–236, 1999.
101. Uwe Jugel. Generating Smart Wrapper Libraries for Arbitrary APIs. In *Software Language Engineering, Second International Conference, SLE 2009, Revised Selected Papers*, volume 5969 of *LNCS*, pages 354–373. Springer, 2010.
102. Huzefa Kagdi, Michael L. Collard, and Jonathan I. Maletic. An approach to mining call-usage patterns with syntactic context. In *ASE '07: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 457–460. ACM, 2007.
103. Cory Kapser and Michael W. Godfrey. Toward a Taxonomy of Clones in Source Code: A Case Study. In *Proceedings of Evolution of Large Scale Industrial Software Architectures, ELISA workshop 2003*, pages 67–78, 2003.
104. J. Karat, C.-M. Karat, E. Bertino, N. Li, Q. Ni, C. Brodie, J. Lobo, S. B. Calo, L. F. Cranor, P. Kumaraguru, and R. W. Reeder. Policy framework for security and privacy management. *IBM J. Res. Dev.*, 53:242–255, March 2009.
105. Günter Karjoth, Matthias Schunter, and Michael Waidner. Platform for enterprise privacy practices: privacy-enabled management of customer data. In *PET'02: Proceedings of the 2nd international conference on Privacy enhancing technologies*, pages 69–84. Springer, 2003.
106. Owen Kaser and Daniel Lemire. Tag-Cloud Drawing: Algorithms for Cloud Visualization. *CoRR*, abs/cs/0703109, 2007.
107. Patrick Gage Kelley, Joanna Bresee, Lorrie Faith Cranor, and Robert W. Reeder. A "nutrition label" for privacy. In *Proceedings of the 5th Symposium on Usable Privacy and Security*, SOUPS '09, pages 4:1–4:12. ACM, 2009.
108. Miryung Kim, Vibha Sazawal, David Notkin, and Gail Murphy. An empirical study of code clone genealogies. *SIGSOFT Softw. Eng. Notes*, 30(5):187–196, 2005.
109. Barbara Kitchenham. Procedures for undertaking systematic reviews. Technical Report TR/SE-0401, Keele University and National ICT Australia Ltd., 2004.
110. Barbara Kitchenham, Hiyam Al-Khilidar, Muhammed Ali Babar, Mike Berry, Karl Cox, Jacky Keung, Felicia Kurniawati, Mark Staples, He Zhang, and Liming Zhu. Evaluating guidelines for reporting empirical software engineering studies. *Empirical Software Engineering*, 13(1):97–121, 2008.
111. Barbara Kitchenham, O. Pearl Brereton, David Budgen, Mark Turner, John Bailey, and Stephen Linkman. Systematic literature reviews in software engineering - A systematic literature review. *Inf. Softw. Technol.*, 51(1):7–15, 2009.
112. Barbara A. Kitchenham, Tore Dybå, and Magne Jørgensen. Evidence-based software engineering. In *ICSE*, pages 273–281, 2004.
113. Anneke Kleppe. *Software Language Engineering: Creating Domain-Specific Languages Using Metamodels*. Addison-Wesley Professional, 2008.
114. A. S. Klusener, Ralf Lämmel, and Chris Verhoef. Architectural modifications to deployed software. *Sci. of Comput. Program.*, 54(2–3):143–211, 2005.
115. Donald E. Knuth. An empirical study of FORTRAN programs. *Software, Practice & Experience*, 1(2):105–133, 1971.
116. Rainer Koschke. Identifying and removing software clones. In *Software Evolution*, pages 15–36. Springer, 2008.
117. Klaus Krippendorff. *Content Analysis: an Introduction to Its Methodology 2nd edition*. Sage, Thousand Oaks, CA, 2004.

118. G. Kumfert and T. Epperly. Software in the DOE: The hidden overhead of "The Build". Technical report, Lawrence Livermore National Laboratory, 2002.
119. Ohbyung Kwon. A pervasive P3P-based negotiation mechanism for privacy-aware pervasive e-commerce. *Decis. Support Syst.*, 50:213–221, December 2010.
120. R. Lämmel, S. Kitsis, and D. Remy. Analysis of XML schema usage. In *Conference Proceedings XML 2005*, November 2005.
121. Michele Lanza and Stéphane Ducasse. Polymetric views - A lightweight visual approach to reverse engineering. *Trans. Software Eng.*, 29(9):782–795, 2003.
122. Zhenmin Li and Yuanyuan Zhou. PR-Miner: automatically extracting implicit programming rules and detecting violations in large software code. In *ESEC/FSE-13: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 306–315, New York, NY, USA, 2005. ACM.
123. Panos Linos, Whitney Lucas, Sig Myers, and Ezekiel Maier. A metrics tool for multi-language software. In *Proceedings of the 11th IASTED International Conference on Software Engineering and Applications*, pages 324–329. ACTA Press, 2007.
124. Erik Linstead, Sushil Bajracharya, Trung Ngo, Paul Rigor, Cristina Lopes, and Pierre Baldi. Sourcerer: Mining and searching Internet-scale software repositories. *DMKD*, pages 300–336, 2009. `http://sourcerer.ics.uci.edu/`.
125. Simone Livieri, Yoshiki Higo, Makoto Matushita, and Katsuro Inoue. Very-large scale code clone analysis and visualization of open source programs using distributed CCFinder: D-CCFinder. In *ICSE '07: Proceedings of the 29th international conference on Software Engineering*, pages 106–115, 2007.
126. Luis Lopez-Fernandez, Gregorio Robles, and Jesus M. Gonzalez-barahona. Applying social network analysis to the information in CVS repositories. In *Proceedings of the Mining Software Repositories Workshop*, 2004.
127. Rafael Lotufo, Steven She, Thorsten Berger, Krzysztof Czarnecki, and Andrzej Wasowski. Evolution of the Linux kernel variability model. In *SPLC*, pages 136–150, 2010.
128. Homan Ma, Robert Amor, and Ewan D. Tempero. Usage patterns of the Java standard API. In *APSEC*, pages 342–352, 2006.
129. Paul Malone, Mark McLaughlin, Ronald Leenes, Pierfranco Ferronato, Nick Lockett, Pedro Bueso Guillen, Thomas Heistracher, and Giovanni Russello. ENDORSE: a legal technical framework for privacy preserving data management. In *Proceedings of the 2010 Workshop on Governance of Technology, Information and Policies*, GTIP '10, pages 27–34. ACM, 2010.
130. David Mandelin, Lin Xu, Rastislav Bodík, and Doug Kimelman. Jungloid mining: helping to navigate the API jungle. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 48–61. ACM, 2005.
131. Radu Marinescu and Daniel Ratiu. Quantifying the quality of object-oriented design: The factor-strategy model. In *Proceedings of the 11th Working Conference on Reverse Engineering, WCRE '04*, pages 192–201. IEEE, 2004.
132. Thomas J. McCabe. A complexity measure. *IEEE Trans. Software Eng.*, 2(4):308–320, 1976.
133. Shane McIntosh, Bram Adams, and Ahmed E. Hassan. The evolution of Java build systems. *Empirical Software Engineering*, 17(4-5):578–608, 2012.
134. Shane McIntosh, Bram Adams, Thanh H.D. Nguyen, Yasutaka Kamei, and Ahmed E. Hassan. An empirical study of build maintenance effort. In *ICSE*, pages 141–150, 2011.

135. Kim Mens and Andy Kellens. IntensiVE, a toolsuite for documenting and checking structural source-code regularities. In *CSMR*, pages 239–248, 2006.
136. Marjan Mernik, Jan Heering, and Anthony M. Sloane. When and how to develop domain-specific languages. *ACM Comput. Surv.*, 37(4):316–344, 2005.
137. Amir Michail. Data mining library reuse patterns using generalized association rules. In *ICSE '00: Proceedings of the 22nd international conference on Software engineering*, pages 167–176. ACM, 2000.
138. Yana Momchilova Mileva, Valentin Dallmeier, Martin Burger, and Andreas Zeller. Mining trends of library usage. In *ERCIM Workshops*, pages 57–62, 2009.
139. Yana Momchilova Mileva, Valentin Dallmeier, and Andreas Zeller. Mining API popularity. In *TAIC PART*, pages 173–180, 2010.
140. Linda D. Misek-Falkoff. The new field of "software linguistics": An early-bird view. In ACM, editor, *ACM SIGMETRICS workshop on Software Metrics*, 1982.
141. Iman Hemati Moghadam and Mel Ó Cinnéide. Automated refactoring using design differencing. In *CSMR*, pages 43–52, 2012.
142. Marco Casassa Mont and Robert Thyne. Privacy policy enforcement in enterprises with identity management solutions. In *Proceedings of the 2006 International Conference on Privacy, Security and Trust: Bridge the Gap Between PST Technologies and Business Services*, PST '06, pages 25:1–25:12. ACM, 2006.
143. Mozilla's privacy icons project. `https://wiki.mozilla.org/Drumbeat/Challenges/Privacy_Icons`. Visited July 2011.
144. MSR mining challenge 2008, 2011. `http://msr.uwaterloo.ca/msr2008/challenge/`, `http://2011.msrconf.org/msr-challenge.html`.
145. Sarah Nadi and Richard C. Holt. Make it or break it: Mining anomalies from Linux kbuild. In *WCRE*, pages 315–324, 2011.
146. Seyed Mehdi Nasehi and Frank Maurer. Unit tests as API usage examples. In *ICSM*, pages 1–10, 2010.
147. G. Nelson. Description and prescription. In Keith Brown, editor, *Encyclopedia of language and linguistics*. Elsevier Science, 2006.
148. Anh Tuan Nguyen, Tung Thanh Nguyen, Hoan Anh Nguyen, Ahmed Tamrawi, Hung Viet Nguyen, Jafar M. Al-Kofahi, and Tien N. Nguyen. Graph-based pattern-oriented, context-sensitive source code completion. In *ICSE*, pages 69–79, 2012.
149. Chris Parnin, Carsten Görg, and Ogechi Nnadi. A catalogue of lightweight visualizations to support code smell inspection. In *SOFTVIS*, pages 77–86, 2008.
150. Heidar Pirzadeh, Abdelwahab Hamou-Lhadj, and Mohak Shah. Exploiting text mining techniques in the analysis of execution traces. In *ICSM*, pages 223–232, 2011.
151. The PROMISE repository of empirical software engineering data. Since 2005. `http://promisedata.googlecode.com`.
152. Erhard Rahm. Comparing the scientific impact of conference and journal publications in computer science. *Inf. Serv. Use*, 28(2):127–128, April 2008.
153. Ian Reay, Patricia Beatty, Scott Dick, and James Miller. A survey and analysis of the P3P protocol's agents, adoption, maintenance, and future. *IEEE Transactions on Dependable and Secure Computing*, 4(2):151–164, April-June 2007.
154. Ian Reay, Scott Dick, and James Miller. A large-scale empirical study of P3P privacy policies: Stated actions vs. legal obligations. *ACM Transactions on the Web (TWEB)*, 3(2), 2009.
155. Christoph Ringelstein and Steffen Staab. PAPEL: Provenance-aware policy definition and execution. *IEEE Internet Computing*, 15:49–58, 2011.
156. Martin P. Robillard and Robert DeLine. A field study of API learning obstacles. *Empir. Softw. Eng.*, 16(6):703–732, 2011.

157. S. K. Robinson and I. S. Torsun. An empirical analysis of FORTRAN programs. *The Computer Journal*, 19(1):56–62, 1976.
158. Atanas Rountev, Scott Kagan, and Michael Gibas. Static and dynamic analysis of call chains in java. In *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2004*, pages 1–11. ACM, 2004.
159. Chanchal Kumar Roy, James R. Cordy, and Rainer Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of Computer Programming*, 74(7):470–495, 2009.
160. Harry J. Saal and Zvi Weiss. An empirical study of APL programs. *Computer Languages*, 2:47–59, 1977.
161. Farzad Salim, Nicholas Paul Sheppard, and Rei Safavi-Naini. Enforcing P3P policies using a digital rights management system. In *PET'07: Proceedings of the 7th international conference on Privacy enhancing technologies*, pages 200–217. Springer, 2007.
162. Thorsten Schäfer, Jan Jonas, and Mira Mezini. Mining framework usage changes from instantiation code. In *Proceedings of the 30th international conference on Software engineering, ICSE '08*, pages 471–480. ACM, 2008.
163. Axel Schmolitzky. Teaching inheritance concepts with Java. In *Proceedings of the 4th international symposium on Principles and practice of programming in Java, PPPJ '06*, pages 203–207. ACM, 2006.
164. Ari Schwartz. Looking back at P3P: Lessons for the future. `http://www.cdt.org/files/pdfs/P3P_Retro_Final_0.pdf`, 2009. Visited July 2011.
165. Francisco Servant and James A. Jones. WhoseFault: Automatic developer-to-fault assignment through fault localization. In *ICSE*, pages 36–46, 2012.
166. Jérôme Siméon and Philip Wadler. The essence of XML. In *POPL*, pages 1–13, 2003.
167. Dag I. K. Sjøberg, Tore Dybå, and Magne Jørgensen. The future of empirical methods in software engineering research. In *FOSE*, pages 358–378, 2007.
168. Dag I. K. Sjøberg, Jo E. Hannay, Ove Hansen, Vigdis By Kampenes, Amela Karahasanović, Nils-Kristian Liborg, and Anette C. Rekdal. A survey of controlled experiments in software engineering. *IEEE Trans. Softw. Eng.*, 31(9):733–753, 2005.
169. Guy L. Steele, Jr., and Richard P. Gabriel. The evolution of Lisp. In *ACM SIGPLAN Notices*, pages 231–270. Press, 1993.
170. Margaret-Anne Storey Steve Easterbrook, Janice Singer and Daniela Damian. Selecting empirical methods for software engineering research. In Forrest Shull, Janice Singer, and Dag I.K. Sjøberg, editors, *Guide to Advanced Empirical Software Engineering*, pages 285–311. Springer, 2008.
171. Jeffrey Stylos. *Making APIs More Usable with Improved API Designs, Documentation and Tools*. PhD thesis, Carnegie Mellon University, 2009.
172. Jeffrey Stylos, Andrew Faulring, Zizhuang Yang, and Brad A. Myers. Improving API documentation using API usage information. In *VL/HCC*, pages 119–126, 2009.
173. Chengnian Sun, Siau-Cheng Khoo, and Shao Jie Zhang. Graph-based detection of library API imitations. In *ICSM*, pages 183–192, 2011.
174. Mark D. Syer, Bram Adams, Ying Zou, and Ahmed E. Hassan. Exploring the development of micro-apps: A case study on the BlackBerry and Android platforms. In *SCAM*, pages 55–64, 2011.
175. Ewan Tempero, Craig Anslow, Jens Dietrich, Ted Han, Jing Li, Markus Lumpe, Hayden Melton, and James Noble. Qualitas corpus: A curated collection of Java code for empirical studies. In *APSEC*, pages 336–345, 2010. `http://qualitascorpus.com/`.
176. Ewan Tempero, Steve Counsell, and James Noble. An empirical study of overriding in open source Java. In *Proceedings of the Thirty-Third Australasian Conferenc on*

*Computer Science - Volume 102, ACSC '10*, pages 3–12. Australian Computer Society, 2010.

177. Ewan D. Tempero, James Noble, and Hayden Melton. How do java programs use inheritance? an empirical study of inheritance in java software. In *ECOOP 2008 - Object-Oriented Programming, 22nd European Conference, Proceedings*, volume 5142 of *LNCS*, pages 667–691. Springer, 2008.
178. The Center for Information Policy Leadership. Multi-layered notices explained. `http://aimp.apec.org/Documents/2005/ECSG/DPM1/05_ecsg_dpm1_003.pdf`, 2004. Visited July 2011.
179. The Center for Information Policy Leadership. Ten steps to develop a multilayered privacy notice. `http://www.informationpolicycentre.com/files/Uploads/Documents/Centre/Ten_Steps_whitepaper.pdf`, 2005. Visited July 2011.
180. Suresh Thummalapenta and Tao Xie. SpotWeb: Detecting framework hotspots and coldspots via mining open source code on the Web. In *ASE*, pages 327–336, 2008.
181. Nikolai Tillmann and Jonathan de Halleux. Pex-white box test generation for .NET. In *Tests and Proofs, Second International Conference, TAP 2008, Proceedings*, volume 4966 of *LNCS*, pages 134–153. Springer, 2008.
182. Adrian Trifu and Radu Marinescu. Diagnosing design problems in object oriented systems. In *Proceedings of the 12th Working Conference on Reverse Engineering*, pages 155–164. IEEE, 2005.
183. TRUSTe. `http://www.truste.com`. Visited July 2011.
184. Tarmo Uustalu and Varmo Vene. The Essence of Dataflow Programming. In *Programming Languages and Systems, APLAS 2005, Proceedings*, volume 3780 of *LNCS*, pages 2–18. Springer, 2005.
185. Stephane Vaucher, Foutse Khomh, Naouel Moha, and Yann-Gael Gueheneuc. Tracking design smells: Lessons from a study of god classes. In *Proceedings of the 2009 16th Working Conference on Reverse Engineering, WCRE '09*, pages 145–154. IEEE, 2009.
186. Todd L. Veldhuizen. Software libraries and their reuse: Entropy, Kolmogorov complexity, and Zipf's law. *CoRR*, abs/cs/0508023, 2005.
187. Joost Visser. Structure Metrics for XML Schema. In *Proceedings of XATA 2006*, 2006.
188. W3C. A P3P preference exchange language 1.0 (APPEL1.0), W3C working draft, 2002. `http://www.w3.org/TR/P3P-preferences/`.
189. W3C. The platform for privacy preferences 1.1 (P3P1.1) specification, 2006. `http://www.w3.org/TR/P3P11/`.
190. Philip Wadler. The Essence of Functional Programming. In *POPL*, pages 1–14, 1992.
191. Westley Weimer and George C. Necula. Mining temporal specifications for error detection. In *In TACAS*, pages 461–476, 2005.
192. Dawid Weiss. Quantitative analysis of open source projects on SourceForge. In Marco Scotto and Giancarlo Succi, editors, *Proceedings of the First International Conference on Open Source Systems, Genova*, pages 140–147, 2005.
193. Richard Wettel, Michele Lanza, and Romain Robbes. Software systems as cities: A controlled experiment. In *ICSE*, pages 551–560, 2011.
194. David S. Wile. Lessons learned from real DSL experiments. *Sci. Comput. Program.*, 51(3):265–290, 2004.
195. Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, Bjöorn Regnell, and Anders Wesslén. *Experimentation in software engineering: An introduction*. Kluwer Academic Publishers, 2000.

196. Tao Xie and Jian Pei. MAPO: mining API usages from open source repositories. In *MSR '06: Proceedings of the 2006 international workshop on Mining software repositories*, pages 54–57, New York, NY, USA, 2006. ACM.
197. Ting Yu, Ninghui Li, and Annie I. Antón. A formal semantics for P3P. In *Proceedings of SWS 2004*, pages 1–8. ACM, 2004.
198. Carmen Zannier, Grigori Melnik, and Frank Maurer. On the success of empirical studies in the international conference on software engineering. In *ICSE*, pages 341–350, 2006.
199. Hao Zhong, Tao Xie, Lu Zhang, Jian Pei, and Hong Mei. MAPO: Mining and recommending API usage patterns. In *ECOOP*, pages 318–343, 2009.

# Curriculum Vitae

| | |
|---|---|
| Contact | Ekaterina Pek,<br>Universität Koblenz-Landau,<br>Universitätsstr. 1,<br>56070 Koblenz,<br>Germany<br>`pek@uni-koblenz.de` ● `www.uni-koblenz.de/~pek` |

## Education

### School

| | |
|---|---|
| 1995–1998 | *Astrakhan Technical Lyceum*, Astrakhan, Russia. Honors graduation (*summa cum laude*). |
| 1987–1995 | *Lyceum Nr. 1*, Astrakhan, Russia. |

### University

| | |
|---|---|
| 1998–2003 | *Rostov State University*, Rostov-on-Don, Russia. Diploma in Applied Mathematics. Honors graduation (*summa cum laude*). |

## Employment

### Academic Experience

| | |
|---|---|
| 01/2009– | *University Koblenz-Landau*, Koblenz, Germany.<br>PhD student and full-time researcher. |

### Industry Experience

| | |
|---|---|
| 05/2012–08/2012 | *Google, Inc.*, Munich, Germany.<br>Intern, CodeSearch project. |

| | |
|---|---|
| 10/2007–10/2008 | *SoftTech, Ltd.*, Taganrog, Russia. J2EE Developer; financial mobile platform. |
| 06/2006–10/2007 | *Inversion, JSC*, Rostov, Russia. J2EE Developer; online banking systems. |
| 08/2004–06/2006 | *Rostov-Clearing, JSC*, Rostov, Russia. Java Software Developer; desktop banking systems. |
| 01/2004–08/2004 | *Pralog, Ltd.*, Moscow, Russia. Microsoft SQL Database Administrator; retailer systems. |

## Conference Presentations

| | |
|---|---|
| 01/07/2010 | *Vivisection of a non-executable, domain-specific language*, 18th IEEE International Conference on Program Comprehension (ICPC 2010), Braga, Portugal. |
| 11/10/2010 | *Towards Privacy Policy-Aware Web-Based Systems*, 1st Doctoral Symposium of the International Conference on Software Language Engineering (SLE-DS 2010), Eindhoven, The Netherlands. |
| 13/10/2010 | *Empirical language analysis in software linguistics*, 3rd International Conference on Software Language Engineering (SLE 2010), Eindhoven, The Netherlands. |
| 23/03/2011 | *Large-scale, AST-based API-usage analysis of open-source Java projects*, 26th ACM Symposium on Applied Computing (SAC 2011), Taichung, Taiwan. |
| 18/10/2011 | *A Framework Profile of .NET*, 18th Working Conference on Reverse Engineering (WCRE 2011), Limerick, Ireland. |

## Professional Activities

| | |
|---|---|
| 2003 | Microsoft Certified Professional: Microsoft SQL Server 2000 Database Administrator. |
| 03/2010 | IEEE Student Member. |
| 10/2010 | ACM Student Member. |
| 11/2010 | SIGAPP Member. |
| 11/2010 | Received the Student Travel Award from ACM SIGAPP for travelling to Taiwan (SAC 2011). |
| 01/2011 | GI (Gesellschaft fuer Informatik) Member. |

| | |
|---|---|
| 04/2011 | Local organizer for 2011 edition of the Seminar Series on Advanced Techniques & Tools for Software Evolution. |
| 2011 | Received the Google Anita Borg Memorial Scholarship. |
| 2013 | A Programm Committee member of ICSM 2013. |
| 2009-2013 | Reviewer for the conferences: CADE, CASTA, CSMR, IECCS, ICPC, MODELS, SCAM. |

# A

# Appendix

## A.1 Appendix for Chapter 3

### A.1.1 Additional Information on P3P Corpus

#### Additional information on ODP

The catalog of Open Directory Project (ODP) is subdivided into categories and subcategories through multiple levels. There are these top-level categories: *Arts, Business, Computers, Games, Health, Home, Kids and Teens, News, Recreation, Reference, Regional, Science, Shopping, Society, Sports, World*. There are two special top-level categories: *World* and *Regional*. The following discussion of these two special categories should be helpful when trying to understand ODP in general and its overall coverage of the WWW.

| | **Domain** | **Description** | **#URLs (cumulative %)** | **#domains (cumulative %)** |
|---|---|---|---|---|
| 1 | .com | commercial | 3590 (58.13 %) | 3402 (60.30 %) |
| 2 | .uk | United Kingdom | 716 (69.72 %) | 686 (72.46 %) |
| 3 | .gov | U.S. governmental | 363 (75.60 %) | 315 (78.04 %) |
| 4 | .it | Italy | 247 (79.60 %) | 84 (79.53 %) |
| 5 | .org | organization | 246 (83.58 %) | 214 (83.32 %) |
| 6 | .de | Germany | 217 (87.10 %) | 211 (87.06 %) |
| 7 | .net | network | 188 (90.14 %) | 175 (90.16 %) |
| 8 | .kr | Republic of Korea | 76 (91.37 %) | 76 (91.51 %) |
| 9 | .au | Australia | 59 (92.33 %) | 53 (92.45 %) |
| 10 | .ca | Canada | 57 (93.25 %) | 54 (93.41 %) |
| 11 | .dk | Denmark | 43 (93.94 %) | 41 (94.13 %) |
| 12 | .edu | educational | 40 (94.59 %) | 25 (94.58 %) |
| 13 | .mx | Mexico | 36 (95.17 %) | 34 (95.18 %) |
| 14 | .es | Spain | 34 (95.73 %) | 18 (95.50 %) |
| 15 | .fr | France | 32 (96.24 %) | 31 (96.05 %) |
| 16 | .ie | Republic of Ireland | 24 (96.63 %) | 22 (96.44 %) |

| | | | |
|---|---|---|---|
| 17 .biz | business | 21 (96.97 %) | 21 (96.81 %) |
| 18 .info | information | 18 (97.26 %) | 16 (97.09 %) |
| 19 .us | United States of America | 18 (97.56 %) | 18 (97.41 %) |
| 20 .at | Austria | 16 (97.81 %) | 15 (97.68 %) |
| 21 .jp | Japan | 16 (98.07 %) | 16 (97.96 %) |
| 22 .cz | Czech Republic | 14 (98.30 %) | 14 (98.21 %) |
| 23 .nz | New Zealand | 12 (98.49 %) | 12 (98.42 %) |
| 24 .ch | Switzerland | 11 (98.67 %) | 11 (98.62 %) |
| 25 .be | Belgium | 9 (98.82 %) | 9 (98.78 %) |
| 26 .mil | U.S. military | 9 (98.96 %) | 8 (98.92 %) |
| 27 .nl | Netherlands | 9 (99.11 %) | 8 (99.06 %) |
| 28 .il | Israel | 5 (99.19 %) | 5 (99.15 %) |
| 29 .bg | Bulgaria | 4 (99.26 %) | 4 (99.22 %) |
| 30 .is | Iceland | 3 (99.30 %) | 3 (99.27 %) |
| 31 .ru | Russia | 3 (99.35 %) | 2 (99.31 %) |
| 32 .za | South Africa | 3 (99.40 %) | 3 (99.36 %) |
| 33 .cc | Cocos (Keeling) Islands | 2 (99.43 %) | 2 (99.40 %) |
| 34 .cn | People's Republic of China | 2 (99.47 %) | 2 (99.43 %) |
| 35 .eu | European Union | 2 (99.50 %) | 2 (99.47 %) |
| 36 .gr | Greece | 2 (99.53 %) | 2 (99.50 %) |
| 37 .ma | Morocco | 2 (99.56 %) | 1 (99.52 %) |
| 38 .ws | Samoa | 2 (99.60 %) | 2 (99.56 %) |
| 39 .ae | United Arab Emirates | 1 (99.61 %) | 1 (99.57 %) |
| 40 .ao | Angola | 1 (99.63 %) | 1 (99.59 %) |
| 41 .ar | Argentina | 1 (99.64 %) | 1 (99.61 %) |
| 42 .bo | Bolivia | 1 (99.66 %) | 1 (99.63 %) |
| 43 .br | Brazil | 1 (99.68 %) | 1 (99.65 %) |
| 44 .cl | Chile | 1 (99.69 %) | 1 (99.66 %) |
| 45 .ee | Estonia | 1 (99.71 %) | 1 (99.68 %) |
| 46 .fo | Faroe Islands | 1 (99.72 %) | 1 (99.70 %) |
| 47 .hk | Hong Kong | 1 (99.74 %) | 1 (99.72 %) |
| 48 .hr | Croatia | 1 (99.76 %) | 1 (99.73 %) |
| 49 .im | Isle of Man | 1 (99.77 %) | 1 (99.75 %) |
| 50 .in | India | 1 (99.79 %) | 1 (99.77 %) |
| 51 .li | Liechtenstein | 1 (99.81 %) | 1 (99.79 %) |
| 52 .md | Moldova | 1 (99.82 %) | 1 (99.81 %) |
| 53 .my | Malaysia | 1 (99.84 %) | 1 (99.82 %) |
| 54 .no | Norway | 1 (99.85 %) | 1 (99.84 %) |
| 55 .pk | Pakistan | 1 (99.87 %) | 1 (99.86 %) |
| 56 .pl | Poland | 1 (99.89 %) | 1 (99.88 %) |
| 57 .sk | Slovakia | 1 (99.90 %) | 1 (99.89 %) |
| 58 .sm | San Marino | 1 (99.92 %) | 1 (99.91 %) |
| 59 .th | Thailand | 1 (99.94 %) | 1 (99.93 %) |
| 60 .tv | Tuvalu | 1 (99.95 %) | 1 (99.95 %) |
| 61 .tw | Republic of China (Taiwan) | 1 (99.97 %) | 1 (99.96 %) |
| 62 .uy | Uruguay | 1 (99.98 %) | 1 (99.98 %) |
| 63 .zm | Zambia | 1 (100.00 %) | 1 (100.00 %) |

Table A.1: Top-level domains in policies URLs

Consider the *World* category: these are "sites in languages other than English" where languages actually serve as subcategories of *World*.[1] Also: "If a site's content is available in more than one language, the site may be listed in more than one language category. For example, if a site is in English, German, and French, the site may be listed in an English-only category, World/Deutsch, and World/Francais".[2] The term 'English-only category' refers to the top-level categories other than *World* and *Regional*.

Consider the *Regional* category: this "category lists sites specific to a particular geographic area. The Regional category as a whole organizes sites according to their geographic focus and relevance to a particular regional population. To this end, individual Regional categories become mini-web directories in their own right, while remaining functionally part of the larger Open Directory."[3]

There may be multiple occurrences of a URL within the directory. That is, a URL may have more than one associated category.[4] However, ODP's rules seem to stipulate preference of fitting each site into one category, if there is a best match. Special reasons for overlapping apply to the categories *World* and *Regional*. That is, *World* can overlap with the rest of the directory when a site has an English version, and *Regional* can overlap with the rest of the directory when "sites are relevant to a subject category and a specific local geographic area".

## Additional information on corpus diversity

Table A.1 breaks down all policies in terms of the top-level domain of the policy URL (i.e., the URL found in the policy reference file). The larger part of policies resides in the com domain. The domains gov, org and net are popular, too. The seven first top-level domains cover about 90 % of all policies. We show such information here merely as a short indication of the corpus' diversity, as it was obtained from ODP. This information must not be misunderstood as being part of the study, which is not concerned with P3P adoption by domain, country, or other means of breakdown; see §3.6.

Fig. A.1 shows the distribution of policies over ODP's website categories. Nearly half of all policies are (also) regional policies. Fig. A.2 shows the distribution of policies in terms of the numbers of associated categories; 81.20 % of all policies are associated with only one category. Table A.2 shows the distribution of policies over subcategories of *World*—that is, over languages. Please note the category *English*

[1] `http://www.dmoz.org/guidelines/site-specific.html#non-english`
[2] `http://www.dmoz.org/guidelines/site-specific.html#multi-lingual`
[3] `http://www.dmoz.org/guidelines/regional/`
[4] `http://www.dmoz.org/guidelines/site-specific.html#multiple`

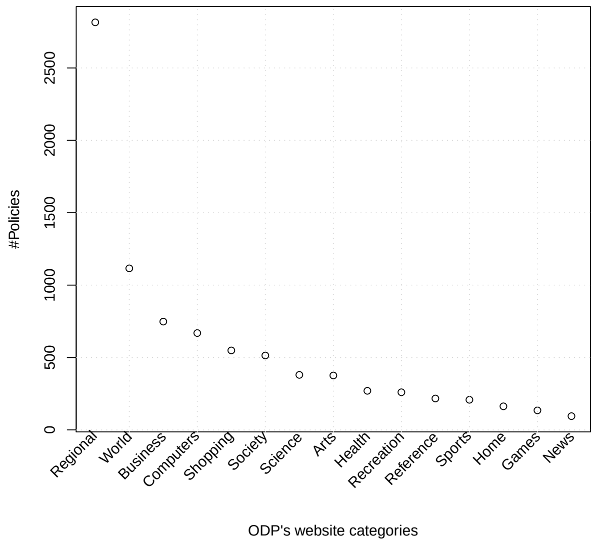


**Figure A.1.** Number of policies per ODP category

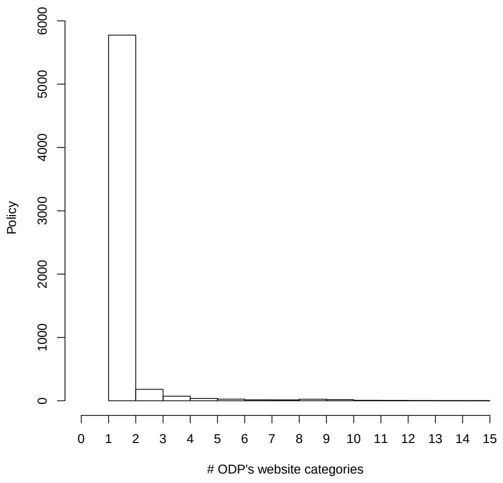


**Figure A.2.** Number of ODP categories per policy

| Subcategory | # Policies | # Websites | Percent |
|---|---|---|---|
| Español | 1065 | 151144 | 0.705 % |
| Deutsch | 606 | 502004 | 0.121 % |
| Dansk | 567 | 46628 | 1.216 % |
| Français | 527 | 244261 | 0.216 % |
| Italiano | 506 | 194942 | 0.260 % |
| Chinese | 156 | 56295 | 0.277 % |
| Japanese | 116 | 106869 | 0.109 % |
| Català | 75 | 36502 | 0.205 % |
| Arabic | 65 | 6228 | 1.044 % |
| Esperanto | 42 | 4167 | 1.008 % |
| Hebrew | 42 | 6379 | 0.658 % |
| Český | 29 | 25307 | 0.115 % |
| Indonesian | 26 | 2987 | 0.870 % |
| Greek | 21 | 2194 | 0.957 % |
| Galego | 15 | 1661 | 0.903 % |
| Hrvatski | 11 | 5099 | 0.216 % |
| Euskara | 10 | 1713 | 0.584 % |
| Hindi | 9 | 493 | 1.826 % |
| Afrikaans | 9 | 489 | 1.840 % |
| Gàidhlig | 9 | 127 | 7.087 % |
| Bulgarian | 6 | 3691 | 0.163 % |
| Brezhoneg | 5 | 229 | 2.183 % |
| Azerbaijani | 4 | 1113 | 0.359 % |
| Melay | 4 | 320 | 1.250 % |
| Føroyskt | 3 | 65 | 4.615 % |
| Íslenska | 2 | 419 | 0.477 % |
| Cymraeg | 2 | 389 | 0.514 % |
| Gujarati | 1 | 51 | 1.961 % |
| Armenian | 1 | 1128 | 0.089 % |
| Eesti | 1 | 1278 | 0.078 % |

**Table A.2.** Policies by subcategories of ODP's World category (raw data)

is not included here because it is not a valid subcategory of *World* in ODP's sense. Table also shows number of websites and percentage of policies per subcategory.

### Additional information on disappeared policies

The corpus was originally downloaded in Dec 2009–Sep 2010. We verified the availability of the corpus' policies in Jan 2012. We found that 1,578 policies (out of 6,182) cannot be obtained anymore. Those disappeared policies come from 1,395 different websites. We also checked if the sites themselves still exist; it turned out that 173 sites disappeared. Such information may be useful in making claims about the potential decline of P3P. Such claims are not central though to the contributions of the present effort.

We analyzed the disappeared policies while only considering those policies whose sites still exist. We looked into clone group information and syntactical size. Fig. A.3 shows the results of the analysis as follows:

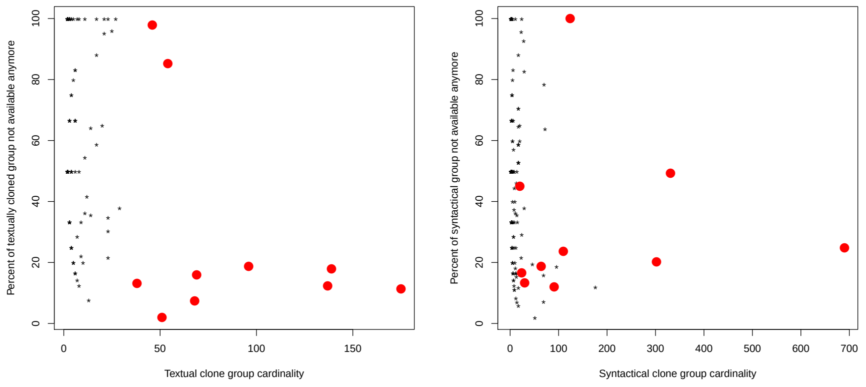


**Figure A.3.** Disappeared policies: (a) Textual cloning; (b) Syntactical cloning

- Subfigure (a) shows how disappeared policies are distributed over textual clone groups where the x axis shows the clone group's cardinality and the y axis shows the percentage of disappeared policies. Large (red) dots show positions of top-ten textual clone groups. We observe that top-ten groups were mostly only slightly affected (losing up to 20 % of their clones). There is one top-ten clone group that was eliminated almost completely. This effect can be associated with a particular hosting service.
- Subfigure (b) applies to syntactical clone groups instead of textual ones.

### A.1.2 Additional Information on P3P Semantics

#### Additional information on '$\leq_{sem}$'

#### '$\leq_{sem}$' for retention levels

The value 'no retention' (i.e., not storing data at all) implies the most privacy; the value 'indefinite retention' (i.e., storing data forever) implies the least privacy. All other values are hard to differentiate in terms of privacy. Hence, we group them between top and bottom.

#### '$\leq_{sem}$' for recipients

The value 'ours' (i.e., essentially the corresponding system itself) implies the most privacy if we assume that adding any further recipient decreases privacy. In fact, P3P anticipates recipients that use the ‘same’ policy, and hence we group ‘same’ and ‘ours’ together at the bottom.[5] The value ‘public’ clearly implies the least privacy. All the other recipients are hard to differentiate in terms of privacy since we simply do not know anything about their privacy policies. Hence, we group them between top and bottom.

## ‘$\leq_{sem}$’ for purposes

The value ‘current’ implies the most privacy since it models use of the system for its primary purpose. We treat the values ‘admin’ and ‘develop’ as equal; these are activities internal to the corresponding system. The value ‘tailoring’ further decreases privacy in that the system is adapted for the user based on data from the current session. For instance, the website’s content or design may be adapted. There are several purposes for ‘analysis’ and ‘decision’ that we all consider to maintain less privacy than ‘tailoring’. A purpose with analysis in the name targets at “research, analysis and reporting” whereas a purpose with decision in the name targets at “a decision that directly affects that individual” [189].[6] We contend that ‘analysis’ implies less exposure than ‘decision’, and the prefix ‘pseudo’ provides implies less exposure than prefix ‘individual’. At the top, we have purposes ‘contact’, ‘historical’, and ‘telemarketing’, as they are typically the least related to the original purpose of the system.

## Additional information on semantically distinct policies

We are left with 1,385 semantically distinct policies of 4,869 semantically valid policies of a total of 6,182 policies in the corpus. This remaining diversity can be further characterized on the grounds of the partial order for the degree of exposure; see §3.3.4. That is, all the semantically distinct policies combined with ‘$\leq_{sem}$’ define a Hasse diagram with the ‘full privacy’ policy as bottom element. Alas, the diagram is too large for inclusion, but we can discuss it in an abstract manner.

- Number of nodes = 1,385 (= number of semantically distinct policies)
- Number of edges = 2,102
- Number of maximal elements = 983
- Longest chain length = 12

As an illustration, Fig. A.4 shows the chains for the policy with the ‘greatest height’ in the Hasse diagram. (This is the policy with the longest chain length = 12; there happens to be only one such policy.) The nodes either refer to clone groups (specified by number and cardinality) or to specific (say, unique) policies. The edges are labeled with the relation between smaller and greater element in compliance with ‘$\leq_{sem}$’. We propose that an inspection of the Hasse diagram, as exercised here, may

[5] One might argue that ‘same’ decreases privacy, when compared to just ‘ours’ since data is shared with another entity. This would mean that we essentially trust the primary entity more than the other entity.

[6] `http://www.w3.org/TR/P3P11/#PURPOSE`

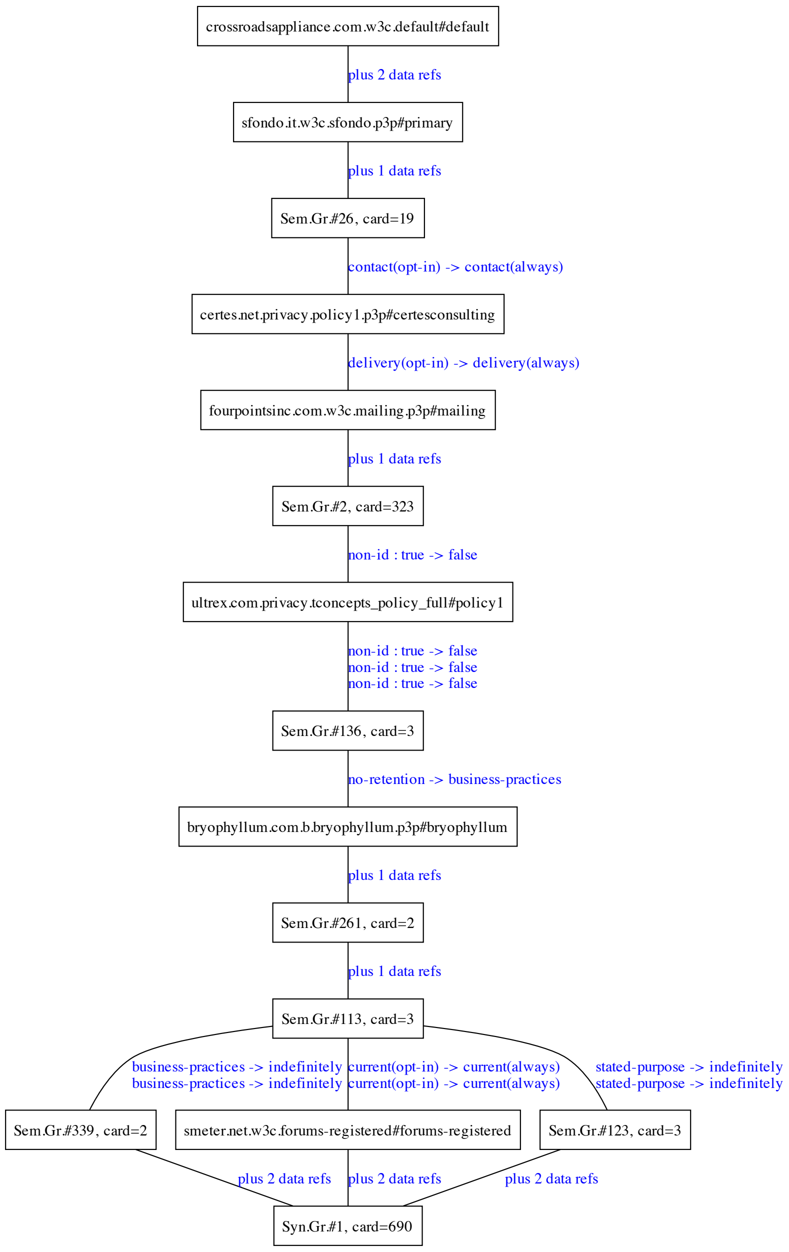

crossroadsappliance.com.w3c.default#default
plus 2 data refs
sfondo.it.w3c.sfondo.p3p#primary
plus 1 data refs
Sem.Gr.#26, card=19
contact(opt-in) -> contact(always)
certes.net.privacy.policy1.p3p#certesconsulting
delivery(opt-in) -> delivery(always)
fourpointsinc.com.w3c.mailing.p3p#mailing
plus 1 data refs
Sem.Gr.#2, card=323
non-id : true -> false
ultrex.com.privacy.tconcepts_policy_full#policy1
non-id : true -> false
non-id : true -> false
non-id : true -> false
Sem.Gr.#136, card=3
no-retention -> business-practices
bryophyllum.com.b.bryophyllum.p3p#bryophyllum
plus 1 data refs
Sem.Gr.#261, card=2
plus 1 data refs
Sem.Gr.#113, card=3
business-practices -> indefinitely
business-practices -> indefinitely
current(opt-in) -> current(always)
current(opt-in) -> current(always)
stated-purpose -> indefinitely
stated-purpose -> indefinitely
Sem.Gr.#339, card=2
smeter.net.w3c.forums-registered#forums-registered
Sem.Gr.#123, card=3
plus 2 data refs
plus 2 data refs
plus 2 data refs
Syn.Gr.#1, card=690

**Figure A.4.** The longest partially ordered chain(s) of semantically distinct policies

be helpful in deciding replacements of uncommon policies by common ones. This may be useful for privacy approaches that stipulate a smaller number of 'common privacy scenarios'.

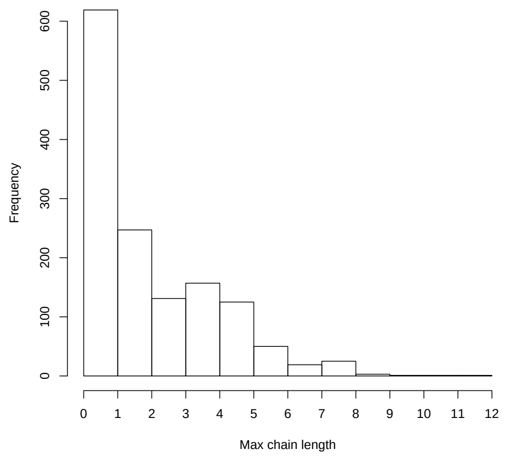


**Figure A.5.** Distribution of longest paths for semantically distinct policies

Fig. A.5 shows the distribution of longest paths from the bottom element no non-bottom elements. About half of all semantically distinct policies are 'alone'; they are not approximated by any policies but the bottom element; they do not approximate any policies. One may expect a top element in the Hasse diagram—something like 'no privacy' (as opposed to 'full privacy'). This top element is not exercised and this may be impractical because of P3P's variable-category data elements. The number of edges is an indicator that many policies serve as joins because the lowest possible number of edges in a partially ordered set with a bottom element equals the number of elements − 1. This means that if uncommon policies are close (in terms of '$\leq_{sem}$') to a common policy, then one could decide to adopt the common policy in favor of the uncommon one. Future work on the diversity of P3P policies is needed.

## Additional information on P3P extensions

To understand the diversity of policies that use extensions other than 'group-info', we examined the domains of the underlying websites. All these sites use distinct domains. Table A.3 lists frequency of top-level domains. Hence, most cases of non-trivial usage of the extension mechanism come from websites of the Republic of Korea and from commercial websites.

| **Entity** | **Top-level domain** | **#** |
|---|---|---|
| Republic of Korea | .kr | 76 |
| commercial | .com | 66 |
| network | .net | 6 |
| U.S. governmental | .gov | 5 |
| organization | .org | 4 |
| People's Republic of China | .cn | 1 |
| Germany | .de | 1 |
| Australia | .au | 1 |
| Italy | .it | 1 |
| Pakistan | .pk | 1 |

**Table A.3.** Top-level domains of policies using extensions aside 'group-info'

We further examined the domains. We observed a few cases of shared subdomains: (cii.samsung.co.kr, shi.samsung.co.kr), (cops.usdoj.gov, usdoj.gov), (hoam.samsungfoundation.org, kids.samsungfoundation.org), (ec21.com, ec21.net). Also, we observed the case of a company, *Samsung*, with several websites using domains without shared subdomains: cii.samsung.co.kr, hoam.samsungfoundation.org, kids.samsungfoundation.org, samsungengineering.co.kr, samsungengineering.com, samsungtechwin.com, sem.samsung.com, shi.samsung.co.kr. Further screening of all domains with top-level domains `.kr` and `.com` confirmed that indeed diverse entities appear in those lists. A noticeable number of commercial sites concerned Korean companies. Hence, we found that most non-trivial extension usage concentrates on Korea while exercising various companies and non-commercial entities. At the time of writing, we do not know the reasons for such non-proportional use of extensions in Korea. One possible reason could be a particular, national legislation.

## A.2 Appendix for Chapter 4

### A.2.1 Additional Information on Java APIs

#### Full list of known Java APIs used in Section 4.2

| | API | Domain | Core | # Projects | # Calls | # Methods called |
|---|---|---|---|---|---|---|
| 1 | **Java Collections** | Collections | yes | 1374 | 392639 | 406 |
| 2 | **AWT** | GUI | yes | 754 | 360903 | 1607 |
| 3 | **Swing** | GUI | yes | 716 | 581363 | 3369 |
| 4 | **Reflection** | Other | yes | 560 | 15611 | 154 |
| 5 | **Core XML** | XML | yes | 413 | 90415 | 537 |
| 6 | **DOM** | XML | yes | 324 | 52593 | 180 |
| 7 | **SAX** | XML | no | 310 | 13725 | 156 |
| 8 | **log4j** | Logging | no | 254 | 43533 | 187 |
| 9 | **JUnit** | Testing | no | 233 | 71481 | 1011 |
| 10 | **Comm.Logging** | Logging | no | 151 | 21996 | 88 |
| 11 | **JNDI** | Networking | yes | 101 | 2900 | 130 |
| 12 | **Comm.Lang** | Other | no | 93 | 4620 | 405 |
| 13 | **JDOM** | XML | no | 86 | 16770 | 423 |
| 14 | **RMI** | Networking | yes | 64 | 1183 | 46 |
| 15 | **Hibernate** | Database | no | 63 | 15192 | 2123 |
| 16 | **Comm.Beanutils** | Other | no | 51 | 407 | 67 |
| 17 | **Xerces** | XML | no | 42 | 3337 | 213 |
| 18 | **Comm.Collections** | Collections | no | 37 | 4085 | 1271 |
| 19 | **dom4j** | XML | no | 37 | 21874 | 157 |
| 20 | **Lucene** | Search | no | 36 | 12302 | 1684 |
| 21 | **Comm.IO** | IO | no | 34 | 450 | 72 |
| 22 | **Comm.CLI** | Other | no | 32 | 2463 | 134 |
| 23 | **Comm.FileUpload** | Networking | no | 31 | 626 | 49 |
| 24 | **Axis** | Webservices | no | 30 | 15746 | 210 |
| 25 | **SWT** | GUI | no | 30 | 56846 | 4361 |
| 26 | **JMF** | Media | no | 28 | 9030 | 488 |
| 27 | **Comm.Codec** | Other | no | 27 | 1064 | 108 |
| 28 | **Struts** | Web Apps | no | 26 | 11938 | 227 |
| 29 | **Comm.Digester** | XML | no | 20 | 1127 | 68 |
| 30 | **Jena** | Semantic Web | no | 20 | 7304 | 787 |
| 31 | **BC Crypto** | Security | no | 16 | 5147 | 569 |
| 32 | **Comm.DBCP** | Database | no | 15 | 119 | 48 |
| 33 | **jMock2** | Testing | no | 15 | 3888 | 46 |
| 34 | **TestNG** | Testing | no | 14 | 3974 | 24 |

Continued on next page

**Table A.4 – continued from previous page**

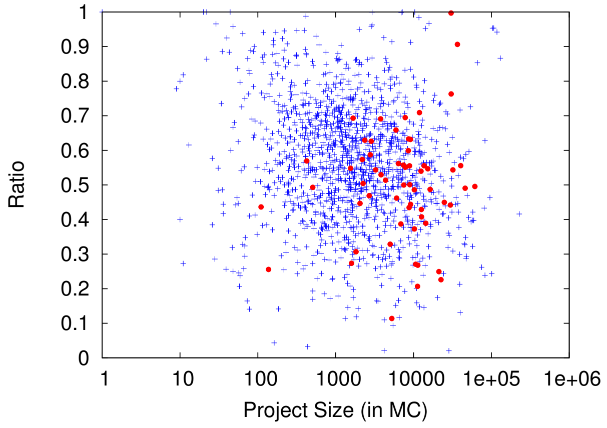
| | API | Domain | Core | # Projects | # Calls | # Methods called |
|---|---|---|---|---|---|---|
| 35 | **Comm.Pool** | Other | no | 14 | 171 | 43 |
| 36 | **GWT** | Web Apps | no | 13 | 12987 | 639 |
| 37 | **Java 3D** | GUI | no | 12 | 490 | 80 |
| 38 | **JFace** | GUI | no | 10 | 3588 | 346 |
| 39 | **Batik** | GUI | no | 10 | 336 | 70 |
| 40 | **Comm.Net** | Networking | no | 10 | 2016 | 646 |
| 41 | **LWJGL** | GUI | no | 10 | 2988 | 325 |
| 42 | **Berkeley DB** | Database | no | 9 | 769 | 128 |
| 43 | **Comm.Configuration** | Other | no | 9 | 485 | 55 |
| 44 | **JAI** | GUI | no | 9 | 111 | 50 |
| 45 | **XMLBeans** | XML | no | 9 | 9891 | 154 |
| 46 | **jogl** | GUI | no | 9 | 261 | 45 |
| 47 | **MySQL Connector/J** | Database | no | 8 | 17738 | 1288 |
| 48 | **JavaHelp** | GUI | no | 8 | 29 | 10 |
| 49 | **XMLPull** | XML | no | 7 | 444 | 20 |
| 50 | **Comm.Math** | Other | no | 6 | 1608 | 487 |
| 51 | **Jaxen** | XML | no | 6 | 93 | 19 |
| 52 | **Comm.Email** | Networking | no | 5 | 64 | 28 |
| 53 | **XOM** | XML | no | 5 | 5801 | 98 |
| 54 | **Comm.DbUtils** | Database | no | 5 | 196 | 13 |
| 55 | **Axis2** | Webservices | no | 5 | 18794 | 164 |
| 56 | **GNU Trove** | Collections | no | 4 | 105 | 44 |
| 57 | **AXIOM** | XML | no | 4 | 3010 | 65 |
| 58 | **j2ssh** | Networking | no | 4 | 4238 | 1075 |
| 59 | **Comm.Betwixt** | Other | no | 3 | 209 | 28 |
| 60 | **OFBiz** | e-Business | no | 3 | 170713 | 6781 |
| 61 | **Xalan** | XML | no | 3 | 6 | 2 |
| 62 | **Java Expression Language** | Other | no | 2 | 9 | 6 |
| 63 | **StAX** | XML | no | 2 | 16 | 9 |
| 64 | **Struts2** | Web Apps | no | 2 | 25 | 8 |
| 65 | **Express4J** | GUI | no | 2 | 31331 | 4417 |
| 66 | **Guice** | Other | no | 2 | 50 | 9 |
| 67 | **Comm.Discovery** | Other | no | 2 | 11 | 3 |
| 68 | **WSMO4J** | Webservices | no | 2 | 774 | 155 |
| 69 | **QuickFIX** | e-Business | no | 1 | 62 | 30 |
| 70 | **Comm.Transaction** | Other | no | 1 | 6 | 4 |
| 71 | **Comm.Chain** | Other | no | 1 | 12 | 6 |

Continued on next page

**Table A.4 – continued from previous page**

| | API | Domain | Core | # Projects | # Calls | # Methods called |
|---|---|---|---|---|---|---|
| 72 | **Comm.EL** | Other | no | 0 | 0 | 0 |
| 73 | **Comm.Daemon** | Other | no | 0 | 0 | 0 |
| 74 | **Comm.Exec** | Other | no | 0 | 0 | 0 |
| 75 | **Comm.Proxy** | Other | no | 0 | 0 | 0 |
| 76 | **Comm.Primitives** | Other | no | 0 | 0 | 0 |
| 77 | **Comm.Attributes** | Other | no | 0 | 0 | 0 |

Table A.4: List of the known Java APIs



| Projects | Min | 1st Q | Median | Mean | 3rd Q | Max |
|---|---|---|---|---|---|---|
| All | 0.02 | 0.45 | 0.56 | 0.56 | 0.67 | 1 |
| Reference | 0.11 | 0.43 | 0.5 | 0.5 | 0.57 | 0.99 |

**Figure A.6.** Ratio of Java API method calls to all method calls

## Additional information on Java API usage in projects

Fig. A.6 shows the usage of known Java API methods relative to all methods in a project—both in terms of calls. The smaller the ratio (the closer to zero), the lower the contribution of API calls. The quartiles show that in most projects, about each second method call is an API call. As far as instance-method calls are concerned, the figure distinguishes API vs. project-based method calls solely on the grounds of the static receiver type of methods.

In [186], library reuse is studied at a level of shared objects in the operating systems Sun OS and Mac OS X. One of the observations is that reuse seems to be low in the sense of Zipf's law. Thus the most frequent function will be referenced approximately twice as often as the second most frequent function, which occurs twice as often as the fourth most frequent function, etc. Fig. A.7 shows the distribution of frequency of method calls in the corpus. Only 0.03% of methods are called more than 10,000 times, while 98.2% of methods are called less than 100 times. The plot suggests a Zipf-style distribution.

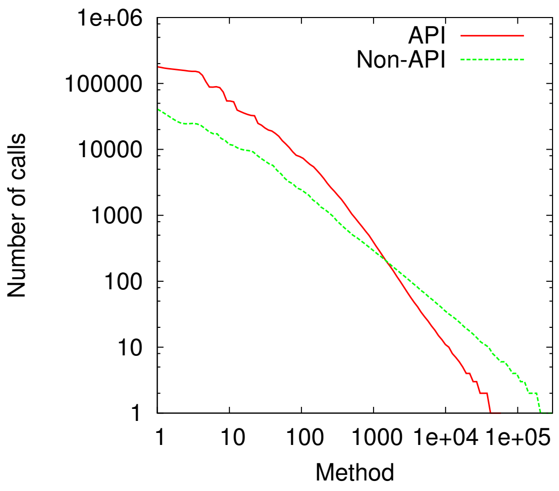


**Figure A.7.** Frequency of calling Java API vs. non-API methods

## Additional information on framework-like usage of Java APIs

Fig. A.8 shows the relative frequency of API-interface implementations for all the known Java APIs (including Core APIs). The picture is dominated by AWT handler types, the interface for iterators, and a few XML-related types.

Fig. A.9 shows the relative frequency of Java API-class extensions. The picture is entirely dominated by Swing's types for GUIs. Many other API types are implemented or extended, but only with a marginal frequency.

## Full list of Java APIs with framework-like usage

Table A.5 provides the complete list of Java APIs with detected framework-like usage.

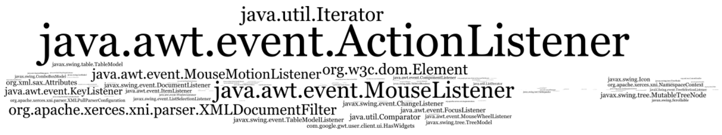


**Figure A.8.** Tag cloud of implemented Java API interfaces

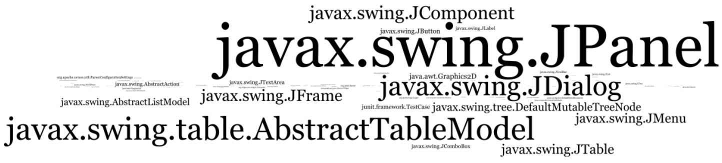


**Figure A.9.** Tag cloud of overridden Java API classes

| API | # projects | | | # methods | | # dist. methods | | # derived types | | # API types | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | impl | ext | any | impl | over | impl | over | int | cl | int | cl |
| **Swing** | 173 | 381 | 391 | 2,512 | 11,150 | 305 | 645 | 443 | 1,859 | 39 | 92 |
| **AWT** | 194 | 75 | 225 | 4,201 | 756 | 593 | 176 | 651 | 120 | 31 | 24 |
| **Java Collections** | 120 | 0 | 120 | 986 | 0 | 16 | 0 | 208 | 0 | 3 | 0 |
| **SAX** | 28 | 21 | 42 | 428 | 90 | 85 | 21 | 37 | 29 | 12 | 3 |
| **JUnit** | 3 | 38 | 40 | 4 | 344 | 4 | 19 | 3 | 46 | 2 | 2 |
| **Core XML** | 11 | 5 | 14 | 89 | 13 | 17 | 4 | 14 | 5 | 9 | 3 |
| **SWT** | 5 | 8 | 10 | 37 | 86 | 4 | 13 | 25 | 11 | 3 | 3 |
| **log4j** | 1 | 8 | 8 | 25 | 87 | 7 | 9 | 2 | 9 | 2 | 3 |
| **Reflection** | 7 | 0 | 7 | 10 | 0 | 1 | 0 | 7 | 0 | 1 | 0 |
| **JMF** | 4 | 2 | 6 | 8 | 6 | 6 | 3 | 4 | 3 | 3 | 3 |
| **DOM** | 6 | 0 | 6 | 572 | 0 | 41 | 0 | 15 | 0 | 10 | 0 |
| **GWT** | 5 | 6 | 6 | 442 | 143 | 59 | 36 | 18 | 26 | 22 | 11 |
| **Hibernate** | 5 | 1 | 5 | 136 | 1 | 78 | 1 | 17 | 1 | 12 | 1 |
| **Lucene** | 1 | 5 | 5 | 1 | 22 | 1 | 11 | 1 | 13 | 1 | 7 |
| **Xerces** | 4 | 3 | 4 | 915 | 143 | 46 | 6 | 9 | 1 | 8 | 1 |
| **Axis** | 2 | 1 | 3 | 8 | 4 | 2 | 1 | 5 | 1 | 1 | 1 |
| **JNDI** | 3 | 0 | 3 | 52 | 0 | 18 | 0 | 3 | 0 | 3 | 0 |
| **Struts** | 0 | 2 | 2 | 0 | 14 | 0 | 2 | 0 | 2 | 0 | 2 |
| **Commons Beanutils** | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 |
| **JFace** | 0 | 1 | 1 | 0 | 4 | 0 | 1 | 0 | 1 | 0 | 1 |
| **RMI** | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 |
| **Commons Collections** | 1 | 0 | 1 | 2 | 0 | 1 | 0 | 2 | 0 | 1 | 0 |
| **jMock2** | 0 | 1 | 1 | 0 | 7 | 0 | 1 | 0 | 1 | 0 | 1 |
| **GNU Trove** | 0 | 1 | 1 | 0 | 5 | 0 | 2 | 0 | 1 | 0 | 1 |
| **Commons Digester** | 0 | 1 | 1 | 0 | 2 | 0 | 1 | 0 | 1 | 0 | 1 |
| **Commons Logging** | 1 | 0 | 1 | 18 | 0 | 1 | 0 | 1 | 0 | 1 | 0 |
| **Bouncy Castle Crypto** | 0 | 1 | 1 | 0 | 2 | 0 | 1 | 0 | 1 | 0 | 1 |
| **Jena** | 1 | 0 | 1 | 3 | 0 | 1 | 0 | 1 | 0 | 1 | 0 |
| **Commons Pool** | 0 | 1 | 1 | 0 | 3 | 0 | 2 | 0 | 1 | 0 | 1 |
| **Commons Chain** | 1 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 |
| **Commons DbUtils** | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 |
| **Commons Lang** | 0 | 1 | 1 | 0 | 4 | 0 | 3 | 0 | 2 | 0 | 1 |
| **Berkeley DB** | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 |
| **Commons Net** | 0 | 1 | 1 | 0 | 3 | 0 | 2 | 0 | 1 | 0 | 1 |
| **LWJGL** | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 |

**Table A.5.** Full list of Java APIs with framework-like usage

### A.2.2 Additional Information on .NET Framework

This part of the appendix is, unfortunately, not entirely printer-friendly. We urge the reader to *view data on the screen*, thereby being able to see the details in the tables and figures. (Please use zooming and rotation features of your PDF viewer.) We do apologize for the inconvenience.

**Reuse-related metrics for frameworks**

We add definitions of a few metrics that we only hinted at in §4.3.2.

Specializability is taken to the method level as follows. Each namespace can be measured in terms of **% Specializable methods**, i.e., the percentage of all methods that are either abstract or non-sealed methods—hence excluding static and sealed as well as non-virtual methods. A metrics is added for **% Sealed methods**, i.e., the percentage of "non-overridable virtual methods"—which is the percentage of all virtual, non-abstract instance method declarations that are sealed (including the case that the hosting class is sealed entirely).

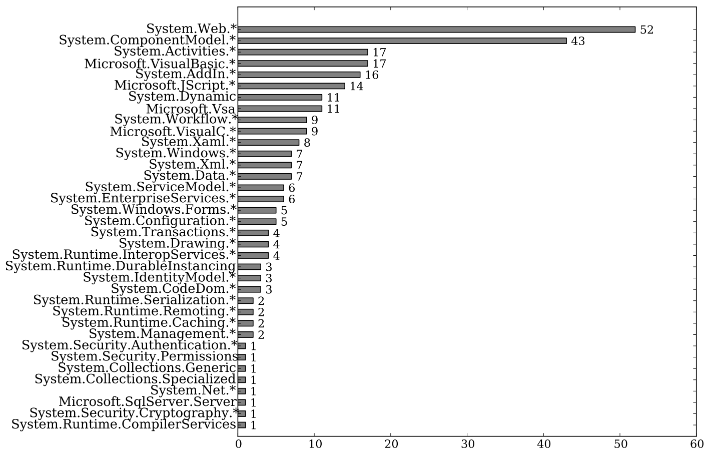


**Figure A.10.** .NET namespaces sorted by the number of orphan types

The notion of orphan types can also be refined as follows. That is, types may be orphaned in a more inclusive sense if we focus specifically on composite frameworks. There is the variation **% Local orphan classes**: the percentage of all abstract classes in a given namespace that are never concretely implemented within the given namespace. Likewise, there is the variation **% Local orphan interfaces**. These metrics show us whether there are namespaces that are incomplete by themselves while they are 'fully illustrated' by other namespaces so that they do not count as hosting 'global' orhpans.

More detailed information about orphan types in .NET is provided by Figure A.10.

| Namespace | System.ServiceModel.* | System.Web.* | Microsoft.VisualBasic.* | System.Workflow.* | System.Activities.* | System.Data.* | System.Windows.Forms.* | System.Net.* | System.ComponentModel.* | System | System.Runtime.Remoting.* | System.Xml.* | System.IdentityModel.* | Microsoft.JScript.* | System.DirectoryServices.* | System.Configuration.* | System.Security.Cryptography.* | System.Deployment.* | System.Diagnostics.* | System.IO.* | System.Runtime.Serialization.* | System.Resources.* | System.Messaging.* | System.CodeDom.* | System.Security.Policy | System.Runtime | System.EnterpriseServices.* | System.Xaml.* | System.Security | Microsoft.Win32.* | System.Threading.* | System.AddIn.* | System.Management.* | System.Reflection.* | System.Drawing.* | System.Security.Permissions | System.ServiceProcess.* | System.Runtime.InteropServices.* | System.Dynamic | Microsoft.CSharp.* | System.Security.Principal | System.Security.AccessControl | System.Linq.* | System.Transactions.* | System.Runtime.CompilerServices | System.Globalization | System.Security.Authentication.* | System.Runtime.Caching.* | System.Windows.* | System.Collections.Generic | System.Text.RegularExpressions | System.Collections.ObjectModel | System.Collections.Concurrent | Microsoft.SqlServer.Server | System.Collections.Specialized | System.Runtime.DurableInstancing | System.Text | System.Media | System.Collections | System.Runtime.Versioning | Microsoft.VisualC.* | System.Runtime.Hosting | System.Device.Location | System.Timers | System.Numerics | System.Runtime.ConstrainedExecution | System.Runtime.ExceptionServices |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| System.ServiceModel.* |⇌|⇌|●|⇌|●|⇌||●|●|⇌|●|⇌|⇌||●|●|●||●|⇌|⇌||●|●||●|●|●|●|●|●|||●||●|●|●|||●||●|●|●|●|●||●|⇌|●|●|||●|●|●||●|||||||||
| System.Web.* |⇌|⇌||||⇌|●|●|●|●|⇌|●|||●|●|●||●|●|●|●||●|●||●||●|●|●|||●|●|●|●|●||●|●|●|●||●|●|●|●||●|●|●|●||●||●||●|●||||||||
| Microsoft.VisualBasic.* ||●|⇌||⇌||●|●|●|●|●|●||||||●|●|●|●|●||●||●||●|●|●|●||●|●|●|●||●|●||●|●|●||●|●|||●|●|●|●|||●||●|●|●||||●||||
| System.Workflow.* |⇌|●||⇌||●|●|●|●|●|●|●|●||●|●|●||●|●|●|●||●||●||||●|●|||●|●|●||●|||●|||●|●|●||||●|●|●|||●||●||●|●||||●||||
| System.Activities.* |||⇌|●|⇌|●|||●|●||●|||||●||●|●|●|||●||●||●||●|●|||●|●|||●|●|||●|●|●|●|||●|●|●|●|●||●|●|●||●||||●||||
| System.Data.* |⇌|⇌|●|||⇌||●|⇌|●|●|⇌||||●|●||●|●|●|||●|||●||●|●|●|||●||●||●||●|||●|●|●|●||||●|●|●||⇌|●||●||●||||||●|||
| System.Windows.Forms.* ||||||●|⇌|●|⇌|●||●||||●||⇌|⇌|●|●|||●|⇌||||●|●|●|||●|⇌|●||●||●|||||●|●||||●|●|●|||●||●|●|●||||||||
| System.Net.* ||||||||⇌|●|⇌||⇌||●||●|⇌||●|●|●||||●||||●|⇌|●|||●||●||●|||●||||●|●|⇌|||●|●|●|||●||●||●|●||||||||
| System.ComponentModel.* ||||||⇌|⇌||⇌|⇌||||||⇌|||⇌|⇌|●|⇌||⇌|||||●|⇌|●|||●|⇌|●||●||⇌|||●||⇌|●||||●|⇌|⇌|||⇌||●||●||||||||
| System |⇌|||||||⇌|⇌|⇌|⇌|||||⇌|||⇌|⇌|⇌||||⇌|⇌|||⇌|⇌|⇌|||⇌||⇌||⇌|||⇌||||⇌|⇌||||⇌||⇌|||⇌||⇌||⇌|||⇌||||||
| System.Runtime.Remoting.* ||⇌||||||●|●|⇌|⇌|●|||||●|||●|⇌|●||●|||||●|●|⇌|||●||●||⇌||⇌|●|●|||●|●||||●|||||●||●||●|||||||||
| System.Xml.* |⇌|||||⇌||⇌|●|●||⇌||||⇌|||⇌|●|⇌|||●|⇌|⇌|||●||●|||●||●||||●|||●||●|●||||●|●||||●||●||●|||||||||
| System.IdentityModel.* |⇌|●||||||●|●|●|●|●|⇌||||●|||●|●|||||●||||●|●|||●||||●|||●||||●|●|●|||●|●|●|||||●||●|||||||||
| Microsoft.JScript.* |||||||||●|●|●|●||⇌||●|●||●|●|●|●||●|●||||●|●|●|||●||●||●|||||||●|●||||||●||||||●||●|||||||||
| System.DirectoryServices.* ||||||||●|●|●||●|||⇌||●||●|●|●||||||||●|||●|||●||●||●|||●|●|||●|●|●|||●|●||||●||●||●|||||||||
| System.Configuration.* |||||||||⇌|⇌||⇌||||⇌|●|●|⇌|●|⇌||||||||●|●|●|||⇌||●||●|||||||●|●||||●|●||||●||●||●|●||||||||
| System.Security.Cryptography.* ||||||||⇌||●||●|||||⇌|⇌|●|⇌|●||||⇌||||⇌|●|●|||⇌||⇌||●||||●|||●|●||||●||●|||||●||●||||||●|||
| System.Deployment.* |||||||⇌|●|●|●|●|●|||||⇌|⇌||●|●||||⇌||||●|●|●|||●|●|||●|||||||●||||●|||||||●||●|||●||||||
| System.Diagnostics.* |||||||⇌||⇌|⇌|●|⇌||||⇌|||⇌|⇌|●||||||||●|⇌|⇌|||⇌||●||●|||●||||⇌|●||||⇌|||||●||●||●|||||||||
| System.IO.* |⇌||||||||⇌|⇌|||||||⇌||⇌|⇌|⇌||||⇌|⇌|||⇌|⇌|⇌|||⇌||⇌||⇌|||●|⇌|||●|●||||●|||||||●||●|||||||||
| System.Runtime.Serialization.* |⇌|||||||||⇌|⇌|⇌||||⇌||||⇌|⇌|||●||●|||⇌||⇌|||⇌||⇌||●|||||||⇌|⇌||||⇌|⇌|●|||||⇌||⇌|||||||||
| System.Resources.* |||||||●||⇌|●||●|||||||●|●|●|⇌||●|||||●||●|||⇌|●|●||●|||||||||●||||●|||||●||●||●|●||||||||
| System.Messaging.* |||||||●||●|●||●|||●|●||||●|●|●|⇌||||||●||●|||●|●|●||●|||||||●||●||||||||||●||●||●|||||||||
| System.CodeDom.* |||||||||⇌|●||||||●|||●|●||||⇌|●||||●|●|●|||●||●||●|||●||||●|●||||●|||||●||●||●|||||||||
| System.Security.Policy |||||||⇌|||⇌||⇌|||||⇌|⇌||⇌|●||||⇌||||⇌|●|●|||⇌||⇌||●|||||||●||||●|||||||●||●|||⇌||||||
| System.Runtime |||||||||●|⇌||⇌||||||||⇌||||||⇌|||●|●|●|||●||||●|||||●|●|●|||||●|||||●|⇌|●||●|||||||●||
| System.EnterpriseServices.* ||||||||●||●|●|●|||●||||●|●|●||||||⇌||●|●|●|||●||●||●|||||||⇌||●||||||||||||●||●|||||||||
| System.Xaml.* |||||||||●|●||●|||||||●|●|●|||||||⇌|●||●|||●||●||||||||||●|||⇌|●||●|●||●||●||●|||||||||
| System.Security ||||||||||⇌|||||||⇌|||⇌|⇌||||⇌||||⇌|⇌|⇌|||⇌||⇌||⇌|||⇌||||●|⇌||||●|||||||●||●|||⇌||||||
| Microsoft.Win32.* ||||||||⇌|⇌|⇌|||||||||⇌|⇌|||||||||⇌|⇌|⇌|||●||●||⇌||||⇌|||●|⇌||||●|||||●||⇌||●|||||||||
| System.Threading.* ||||||||||⇌|⇌||||||||⇌|⇌|⇌||||||||⇌|⇌|⇌|||||||⇌|||⇌|⇌|||⇌|⇌||||⇌||⇌|⇌||||||⇌||||||||●||
| System.AddIn.* |||||||||●|●|●||||||||●|●|●||||●||||●||●|⇌||●||●||●||||||||●||||●||●|||||●||●|||||||||
| System.Management.* |||||||||●|●||||||●|||●|●|●|||●|||||●||●||⇌|●||●||●||||||||●|||||●||||●||●||●|||||||||
| System.Reflection.* ||||||||||⇌||||||⇌|⇌||⇌|⇌|⇌|⇌|||⇌||||⇌||●|||⇌||⇌||⇌|||||||⇌|⇌||||⇌|||||||⇌|||||||||||
| System.Drawing.* |||||||⇌||⇌|●|●||||||||●|●|●||||||||●||●|||●|⇌|●||●||||||||●||||●|||||||●||●|||||||||
| System.Security.Permissions ||||||||||⇌|||||||⇌|||⇌|⇌||||⇌||||⇌||●|||⇌||⇌||⇌|||●|⇌||||⇌||||●|||||||●||●|||||||||
| System.ServiceProcess.* |||||||●||●|●||||||●|||●|||||||||●||●|||●|●|●|⇌|●||||||||●||||||||||●||●||●|||||||||
| System.Runtime.InteropServices.* ||||||||||⇌|⇌|||||||||⇌|||||||||⇌|⇌|⇌|||⇌||⇌||⇌|||||||⇌|⇌||||⇌|||||||⇌||⇌||||||||⇌||
| System.Dynamic |||||||||●|●|●||||||||●||||||||||●||●|||●||||●|⇌||||⇌||⇌|●||||●||●|||||||●|||||||||
| Microsoft.CSharp.* |||||||||⇌|●|⇌||||||||||●||||●|||||||●|||●||||●|●|⇌|||●||●|||||●|||||||||●|||||||||
| System.Security.Principal ||||||||||⇌|||||||||||●||||||||⇌|●|⇌|||●||||●|||⇌|⇌|||●|●||||●|||||||●||●|||||||||
| System.Security.AccessControl ||||||||||●||||||||||⇌|●||||||||●|⇌|⇌|||●||⇌||●|||⇌|⇌|||●|●||||||||||||||●|||||||||
| System.Linq.* ||||||||||●|||||||||●|●|||||||||||●|||●|||||⇌||||⇌||⇌|●||||●||●|●||||●||●|||||||||
| System.Transactions.* ||||||||||●||||||●||||●|●||||||⇌||●||●|||||●||●|||||||⇌|●|●|||||||||||●||●|||||||||
| System.Runtime.CompilerServices |||||||||⇌|⇌|||||||||⇌||⇌||||||||||⇌|||⇌||||⇌|⇌||||⇌||⇌|||||⇌||⇌|||||||●|||||||||
| System.Globalization ||||||||||⇌|||||||||||⇌||||||||⇌|⇌|⇌|||⇌||⇌||⇌|||||||●|⇌||||●|||||||⇌||⇌|||||||||
| System.Security.Authentication.* ||||||||⇌|●|●||||||●|||||●|||||||||●||||●||||●||||||||●|⇌|||●|||||||●||●|||||||||
| System.Runtime.Caching.* ||||||●||||●||||||●||||●|||||||||||●|||●|||||||||||●||⇌||●||●|||●||●||●|||||||||
| System.Windows.* |||||||||●|●||●||||||||●||||||||⇌|||●|||●|||||||||||●|●|||⇌|●|||||||●||●|||||||||
| System.Collections.Generic |⇌|||||||||⇌|||||||||⇌||⇌||||||||||⇌|||⇌||||⇌|||||||⇌|||||⇌||⇌|||||●||⇌|||||||||
| System.Text.RegularExpressions |||||||||⇌|●|||||||||||⇌||||||||●||●|||●||●||||||||||●||||●|⇌||||||●||●|||||||||
| System.Collections.ObjectModel |||||||||⇌|⇌|||||||||●|||||||||||||⇌|||●||||●|||||||⇌|||||⇌||⇌|||●||||●|||||||||
| System.Collections.Concurrent ||||||||||●|||||||||●||●||||||||||⇌|||●||||●||||||||●||||●||●|⇌||||||●|||||||||
| Microsoft.SqlServer.Server ||||||⇌||||●||||||||||●|●||||||||||||||●|||||||●|||||●||||●||||⇌|||●||●|||||||||
| System.Collections.Specialized |||||||||⇌|⇌||||||||||||●||||||||||●|||●|||||||||||●|●||||●|||||⇌||●||●|||||||||
| System.Runtime.DurableInstancing ||||||||||●||●||||||||||●|||||⇌|||||●|||||||||||●|●||||||●||●||||⇌|||●|||||||||
| System.Text ||||||||||⇌|||||||||||⇌||||||||||⇌|●|||⇌||||⇌|||||||●|⇌||||||||||||⇌||●|||||||||
| System.Media ||||||||●|●|●||||||||||●|●||||||||●|||●|||||●||●|||||||||||||||||||||||⇌||||||||||||
| System.Collections ||||||||||⇌|||||||||⇌||⇌||||||||||⇌|||●||||⇌|||||||⇌||||⇌|||||||||⇌|||||||||
| System.Runtime.Versioning ||||||||||●|||||||||●||||||||||||●||||●||||||||||●|||||||||||●|||⇌||||||||
| Microsoft.VisualC.* ||||||||||●|||||||||●||||||||||||||●|||||||||||||||||●|||||||||●||⇌|||||||
| System.Runtime.Hosting ||||||||||⇌|●||||||||||||||⇌||||⇌||||||||●||||||||||||||||||||||||||||⇌||||||
| System.Device.Location |||||||||●|●|||||||||||||||||||●||●|||||||||||||||●||||||||||||||||⇌|||||
| System.Timers |||||||||●|●|||||||||||||||||||||●|||●||||||||||||||||||||||||||||||⇌||||
| System.Numerics ||||||||||●|||||||||||||||||||||||||||||||||||||●|||||||||||●|||||||⇌|||
| System.Runtime.ConstrainedExecution ||||||||||●|||||||||||||||||||||||||||||⇌||||||||||||||||||||||||||||||⇌||
| System.Runtime.ExceptionServices ||||||||||●|||||||||||||||||||||||||||||||||||||||||||||||||||||||||||⇌|

**Table A.6.** Inter-namespace referencing within the .NET Framework

| Namespace | System.Web.* | System.ServiceModel.* | System.Data.* | System.Workflow.* | System.Net.* | System.Activities.* | System.Windows.Forms.* | Microsoft.VisualBasic.* | System.ComponentModel.* | System.Messaging.* | System.DirectoryServices.* | System | System.IO.* | System.Configuration.* | System.Xaml.* | Microsoft.JScript.* | System.Runtime.Serialization.* | System.Runtime.Remoting.* | System.Diagnostics.* | System.Management.* | System.Dynamic | System.Xml.* | System.Security.Authentication.* | System.Drawing.* | System.Deployment.* | System.Threading.* | System.Security.Cryptography.* | System.ServiceProcess.* | System.Transactions.* | System.Runtime.InteropServices.* | System.Collections.Generic | System.Security.Policy | System.Reflection.* | System.Collections.ObjectModel | System.Resources.* | System.Security.Principal | System.AddIn.* | System.CodeDom.* | System.IdentityModel.* | System.Runtime.Caching.* | System.Windows.* | Microsoft.VisualC.* | System.Security | System.Runtime.CompilerServices | Microsoft.Win32.* | System.Collections.Concurrent | System.Globalization | System.Security.Permissions | System.Security.AccessControl | System.Text.RegularExpressions | System.Collections.Specialized | System.Collections | System.Text | System.Linq.* | Microsoft.SqlServer.Server | System.Device.Location | System.EnterpriseServices.* | Microsoft.CSharp.* | System.Runtime | System.Media | System.Timers | System.Runtime.DurableInstancing | System.Runtime.Hosting | System.Runtime.ConstrainedExecution | System.Runtime.ExceptionServices | System.Runtime.Versioning | System.Numerics |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| System.Web.* | ⇌ | ⇌ |  |  |  |  | ● |  | ● |  |  | ● | ● | ● |  |  | ● |  | ● |  |  | ● |  | ● |  |  |  |  |  | ● | ● |  |  | ● | ● | ● |  |  |  | ● |  |  | ● |  |  |  |  | ● |  | ● | ● | ● |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| System.ServiceModel.* | ⇌ | ⇌ |  | ⇌ |  | ● |  |  | ● |  |  | ● | ● | ● |  |  | ● | ● |  |  |  | ● |  |  |  | ● |  |  |  | ● | ● |  |  | ● |  |  |  |  | ● |  | ● |  |  |  |  |  |  |  |  |  |  | ● |  |  |  |  | ● |  | ● |  |  | ● |  |  |  |  |  |
| System.Data.* | ● | ● | ⇌ |  |  |  |  |  | ● |  |  | ● | ● | ● |  |  | ● |  |  |  |  | ● |  |  |  |  |  |  |  | ● | ● |  |  | ● |  |  |  |  |  |  |  |  | ● |  |  |  |  | ● |  |  |  | ● |  | ● |  |  |  |  |  |  |  |  |  |  |  |  |  |
| System.Workflow.* | ● | ⇌ |  | ⇌ |  |  | ● |  | ● |  |  | ● |  | ● |  |  | ● | ● |  |  |  |  |  | ● |  |  |  |  |  |  | ● |  | ● | ● |  |  |  | ● |  |  |  |  |  |  |  |  |  |  |  |  |  | ● |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| System.Net.* |  |  |  |  | ⇌ |  |  |  | ● |  |  | ● | ● | ● |  |  | ● |  |  |  |  |  |  |  |  |  |  |  |  |  | ● |  | ● | ● |  | ● |  |  |  |  |  |  | ● |  | ⇌ |  |  | ● |  |  | ● | ● |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| System.Activities.* |  |  |  | ● |  | ⇌ |  |  | ● |  |  | ● |  |  | ● |  | ● |  |  |  | ● |  |  |  |  |  |  |  |  |  | ● |  |  |  |  |  |  |  |  |  | ● |  |  |  |  |  |  |  |  |  | ● | ● |  |  |  |  |  |  | ● |  |  | ● |  |  |  |  |  |
| System.Windows.Forms.* |  |  |  |  |  |  | ⇌ |  | ⇌ |  |  | ● |  | ● |  |  | ● |  |  |  |  |  |  | ● |  | ● |  |  |  | ● | ● |  | ● | ● |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | ● |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Microsoft.VisualBasic.* |  |  |  |  |  | ● | ● | ⇌ | ● |  |  | ● |  |  |  |  | ● |  | ● |  |  |  |  |  |  | ● |  |  |  |  |  |  | ● |  |  |  |  | ● |  |  | ● |  |  |  |  |  |  |  |  |  |  | ● |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| System.ComponentModel.* |  |  |  |  |  |  | ⇌ |  | ⇌ |  |  | ⇌ |  |  |  |  | ● |  |  |  |  |  |  | ⇌ |  |  |  |  |  | ● | ● |  | ● | ⇌ | ● |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | ● |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| System.Messaging.* |  |  |  |  |  |  | ● |  | ● | ⇌ |  | ● |  | ● |  |  | ● |  |  |  |  |  |  | ● |  |  |  |  |  | ● |  |  |  |  |  |  |  |  |  |  |  |  | ● |  |  |  |  | ● |  |  |  | ● |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| System.DirectoryServices.* |  |  |  |  |  |  |  |  | ● |  | ⇌ | ● |  |  |  |  | ● |  |  |  |  |  |  |  |  |  |  |  |  | ● | ● |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | ● | ● |  |  | ● |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| System |  |  |  |  |  |  |  |  | ⇌ |  |  | ⇌ |  |  |  |  | ⇌ |  |  |  |  |  |  |  |  |  |  |  |  | ⇌ | ⇌ |  | ⇌ |  |  |  |  |  |  |  |  |  | ⇌ |  |  |  |  |  |  |  |  | ⇌ |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| System.IO.* |  |  |  |  |  |  |  |  | ● |  |  | ● | ⇌ |  |  |  | ⇌ |  |  |  |  |  |  |  |  |  |  |  |  |  | ● |  |  |  |  |  |  |  |  |  |  |  | ● |  |  |  |  |  | ● |  |  | ● |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| System.Configuration.* |  |  |  |  |  |  |  |  | ● |  |  | ● |  | ⇌ |  |  |  |  |  |  |  | ⇌ |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | ● |  |  |  |  | ● |  |  | ● | ● |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| System.Xaml.* |  |  |  |  |  |  |  |  | ● |  |  | ● |  |  | ⇌ |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | ● |  |  |  |  |  |  |  |  |  | ● |  | ● |  |  |  |  | ● |  |  |  | ● |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Microsoft.JScript.* |  |  |  |  |  |  |  |  |  |  |  | ● | ● |  |  | ⇌ |  |  |  |  |  |  |  |  |  |  |  |  |  | ● |  |  | ● |  |  |  |  | ● |  |  |  |  |  |  |  |  |  |  |  |  |  | ● |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| System.Runtime.Serialization.* |  |  |  |  |  |  |  |  |  |  |  | ⇌ | ⇌ | ● |  |  | ⇌ | ⇌ |  |  |  | ⇌ |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | ⇌ |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| System.Runtime.Remoting.* | ● |  |  |  |  |  |  |  | ● |  |  | ● | ● |  |  |  | ⇌ | ⇌ |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | ● |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| System.Diagnostics.* |  |  |  |  |  |  |  |  | ● |  |  | ● |  | ● |  |  | ● |  | ⇌ |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | ● |  |  |  | ● |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| System.Management.* |  |  |  |  |  |  |  |  | ● |  |  | ● |  | ● |  |  | ● |  |  | ⇌ |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | ● | ● |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| System.Dynamic |  |  |  |  |  |  |  |  | ● |  |  | ● |  |  |  |  |  |  |  |  | ⇌ |  |  |  |  |  |  |  |  |  | ● |  | ● |  |  |  |  |  |  |  |  |  |  | ● |  |  |  |  |  |  |  | ● |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| System.Xml.* |  |  |  |  |  |  |  |  |  |  |  | ● | ● | ⇌ |  |  | ⇌ |  |  |  |  | ⇌ |  |  |  |  |  |  |  |  | ● |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | ● |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| System.Security.Authentication.* |  |  |  |  |  |  |  |  | ● |  |  | ● |  | ● |  |  | ● |  |  |  |  |  | ⇌ |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | ● |  |  |  |  |  |  | ● |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| System.Drawing.* |  |  |  |  |  |  |  |  | ⇌ |  |  | ● |  |  |  |  | ● |  |  |  |  |  |  | ⇌ |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | ● |  |  |  |  | ● |  |  |  | ● |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| System.Deployment.* |  |  |  |  |  |  | ● |  | ● |  |  | ● | ● |  |  |  |  |  |  |  |  |  |  |  | ⇌ |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | ● |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| System.Threading.* |  |  |  |  |  |  |  |  |  |  |  | ● |  |  |  |  | ● |  | ● |  |  |  |  |  |  | ⇌ |  |  |  | ● |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | ● |  |  |  |
| System.Security.Cryptography.* |  |  |  |  |  |  |  |  |  |  |  | ● | ● |  |  |  | ● |  |  |  |  |  |  |  |  |  | ⇌ |  |  |  |  |  |  | ● |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | ● |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| System.ServiceProcess.* |  |  |  |  |  |  | ● |  | ● |  |  | ● |  | ● |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | ● |  |  |  | ● |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| System.Transactions.* |  |  |  |  |  |  |  |  |  |  |  | ● |  | ● |  |  | ● |  |  |  |  |  |  |  |  |  |  |  | ⇌ |  |  |  |  |  |  |  |  |  |  |  |  |  | ● |  |  |  |  | ● |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| System.Runtime.InteropServices.* |  |  |  |  |  |  |  |  |  |  |  | ⇌ |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | ⇌ |  |  | ⇌ |  |  |  |  |  |  |  |  |  |  |  | ⇌ |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | ● |  |  |  |
| System.Collections.Generic |  |  |  |  |  |  |  |  |  |  |  | ⇌ |  |  |  |  | ● |  |  |  |  |  |  |  |  |  |  |  |  |  | ⇌ |  |  | ⇌ |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | ● |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| System.Security.Policy |  |  |  |  |  |  |  |  |  |  |  | ● |  |  |  |  | ● |  |  |  |  |  |  |  |  |  |  |  |  |  |  | ⇌ |  |  |  |  |  |  |  |  |  |  | ● |  |  |  |  |  |  |  |  | ● |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| System.Reflection.* |  |  |  |  |  |  |  |  |  |  |  | ⇌ |  |  |  |  | ● |  |  |  |  |  |  |  |  |  |  |  |  | ⇌ |  |  | ⇌ |  |  |  |  |  |  |  |  |  | ● |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| System.Collections.ObjectModel |  |  |  |  |  |  |  |  | ⇌ |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | ⇌ |  |  | ⇌ |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | ● | ● |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| System.Resources.* |  |  |  |  |  |  |  |  |  |  |  | ● |  |  |  |  | ● |  |  |  |  |  |  |  |  |  |  |  |  |  | ● |  |  |  | ⇌ |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | ● |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| System.Security.Principal |  |  |  |  |  |  |  |  |  |  |  | ● |  |  |  |  | ● |  |  |  |  |  |  |  |  |  |  |  |  |  | ● |  |  |  |  | ⇌ |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | ● |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| System.AddIn.* |  |  |  |  |  |  |  |  |  |  |  | ● |  |  |  |  |  | ● |  |  |  |  |  |  |  |  |  |  |  |  | ● |  |  |  |  |  | ⇌ |  |  |  |  |  |  |  |  |  |  |  |  |  |  | ● |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| System.CodeDom.* |  |  |  |  |  |  |  |  | ● |  |  | ● | ● |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | ⇌ |  |  |  |  |  |  |  |  |  |  |  |  |  | ● |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| System.IdentityModel.* |  |  |  |  |  |  |  |  |  |  |  | ● |  |  |  |  |  |  |  |  |  |  |  |  |  |  | ● |  |  |  | ● |  |  |  |  |  |  |  | ⇌ |  |  |  |  |  |  |  |  |  |  |  |  | ● |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| System.Runtime.Caching.* |  |  |  |  |  |  |  |  |  |  |  | ● |  | ● |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | ● |  |  |  |  |  |  |  |  | ⇌ |  |  |  |  |  |  |  |  |  |  |  | ● |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| System.Windows.* |  |  |  |  |  |  |  |  | ● |  |  | ● |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | ● |  |  |  |  |  |  |  |  |  | ⇌ |  |  |  |  |  |  |  |  |  |  | ● |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Microsoft.VisualC.* |  |  |  |  |  |  |  |  |  |  |  | ● |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | ● |  |  |  |  |  |  |  |  |  |  | ⇌ |  |  |  |  |  |  |  |  |  | ● |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| System.Security |  |  |  |  |  |  |  |  |  |  |  | ⇌ |  |  |  |  | ● |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | ⇌ |  |  |  |  |  |  |  |  | ● |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| System.Runtime.CompilerServices |  |  |  |  |  |  |  |  |  |  |  | ● |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | ● |  |  |  |  |  |  |  |  |  |  |  |  | ⇌ |  |  |  |  |  |  |  | ● |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Microsoft.Win32.* |  |  |  |  | ⇌ |  |  |  |  |  |  | ● |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | ⇌ |  |  |  |  |  |  |  |  |  |  |  |  |  |  | ⇌ |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| System.Collections.Concurrent |  |  |  |  |  |  |  |  |  |  |  | ● |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | ● |  |  |  |  |  |  |  |  |  |  |  |  |  |  | ⇌ |  |  |  |  |  | ● |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| System.Globalization |  |  |  |  |  |  |  |  |  |  |  | ● |  |  |  |  | ● |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | ⇌ |  |  |  |  | ● |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| System.Security.Permissions |  |  |  |  |  |  |  |  |  |  |  | ● |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | ● |  |  |  |  | ⇌ |  |  |  | ● |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| System.Security.AccessControl |  |  |  |  |  |  |  |  |  |  |  | ● |  |  |  |  | ● |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | ⇌ |  |  | ● |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| System.Text.RegularExpressions |  |  |  |  |  |  |  |  |  |  |  | ● |  |  |  |  | ● |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | ⇌ |  | ● |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| System.Collections.Specialized |  |  |  |  |  |  |  |  |  |  |  | ● |  |  |  |  | ● |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | ⇌ | ● |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| System.Collections |  |  |  |  |  |  |  |  |  |  |  | ⇌ |  |  |  |  | ⇌ |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | ⇌ |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| System.Text |  |  |  |  |  |  |  |  |  |  |  | ● |  |  |  |  | ● |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | ⇌ |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| System.Linq.* |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | ● |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | ● |  | ⇌ |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Microsoft.SqlServer.Server |  |  | ● |  |  |  |  |  |  |  |  | ● |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | ⇌ |  |  |  |  |  |  |  |  |  |  |  |  |
| System.Device.Location |  |  |  |  |  |  |  |  | ● |  |  | ● |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | ⇌ |  |  |  |  |  |  |  |  |  |  |  |
| System.EnterpriseServices.* |  |  |  |  |  |  |  |  |  |  |  | ● |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | ● |  |  |  |  | ⇌ |  |  |  |  |  |  |  |  |  |  |
| Microsoft.CSharp.* |  |  |  |  |  |  |  |  |  |  |  | ● |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | ● |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| System.Runtime |  |  |  |  |  |  |  |  |  |  |  | ● |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | ● |  |  |  |
| System.Media |  |  |  |  |  |  |  |  | ● |  |  |  |  |  |  |  | ● |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| System.Timers |  |  |  |  |  |  |  |  | ● |  |  | ● |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| System.Runtime.DurableInstancing |  |  |  |  |  |  |  |  |  |  |  | ● |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | ⇌ |  |  |  |  |  |
| System.Runtime.Hosting |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | ● |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| System.Runtime.ConstrainedExecution |  |  |  |  |  |  |  |  |  |  |  | ● |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| System.Runtime.ExceptionServices |  |  |  |  |  |  |  |  |  |  |  | ● |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| System.Runtime.Versioning |  |  |  |  |  |  |  |  |  |  |  | ● |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| System.Numerics |  |  |  |  |  |  |  |  |  |  |  | ● |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |

**Table A.7.** Inter-namespace specialization within the .NET framework

Instead of unspecific usage, ‘inter-namespace specialization’ can also be considered. That is, each namespace can be measured in terms of **# Specialized namespaces**, i.e., the number of namespaces with at least one type that is specialized (implemented or extended) by a type of the given namespace versus **# Specializing namespaces**, i.e., the number of namespaces with at least one type that specialize a type of the given namespace. Here, direct relationships for references and specialization are counted, only—as opposed to taking the transitive closure of those relations.

Tables A.6 and A.7 provide additional views on the ’inter-namespace referencing’ and ’inter-namespace specialization’. As we can see from the ’specialization’ view, there are very few ’top’ namespaces which are heavily specialized; others have noticeable less specialization cases. This also to some extend proves the hypothesis that classic forms of OO-extensibility is not very much exercised within .NET Framework itself; they are rather observable on the limited set of very specific namespaces (e.g. collections).

Marks in Table A.6: • indicates that types from a horizontal namespace reference types from a vertical one; $\rightleftharpoons$ indicates that referencing happens in both ways. This also applies when types reference other types from the same namespace.

Marks in Table A.7: • indicates that types from a horizontal namespace specialize types from a vertical one. $\rightleftharpoons$ indicates that specialization happens in both ways. This also applies when types specialize other types from the same namespace.

## Additions to classification of frameworks

Table A.8 lists all namespaces, classified automatically based on definitions introduced in §4.3.3.

| **Namespace** | **Application** | **Core** | **Open** | **Closed** | **Incomplete** | **Branched** | **Flat** | **Interface-intensive** | **Delegate-intensive** | **Event-based** |
|---|---|---|---|---|---|---|---|---|---|---|
| | **4** | **18** | **20** | **20** | **19** | **20** | **20** | **19** | **21** | **12** |
| System.Web.* | | | | | | ✓ | | | ✓ | ✓ |
| System.Windows.* | | | | | | ✓ | | | ✓ | ✓ |
| System.ServiceModel.* | | | | | ✓ | ✓ | | | ✓ | |
| System.Windows.Forms.* | | | | | | ✓ | | | ✓ | ✓ |
| System.Data.* | | | | ✓ | | ✓ | | | | |
| System.Activities.* | | | | ✓ | | ✓ | | | ✓ | |
| System.ComponentModel.* | | ✓ | | | ✓ | ✓ | | ✓ | ✓ | ✓ |
| System.Workflow.* | | | | | | | | ✓ | ✓ | |
| System.Xml.* | | ✓ | | | | | | | | |
| System.Net.* | | | | | | | | | ✓ | ✓ |
| System.DirectoryServices.* | | | ✓ | | | ✓ | | | | |
| System | | ✓ | | | | ✓ | | ✓ | | |
| System.Security.Cryptography.* | | | | | | ✓ | | | | |

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Microsoft.VisualBasic.* | | | | | ✓ | ✓ | | | ✓ | |
| System.Runtime.InteropServices.* | | ✓ | | ✓ | ✓ | | | ✓ | | |
| Microsoft.JScript.* | | | | ✓ | ✓ | ✓ | | | | |
| System.Drawing.* | | | | ✓ | | | | | | |
| System.Runtime.Remoting.* | | | | | ✓ | | | ✓ | | |
| System.Configuration.* | | | | ✓ | | ✓ | | | | |
| System.Diagnostics.* | | ✓ | | | | | | | | |
| System.IO.* | | ✓ | | | | ✓ | | | | |
| System.Reflection.* | | ✓ | | ✓ | | | | | | |
| System.EnterpriseServices.* | | | | ✓ | | | ✓ | | | |
| System.CodeDom.* | | | ✓ | | | ✓ | | | | |
| System.IdentityModel.* | | | ✓ | | | | | | | |
| Microsoft.Build.* | | | ✓ | | | ✓ | | ✓ | | |
| System.Management.* | | | | | | | | ✓ | ✓ | ✓ |
| System.Threading.* | | ✓ | | | | | | | ✓ | ✓ |
| System.Runtime.Serialization.* | | ✓ | | ✓ | ✓ | | | | | |
| System.Security.AccessControl | | | | ✓ | | | | | | |
| System.Security.Permissions | | ✓ | | ✓ | | | | ✓ | | |
| System.Runtime.CompilerServices | | ✓ | | ✓ | | | | | | |
| System.Linq.* | | | | | | ✓ | | ✓ | ✓ | |
| System.AddIn.* | ✓ | | | ✓ | ✓ | | ✓ | ✓ | | |
| System.Xaml.* | | | ✓ | | ✓ | | | ✓ | | |
| System.Messaging.* | | | ✓ | | | | | | | |
| Microsoft.Win32.* | | ✓ | | | | | | | ✓ | ✓ |
| System.Security.Policy | | | | ✓ | | ✓ | | | | |
| System.Globalization | | ✓ | ✓ | | | ✓ | | | | |
| Microsoft.VisualC.* | ✓ | | ✓ | | ✓ | | ✓ | ✓ | | |
| System.Transactions.* | | | | | ✓ | | | ✓ | ✓ | ✓ |
| System.Security | | ✓ | | ✓ | | | ✓ | ✓ | | |
| System.Collections.Generic | | ✓ | ✓ | | ✓ | | ✓ | ✓ | ✓ | |
| System.Runtime.DurableInstancing | | | | | | | | ✓ | | |
| System.Collections | | ✓ | ✓ | | | | ✓ | | | |
| System.Text | | ✓ | | ✓ | | | | | | |
| System.Deployment.* | | | ✓ | | | | | | ✓ | ✓ |
| System.Runtime.Caching.* | | | | | ✓ | | | | ✓ | ✓ |
| System.ServiceProcess.* | | | ✓ | | | | ✓ | | | |
| System.Resources.* | | | | | | | | | | |
| System.Dynamic | | | ✓ | | ✓ | ✓ | | | | |
| System.Security.Principal | | | | | | | | | | |
| Microsoft.SqlServer.Server | | | | ✓ | | | | | | |
| System.Security.Authentication.* | | | | | ✓ | | | ✓ | | |
| System.Collections.Specialized | | ✓ | ✓ | | | | | | | |
| System.Device.Location | ✓ | | ✓ | | | | ✓ | | ✓ | |
| System.Text.RegularExpressions | | | ✓ | | ✓ | | | | | |
| Accessibility | | | ✓ | | ✓ | | ✓ | | | |

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| System.Collections.Concurrent | | ✓ | | ✓ | ✓ | | | |
| System.Runtime.Versioning | | | ✓ | | ✓ | | | |
| Microsoft.CSharp.* | | | | | ✓ | ✓ | | |
| System.Collections.ObjectModel | | ✓ | | ✓ | ✓ | | ✓ | |
| System.Runtime.ConstrainedExecution | | | ✓ | | ✓ | | | |
| System.Runtime | | | ✓ | | ✓ | | | |
| System.Timers | | ✓ | | | ✓ | ✓ | ✓ | ✓ |
| System.Media | | | | | ✓ | | ✓ | |
| System.Runtime.Hosting | | | | | ✓ | | | |
| System.Runtime.ExceptionServices | ✓ | | | | ✓ | | | |
| System.Numerics | | | | | ✓ | | | |

Table A.8: Classification of .NET namespaces

## Additions to comparison of potential and actual reuse

The notion of 'late-bound type' was only explained very briefly in §4.3.4. Additional details follow.

We start from the most basic form of framework usage: client code references a framework or a namespace thereof. Such a *reference* could relate to various language concepts, e.g., a reference to a class in a constructor call, a reference to a type in the declaration of a method argument, or a reference to a base class in a class declaration.

In terms of OO-based reuse of a framework in client code, usage in the sense of type *specialization* is of special interest. Yet more advanced usage is resembled by *late binding* at the boundary of framework and client code. We are are concerned here with late binding in the sense that a method call of the client code uses a framework type as static receiver type, but the actual runtime receiver type ends up being a type of the client code which necessarily derives from the static receiver type.

For clarity, consider the following client code:

```
public class MyList<T> : List<T> { ... }
public static Program {
    public static void printResult(List<Item> l)
    {
        ...
        Console.WriteLine("Count: {0}", l.Count);
        ...
    }
    public static void Main(string[] args)
    {
        List<Item> r = new MyList<Item>();
        ...
        printResult(r);
        ...
    }
}
```

| Namespace | # Types | % Specializable types | ActiveRecord | CastleCore | MonoRail | Windsor | Json.NET | log4net | MEF | Moq | NAnt | NHibernate | NUnit | Prism | Rhino.Mocks | Spring.NET | xUnit | SharpZipLib | Lucene.Net | Dominator | % Referenced types | % Specializable types (rel.) | % Specialized types (rel.) | % Late-bound types (rel.) |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Framework | | | 3.5 | 4.0 | 3.5 | 4.0 | 4.0 | 2.0 | 4.0 | 4.0 | 2.0 | 3.5 | 3.5 | 4.0 | 3.5 | 2.0 | 2.0 | 2.0 | 2.0 | | | | | |
| System.Web.* | 2327 | 73 | (3, 1, 0) | (0, 0, 0) | (37, 6, 0) | (7, 1, 0) | (0, 0, 0) | (5, 0, 0) | (0, 0, 0) | (0, 0, 0) | (5, 0, 0) | (0, 0, 0) | (0, 0, 0) | (0, 0, 0) | (0, 0, 0) | (146, 31, 4) | (0, 0, 0) | (0, 0, 0) | (0, 0, 0) | (168, 33, 4) | 7 | 44.0 | 45 | 5 |
| System.Windows.* | 1658 | 74 | (0, 0, 0) | (0, 0, 0) | (0, 0, 0) | (0, 0, 0) | (0, 0, 0) | – | (0, 0, 0) | (0, 0, 0) | – | (0, 0, 0) | (0, 0, 0) | (31, 3, 5) | (0, 0, 0) | – | – | – | – | (31, 3, 5) | 2 | 64.5 | 15 | 25 |
| System.ServiceModel.* | 1153 | 68 | (0, 0, 0) | (0, 0, 0) | (0, 0, 0) | (0, 0, 0) | (0, 0, 0) | – | (0, 0, 0) | (0, 0, 0) | – | (3, 1, 0) | (0, 0, 0) | (0, 0, 0) | (0, 0, 0) | – | – | – | – | (3, 1, 0) | 0 | 66.7 | 50 | 0 |
| System.Windows.Forms.* | 963 | 89 | (0, 0, 0) | (0, 0, 0) | (0, 0, 0) | (0, 0, 0) | (0, 0, 0) | (0, 0, 0) | (0, 0, 0) | (0, 0, 0) | (0, 0, 0) | (0, 0, 0) | (128, 11, 7) | (0, 0, 0) | (0, 0, 0) | (0, 0, 0) | (73, 2, 0) | (0, 0, 0) | (0, 0, 0) | (140, 11, 7) | 15 | 52.9 | 15 | 10 |
| System.Data.* | 745 | 46 | (5, 0, 0) | (0, 0, 0) | (2, 0, 0) | (0, 0, 0) | (15, 0, 0) | (9, 0, 0) | (0, 0, 0) | (0, 0, 0) | (0, 0, 0) | (37, 2, 0) | (0, 0, 0) | (0, 0, 0) | (0, 0, 0) | (26, 2, 0) | (8, 0, 0) | (0, 0, 0) | (0, 0, 0) | (59, 2, 0) | 8 | 52.5 | 7 | 0 |
| System.ComponentModel.* | 434 | 77 | (1, 0, 0) | (24, 9, 2) | (4, 1, 0) | (14, 4, 1) | (28, 6, 1) | (0, 0, 0) | (4, 2, 0) | (2, 0, 0) | (13, 1, 1) | (8, 1, 0) | (21, 1, 0) | (25, 5, 0) | (0, 0, 0) | (22, 3, 2) | (9, 0, 0) | (0, 0, 0) | (0, 0, 0) | (88, 27, 5) | 20 | 69.3 | 44 | 8 |
| System.Xml.* | 358 | 87 | (9, 0, 0) | (37, 3, 1) | (5, 0, 0) | (12, 0, 0) | (31, 0, 0) | (15, 0, 0) | (0, 0, 0) | (0, 0, 0) | (57, 1, 0) | (37, 0, 0) | (23, 0, 0) | (0, 0, 0) | (0, 0, 0) | (40, 5, 4) | (13, 0, 0) | (0, 0, 0) | (0, 0, 0) | (111, 8, 5) | 31 | 82.0 | 9 | 6 |
| System.Net.* | 285 | 74 | (0, 0, 0) | (5, 0, 0) | (6, 0, 0) | (0, 0, 0) | (0, 0, 0) | (17, 0, 0) | (0, 0, 0) | (0, 0, 0) | (15, 0, 0) | (0, 0, 0) | (0, 0, 0) | (0, 0, 0) | (0, 0, 0) | (10, 0, 0) | (0, 0, 0) | (0, 0, 0) | (9, 0, 0) | (42, 0, 0) | 15 | 71.4 | 0 | 0 |
| System | 257 | 79 | (51, 7, 3) | (74, 7, 2) | (72, 10, 3) | (75, 6, 4) | (76, 5, 3) | (61, 7, 0) | (64, 5, 2) | (74, 3, 3) | (77, 9, 4) | (94, 10, 3) | (82, 9, 4) | (55, 6, 2) | (46, 4, 3) | (91, 11, 5) | (64, 4, 3) | (42, 4, 0) | (69, 8, 1) | (167, 19, 8) | 65 | 41.9 | 27 | 11 |
| System.Security.Cryptography.* | 250 | 55 | (0, 0, 0) | (0, 0, 0) | (0, 0, 0) | (0, 0, 0) | (0, 0, 0) | (0, 0, 0) | (0, 0, 0) | (0, 0, 0) | (5, 0, 0) | (0, 0, 0) | (0, 0, 0) | (0, 0, 0) | (0, 0, 0) | (0, 0, 0) | (0, 0, 0) | (14, 3, 0) | (3, 0, 0) | (19, 3, 0) | 8 | 68.4 | 23 | 0 |
| System.Runtime.InteropServices.* | 235 | 65 | (0, 0, 0) | (7, 0, 0) | (2, 0, 0) | (2, 0, 0) | (5, 0, 0) | (8, 0, 0) | (3, 0, 0) | (5, 0, 0) | (11, 0, 0) | (3, 0, 0) | (7, 0, 0) | (3, 0, 0) | (5, 0, 0) | (5, 0, 0) | (3, 0, 0) | (4, 0, 0) | (4, 0, 0) | (15, 0, 0) | 6 | 26.7 | 0 | 0 |
| Microsoft.VisualBasic.* | 235 | 69 | (0, 0, 0) | (0, 0, 0) | (0, 0, 0) | (0, 0, 0) | (0, 0, 0) | (0, 0, 0) | (0, 0, 0) | (0, 0, 0) | (1, 0, 0) | (0, 0, 0) | (0, 0, 0) | (0, 0, 0) | (0, 0, 0) | (2, 0, 0) | (0, 0, 0) | (0, 0, 0) | (0, 0, 0) | (3, 0, 0) | 1 | 33.3 | 0 | 0 |
| System.Drawing.* | 191 | 47 | (0, 0, 0) | (0, 0, 0) | (0, 0, 0) | (0, 0, 0) | (0, 0, 0) | (0, 0, 0) | (0, 0, 0) | (0, 0, 0) | (0, 0, 0) | (0, 0, 0) | (28, 0, 0) | (0, 0, 0) | (0, 0, 0) | (2, 0, 0) | (17, 0, 0) | (0, 0, 0) | (0, 0, 0) | (31, 0, 0) | 16 | 6.5 | 0 | 0 |
| System.Runtime.Remoting.* | 190 | 77 | (0, 0, 0) | (3, 0, 0) | (0, 0, 0) | (5, 0, 0) | (0, 0, 0) | (3, 0, 0) | (0, 0, 0) | (0, 0, 0) | (3, 0, 0) | (1, 0, 0) | (21, 1, 1) | (1, 0, 0) | (6, 1, 1) | (7, 0, 0) | (3, 2, 0) | (0, 0, 0) | (0, 0, 0) | (29, 3, 1) | 15 | 79.3 | 13 | 4 |
| System.Configuration.* | 186 | 54 | (5, 1, 0) | (2, 0, 0) | (7, 1, 0) | (1, 1, 0) | (0, 0, 0) | (3, 1, 0) | (0, 0, 0) | (0, 0, 0) | (3, 1, 0) | (4, 1, 0) | (1, 0, 0) | (6, 3, 1) | (0, 0, 0) | (24, 3, 0) | (7, 4, 0) | (0, 0, 0) | (1, 0, 0) | (32, 7, 1) | 17 | 59.4 | 37 | 5 |
| System.Diagnostics.* | 173 | 66 | (4, 0, 0) | (18, 0, 0) | (4, 0, 0) | (8, 0, 0) | (6, 0, 0) | (7, 0, 0) | (13, 0, 0) | (9, 0, 0) | (3, 0, 0) | (10, 0, 0) | (12, 0, 0) | (6, 0, 0) | (4, 0, 0) | (6, 0, 0) | (8, 1, 0) | (0, 0, 0) | (4, 0, 0) | (34, 1, 0) | 20 | 47.1 | 6 | 0 |
| System.IO.* | 123 | 68 | (6, 0, 0) | (14, 0, 0) | (20, 1, 0) | (9, 0, 0) | (8, 0, 0) | (23, 3, 2) | (4, 0, 0) | (1, 0, 0) | (25, 1, 1) | (13, 0, 0) | (25, 2, 3) | (9, 0, 1) | (1, 0, 0) | (22, 0, 0) | (16, 0, 0) | (21, 1, 2) | (22, 5, 2) | (35, 8, 6) | 29 | 68.6 | 33 | 25 |
| System.Reflection.* | 113 | 42 | (11, 0, 0) | (58, 0, 0) | (24, 0, 0) | (26, 0, 0) | (43, 0, 0) | (25, 0, 0) | (45, 0, 0) | (24, 0, 0) | (28, 0, 0) | (41, 0, 0) | (29, 0, 0) | (23, 0, 0) | (30, 0, 0) | (64, 0, 0) | (35, 0, 0) | (12, 0, 0) | (21, 0, 0) | (77, 0, 0) | 68 | 29.9 | 0 | 0 |
| System.EnterpriseServices.* | 111 | 44 | (0, 0, 0) | (0, 0, 0) | (0, 0, 0) | (0, 0, 0) | (0, 0, 0) | (0, 0, 0) | (0, 0, 0) | (0, 0, 0) | (0, 0, 0) | (0, 0, 0) | (0, 0, 0) | (0, 0, 0) | (0, 0, 0) | (19, 0, 0) | (0, 0, 0) | (0, 0, 0) | (0, 0, 0) | (19, 0, 0) | 17 | 10.5 | 0 | 0 |
| System.CodeDom.* | 105 | 98 | (0, 0, 0) | (0, 0, 0) | (0, 0, 0) | (0, 0, 0) | (0, 0, 0) | (0, 0, 0) | (1, 0, 0) | (1, 0, 0) | (34, 0, 0) | (6, 0, 0) | (5, 0, 0) | (1, 0, 0) | (0, 0, 0) | (0, 0, 0) | (1, 0, 0) | (0, 0, 0) | (0, 0, 0) | (35, 0, 0) | 33 | 94.3 | 0 | 0 |
| Microsoft.Build.* | 96 | 92 | (0, 0, 0) | (0, 0, 0) | (0, 0, 0) | (0, 0, 0) | (0, 0, 0) | (0, 0, 0) | (0, 0, 0) | (0, 0, 0) | (20, 1, 0) | (0, 0, 0) | (0, 0, 0) | (0, 0, 0) | (0, 0, 0) | (0, 0, 0) | (4, 0, 0) | (0, 0, 0) | (0, 0, 0) | (23, 1, 0) | 24 | 8.7 | 50 | 0 |
| System.Threading.* | 80 | 76 | (4, 0, 0) | (5, 0, 0) | (6, 0, 0) | (3, 0, 0) | (3, 0, 0) | (11, 0, 0) | (6, 0, 0) | (1, 0, 0) | (3, 0, 0) | (3, 0, 0) | (10, 0, 0) | (5, 0, 0) | (2, 0, 0) | (9, 1, 0) | (8, 0, 0) | (2, 0, 0) | (5, 0, 0) | (21, 1, 0) | 26 | 33.3 | 14 | 0 |
| System.Runtime.Serialization.* | 74 | 49 | (2, 0, 0) | (8, 3, 0) | (2, 0, 0) | (5, 2, 0) | (15, 2, 0) | (5, 1, 0) | (4, 1, 0) | (2, 0, 0) | (5, 1, 0) | (7, 3, 0) | (4, 0, 0) | (2, 0, 0) | (2, 0, 0) | (8, 3, 0) | (2, 0, 0) | (2, 0, 0) | (6, 2, 0) | (23, 6, 0) | 31 | 34.8 | 75 | 0 |
| System.Security.Permissions | 70 | 19 | (0, 0, 0) | (4, 0, 0) | (0, 0, 0) | (2, 0, 0) | (4, 0, 0) | (2, 0, 0) | (2, 0, 0) | (3, 0, 0) | (6, 0, 0) | (4, 0, 0) | (1, 0, 0) | (2, 0, 0) | (2, 0, 0) | (6, 0, 0) | (0, 0, 0) | (0, 0, 0) | (0, 0, 0) | (12, 0, 0) | 17 | 0.0 | 0 | 0 |
| System.Runtime.CompilerServices | 69 | 51 | (6, 0, 0) | (5, 0, 0) | (6, 0, 0) | (7, 0, 0) | (6, 0, 0) | (5, 0, 0) | (7, 0, 0) | (5, 0, 0) | (6, 0, 0) | (7, 0, 0) | (4, 0, 0) | (4, 0, 0) | (10, 0, 0) | (8, 0, 0) | (3, 0, 0) | (4, 0, 0) | (5, 0, 0) | (14, 0, 0) | 20 | 28.6 | 0 | 0 |
| System.Linq.* | 62 | 63 | (2, 0, 0) | (2, 0, 0) | (2, 0, 0) | (12, 0, 0) | (3, 0, 0) | – | (14, 0, 0) | (29, 5, 1) | – | (30, 6, 1) | (0, 0, 0) | (8, 0, 0) | (4, 0, 0) | – | – | – | – | (33, 7, 2) | 53 | 63.6 | 33 | 10 |
| System.Messaging.* | 52 | 90 | (0, 0, 0) | (0, 0, 0) | (0, 0, 0) | (0, 0, 0) | (0, 0, 0) | (0, 0, 0) | (0, 0, 0) | (0, 0, 0) | (0, 0, 0) | (0, 0, 0) | (0, 0, 0) | (0, 0, 0) | (0, 0, 0) | (12, 0, 0) | (0, 0, 0) | (0, 0, 0) | (0, 0, 0) | (12, 0, 0) | 23 | 75.0 | 0 | 0 |
| Microsoft.Win32.* | 49 | 56 | (0, 0, 0) | (0, 0, 0) | (0, 0, 0) | (0, 0, 0) | (0, 0, 0) | (0, 0, 0) | (0, 0, 0) | (0, 0, 0) | (3, 0, 0) | (0, 0, 0) | (2, 0, 0) | (0, 0, 0) | (0, 0, 0) | (2, 0, 0) | (2, 0, 0) | (0, 0, 0) | (0, 0, 0) | (3, 0, 0) | 6 | 33.3 | 0 | 0 |
| System.Security.Policy | 45 | 31 | (0, 0, 0) | (0, 0, 0) | (0, 0, 0) | (0, 0, 0) | (0, 0, 0) | (0, 0, 0) | (0, 0, 0) | (0, 0, 0) | (1, 0, 0) | (0, 0, 0) | (5, 0, 0) | (1, 0, 0) | (0, 0, 0) | (1, 0, 0) | (1, 0, 0) | (0, 0, 0) | (0, 0, 0) | (5, 0, 0) | 11 | 0.0 | 0 | 0 |
| System.Globalization | 39 | 90 | (1, 0, 0) | (1, 0, 0) | (4, 0, 0) | (4, 0, 0) | (4, 0, 0) | (4, 0, 0) | (2, 0, 0) | (1, 0, 0) | (6, 0, 0) | (3, 0, 0) | (2, 0, 0) | (1, 0, 0) | (0, 0, 0) | (8, 0, 0) | (1, 0, 0) | (2, 0, 0) | (6, 0, 0) | (12, 0, 0) | 31 | 50.0 | 0 | 0 |
| System.Transactions.* | 38 | 68 | (0, 0, 0) | (0, 0, 0) | (0, 0, 0) | (0, 0, 0) | (0, 0, 0) | (0, 0, 0) | (0, 0, 0) | (0, 0, 0) | (0, 0, 0) | (12, 1, 0) | (0, 0, 0) | (0, 0, 0) | (0, 0, 0) | (11, 0, 0) | (4, 0, 0) | (0, 0, 0) | (0, 0, 0) | (17, 1, 0) | 45 | 52.9 | 11 | 0 |
| System.Security | 36 | 50 | (0, 0, 0) | (6, 0, 0) | (0, 0, 0) | (3, 0, 0) | (3, 0, 0) | (2, 0, 0) | (9, 0, 0) | (1, 0, 0) | (1, 0, 0) | (3, 0, 0) | (2, 0, 0) | (0, 0, 0) | (0, 0, 0) | (7, 0, 0) | (1, 0, 0) | (1, 0, 0) | (2, 0, 0) | (13, 0, 0) | 36 | 46.2 | 0 | 0 |
| System.Collections.Generic | 31 | 97 | (15, 2, 2) | (18, 9, 1) | (15, 0, 0) | (20, 4, 3) | (14, 6, 5) | (1, 0, 0) | (15, 5, 1) | (11, 3, 0) | (1, 0, 0) | (24, 6, 4) | (12, 1, 1) | (20, 6, 1) | (12, 3, 2) | (8, 3, 0) | (16, 4, 1) | (0, 0, 0) | (17, 4, 1) | (20, 10, 6) | 65 | 90.0 | 56 | 33 |
| System.Text | 25 | 46 | (2, 0, 0) | (2, 0, 0) | (2, 0, 0) | (1, 0, 0) | (2, 0, 0) | (2, 0, 0) | (1, 0, 0) | (1, 0, 0) | (3, 0, 0) | (2, 0, 0) | (2, 0, 0) | (1, 0, 0) | (1, 0, 0) | (3, 0, 0) | (1, 0, 0) | (2, 0, 0) | (2, 0, 0) | (3, 0, 0) | 12 | 66.7 | 0 | 0 |
| System.Collections | 25 | 92 | (10, 5, 1) | (8, 6, 0) | (15, 7, 4) | (11, 7, 0) | (7, 6, 0) | (11, 8, 1) | (7, 5, 2) | (2, 2, 0) | (14, 10, 3) | (15, 8, 3) | (14, 6, 4) | (4, 4, 0) | (10, 2, 2) | (17, 9, 5) | (4, 2, 1) | (5, 3, 0) | (13, 6, 2) | (21, 13, 8) | 84 | 85.7 | 72 | 44 |
| System.ServiceProcess.* | 21 | 92 | (0, 0, 0) | (0, 0, 0) | (0, 0, 0) | (0, 0, 0) | (0, 0, 0) | (0, 0, 0) | (0, 0, 0) | (0, 0, 0) | (2, 0, 0) | (0, 0, 0) | (0, 0, 0) | (0, 0, 0) | (0, 0, 0) | (0, 0, 0) | (0, 0, 0) | (0, 0, 0) | (0, 0, 0) | (2, 0, 0) | 10 | 50.0 | 0 | 0 |
| System.Resources.* | 20 | 74 | (0, 0, 0) | (2, 0, 0) | (2, 0, 0) | (0, 0, 0) | (1, 0, 0) | (0, 0, 0) | (1, 0, 0) | (2, 0, 0) | (1, 0, 0) | (0, 0, 0) | (1, 0, 0) | (2, 0, 0) | (0, 0, 0) | (4, 1, 1) | (1, 0, 0) | (0, 0, 0) | (2, 0, 0) | (5, 1, 1) | 25 | 80.0 | 25 | 25 |
| System.Security.Principal | 18 | 75 | (0, 0, 0) | (0, 0, 0) | (2, 0, 0) | (0, 0, 0) | (0, 0, 0) | (4, 0, 0) | (0, 0, 0) | (0, 0, 0) | (0, 0, 0) | (0, 0, 0) | (1, 0, 0) | (0, 0, 0) | (0, 0, 0) | (0, 0, 0) | (4, 0, 0) | (0, 0, 0) | (0, 0, 0) | (6, 0, 0) | 33 | 100.0 | 0 | 0 |
| System.Collections.Specialized | 15 | 100 | (0, 0, 0) | (2, 1, 0) | (4, 1, 1) | (1, 0, 1) | (0, 0, 0) | (2, 0, 0) | (0, 0, 0) | (0, 0, 0) | (7, 1, 1) | (5, 0, 0) | (5, 0, 0) | (4, 1, 0) | (2, 0, 0) | (8, 1, 1) | (0, 0, 0) | (0, 0, 0) | (2, 0, 0) | (13, 4, 3) | 87 | 84.6 | 36 | 27 |
| System.Text.RegularExpressions | 12 | 100 | (6, 0, 0) | (0, 0, 0) | (7, 0, 0) | (6, 0, 0) | (2, 0, 0) | (4, 0, 0) | (0, 0, 0) | (2, 0, 0) | (9, 0, 0) | (7, 0, 0) | (2, 0, 0) | (0, 0, 0) | (1, 0, 0) | (6, 0, 0) | (5, 0, 0) | (2, 0, 0) | (3, 0, 0) | (9, 0, 0) | 75 | 77.8 | 0 | 0 |
| System.Runtime.Versioning | 8 | 17 | (0, 0, 0) | (1, 0, 0) | (0, 0, 0) | (1, 0, 0) | (1, 0, 0) | (0, 0, 0) | (1, 0, 0) | (1, 0, 0) | (0, 0, 0) | (0, 0, 0) | (0, 0, 0) | (1, 0, 0) | (0, 0, 0) | (0, 0, 0) | (0, 0, 0) | (0, 0, 0) | (0, 0, 0) | (1, 0, 0) | 13 | 0.0 | 0 | 0 |
| System.Collections.ObjectModel | 7 | 100 | (0, 0, 0) | (1, 1, 1) | (1, 1, 1) | (2, 1, 1) | (3, 2, 2) | (0, 0, 0) | (2, 0, 1) | (1, 0, 0) | (0, 0, 0) | (2, 0, 1) | (0, 0, 0) | (2, 1, 1) | (1, 0, 0) | (1, 0, 0) | (1, 0, 0) | (0, 0, 0) | (2, 1, 2) | (4, 2, 2) | 57 | 100.0 | 50 | 50 |
| Microsoft.CSharp.* | 7 | 80 | (0, 0, 0) | (0, 0, 0) | (0, 0, 0) | (0, 0, 0) | (0, 0, 0) | (0, 0, 0) | (0, 0, 0) | (0, 0, 0) | (1, 0, 0) | (1, 0, 0) | (1, 0, 0) | (0, 0, 0) | (0, 0, 0) | (0, 0, 0) | (0, 0, 0) | (0, 0, 0) | (0, 0, 0) | (1, 0, 0) | 14 | 100.0 | 0 | 0 |
| System.Timers | 4 | 100 | (0, 0, 0) | (0, 0, 0) | (0, 0, 0) | (0, 0, 0) | (0, 0, 0) | (0, 0, 0) | (0, 0, 0) | (0, 0, 0) | (0, 0, 0) | (0, 0, 0) | (3, 0, 0) | (0, 0, 0) | (0, 0, 0) | (0, 0, 0) | (2, 0, 0) | (0, 0, 0) | (0, 0, 0) | (3, 0, 0) | 75 | 66.7 | 0 | 0 |
| # Referenced types | | | 143 | 307 | 251 | 237 | 280 | 229 | 205 | 178 | 369 | 382 | 453 | 217 | 139 | 605 | 317 | 113 | 198 | | | | | |
| # Specialized types | | | 16 | 39 | 28 | 26 | 27 | 20 | 18 | 13 | 26 | 39 | 31 | 29 | 10 | 73 | 19 | 11 | 26 | | | | | |
| # Late bound types | | | 6 | 7 | 9 | 10 | 11 | 3 | 6 | 4 | 10 | 12 | 20 | 11 | 8 | 22 | 5 | 2 | 8 | | | | | |
| **75 %** | 235 | 89 | | | | | | | | | | | | | | | | | | | 33 | 78 | 33 | 8 |
| **Median** | 80 | 73 | | | | | | | | | | | | | | | | | | | 20 | 53 | 7 | 0 |
| **25 %** | 36 | 54 | | | | | | | | | | | | | | | | | | | 13 | 33 | 0 | 0 |

**Table A.9.** Usage of .NET in the corpus: (numbers of referencing, specialization, late binding)

The client code specializes the framework class *List* for generic collections resulting in a subclass *MyList*. The client code also defines a method *printResult* that works on the framework type *List*. In the body of that method, *List*'s virtual member *Count* (which is a property, in fact) is invoked. Further, the client code instantiates *MyList* and passes that list to *printResult*. Subject to a dynamic program analysis, it can be determined that late binding is used on *List*. In fact, in this specific example, an *inter-procedural* analysis would be sufficient as opposed to a full-blown dynamic analysis. The term *static receiver type* refers the receiver type essentially as it is declared in the source code or as the declaration is recoverable from the byte code. Hence, the static receiver type in the call *l.Count* is *List<Item>*, but the dynamic receiver type is *MyList<Item>*.

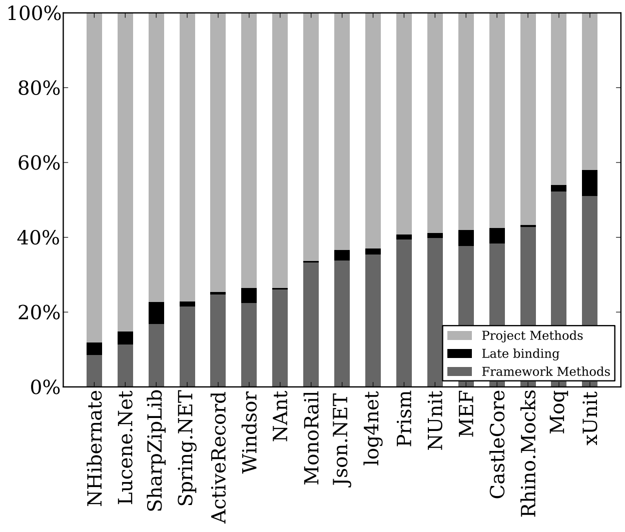


**Figure A.11.** .NET calls w/ and w/o late binding

Actual reuse of .NET platform by projects is shown in Table A.9, providing numbers for the infographics of Table 4.8. Figure A.11 shows how often late binding occurs. Figure A.12 gives detailed overview of the breakdown of referenced types into late-bound, specialized, specializable, and non-specializable. Figure A.13 provides such breakdown per each namespace, for all types, including non-referenced.

Figures A.14 and A.15 provide information about .NET interfaces and classes that were derived/inherited in the corpus. Figure A.16 shows top 30 .NET interfaces that were implemented in the corpus. Table A.10 lists .NET orphan types that were implemented in the corpus.

The derived .NET classes in Figure A.15 can be classified as follows. Again, collections dominate the picture, followed by custom attributes (say, annotations) and

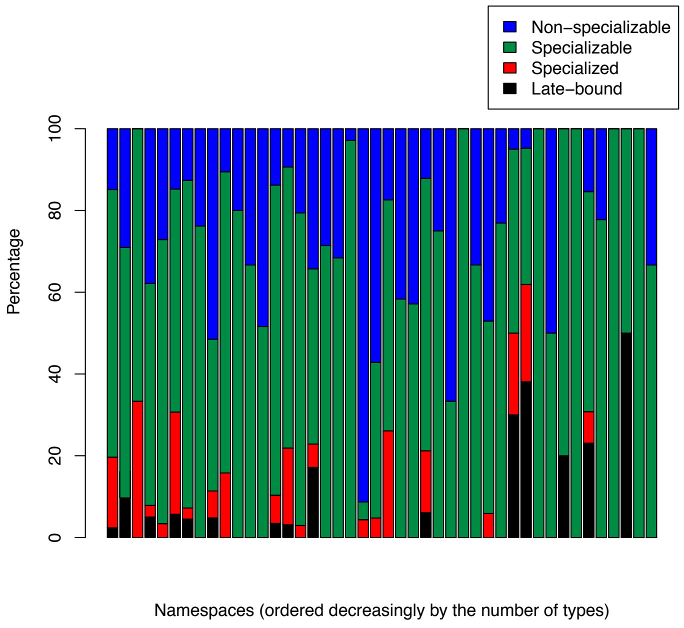


**Figure A.12.** Breakdown of referenced types

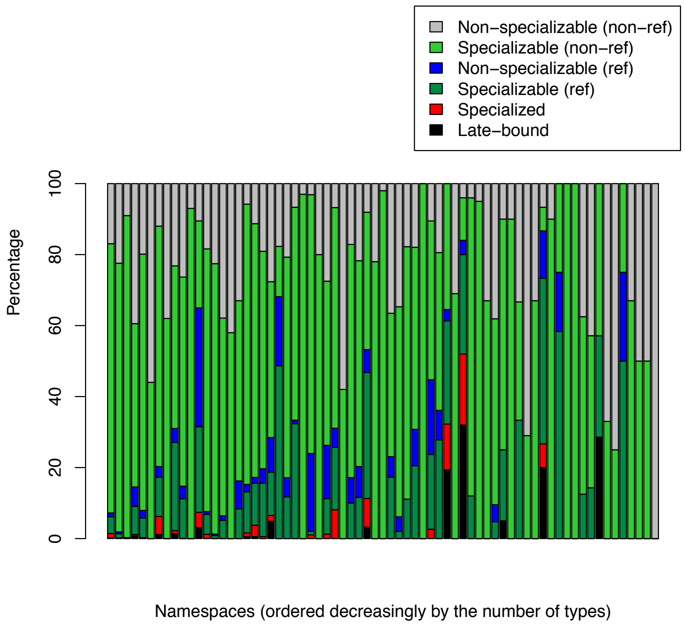
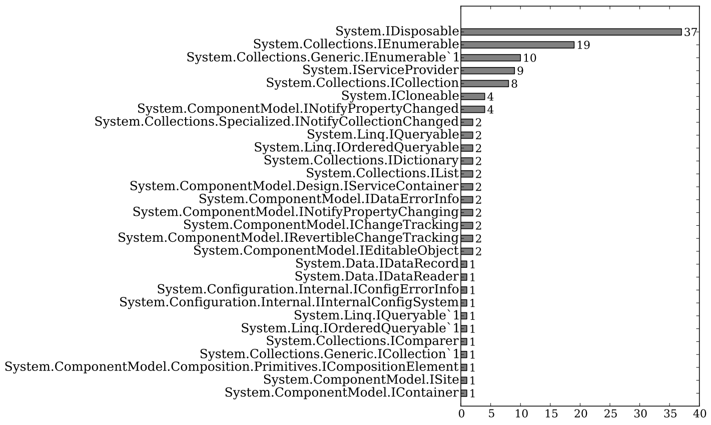
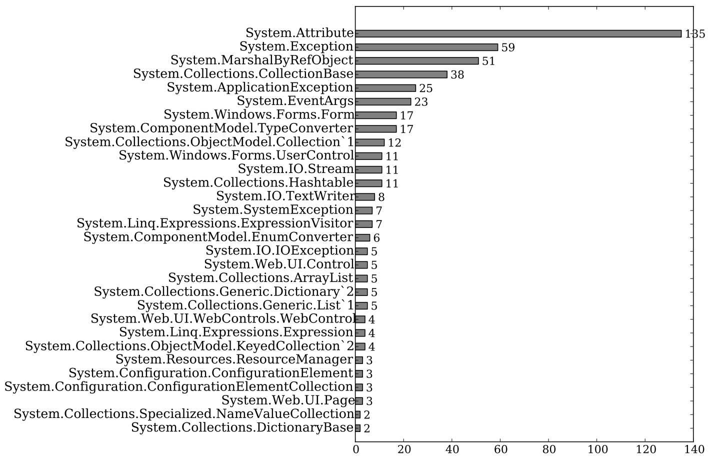


**Figure A.13.** Breakdown of types in terms of the actual reuse metrics



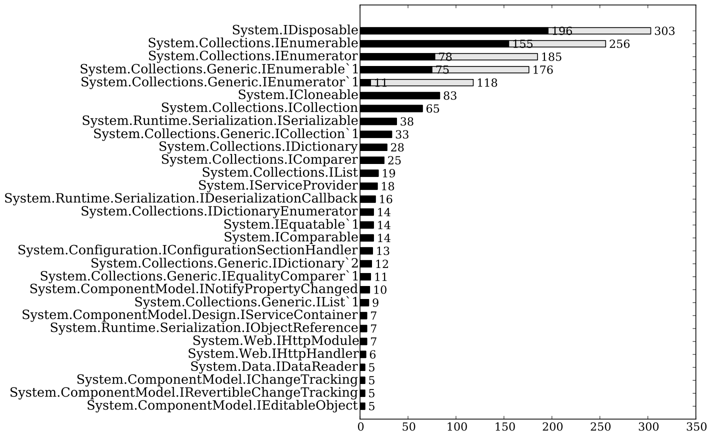
For each interface, the number of sub-interfaces in the corpus are shown.

**Figure A.14.** All .NET interfaces with sub-interfaces in the corpus



For each class, the number of subclasses in the corpus are shown.

**Figure A.15.** Top 30 .NET classes inherited in the corpus



For each interfaces, the number of implementing classes in the corpus are shown. Note: the full bar counts all implementations whereas the black part excludes classes that can be reliably classified as being compiler-generated.

**Figure A.16.** Top 30 .NET interfaces implemented in the corpus

| Namespace | % | I/C | Implemented types | Count |
|---|---|---|---|---|
| System | 28.57 | I | `IServiceProvider` | 18 |
| | | I | `IAsyncResult` | 1 |
| System.Collections | 60.00 | C | `DictionaryBase` | 2 |
| | | C | `.CollectionBase` | 38 |
| | | C | `ReadOnlyCollectionBase` | 1 |
| System.Collections.ObjectModel | 100.00 | C | `KeyedCollection'2` | 4 |
| System.Collections.Specialized | 100.00 | C | `INotifyCollectionChanged` | 5 |
| System.ComponentModel.* | 12.77 | I | `IEditableObject` | 5 |
| | | I | `IRevertibleChangeTracking` | 5 |
| | | I | `INotifyPropertyChanging` | 4 |
| | | I | `IDataErrorInfo` | 4 |
| | | I | `ITypedList` | 1 |
| | | I | `Composition.IPartImportsSatis-fiedNotification` | 1 |
| System.Configuration.* | 20.00 | C | `ApplicationSettingsBase` | 1 |
| System.Runtime.Serialization.* | 50.00 | I | `IDeserializationCallback` | 16 |
| | | C | `SerializationBinder` | 1 |
| | | I | `ISafeSerializationData` | 1 |
| System.Xml.* | 42.86 | C | `Xsl.XsltContext` | 1 |
| | | I | `Xsl.IXsltContextFunction` | 1 |
| | | I | `Xsl.IXsltContextVariable` | 1 |

**Table A.10.** .NET orphan types implemented in the corpus exceptions. Marginally exercised aspects include conversion, remoting, user interfaces, configuration, I/O, and visitors for LINQ.

The implemented .NET interfaces in Figure A.16 can be classified as follows. The list of interfaces is headed by *IDisposable*, which is used for the release of resources; primarily, these are unmanaged resources. 12 of the 30 interfaces deal with collections. Among the top 10, there are additionally interfaces for serialization and cloning. In the rest of the list, yet other aspects are exercised: comparison, services, streaming, and change tracking.

## On correlation

Where appropriate, we calculated Spearman's rank correlation coefficient, a non-parametric measure of statistical dependence between variables. The sign of the coefficient indicates the direction of the association: "+" means that when *A* increases, *B* increases, too; "-" means that when *A* increases, *B* decreases. The value of the coefficient indicates the degree of the correlation: "0" means that there is no tendency for *B* to either increase or decrease when *A* increases; "1" means that *A* and *B* perfectly monotonically related. (Please note that correlation does not imply causation!)

Spearman's rank correlation coefficient is calculated for reuse-related metrics (Table A.11), for classification of namespaces (Table A.12), and for actual reuse metrics (Table A.13 and Table A.14). Stars highlight statistically significant results: three stars mean that p-value is less than 0.001, two stars mean that p-value is less than 0.01, and one star means that p-value is less than 0.05. P-value in its turn is a probability to obtain something like what is observed, assuming that null hypothesis (that there is no correlation) is true. So "statistically significant results" mean that the null hypothesis can be rejected with low risk of making a type I error (i.e., rejecting the true null hypothesis) and that the alternative hypothesis (that correlation is present, with such-and-such Spearman's rho) can be accepted instead.

We neither analyze the results in detail nor interpret them. Let us just mention some of the observed (non-trivial) correlations. For instance, there is a positive correlation between the number of types in a namespace and the MAX size class tree within the namespace (Table A.11), meaning that the more types a namespace has, the bigger the maximal inheritance tree of namespace's classes. According to Table A.13, there is a positive correlation such that the *number* of late-bound types increases with the number specialized types, which in turn increases with the number specializable, referenced types, which in turn increases with the number referenced types.

| | types | methods | rC | rI | maxCT | maxIT | refdN | refgN | specdN | specgN | pC | pI | pVT | pDT | pGT | pCA | pIA | pVTA | pDA | pStC | pAC | pCrC | pStM | pAM | pCrM | pSldC | pSpC | pSpT | pSldM | pSpM | pOC | pOI | pOT | pLOC |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| types | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| methods | 0.95*** | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| rC | 0.93*** | 0.87*** | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| rI | 0.76*** | 0.74*** | 0.64*** | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| maxCT | 0.77*** | 0.77*** | 0.74*** | 0.50*** | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| maxIT | 0.67*** | 0.67*** | 0.49*** | 0.86*** | 0.41*** | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| refdN | 0.86*** | 0.81*** | 0.86*** | 0.65*** | 0.73*** | 0.60*** | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| refgN | 0.25* | 0.27* | 0.20 | 0.11 | 0.19 | 0.17 | 0.18 | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| specdN | 0.80*** | 0.77*** | 0.80*** | 0.62*** | 0.70*** | 0.54*** | 0.85*** | 0.15 | | | | | | | | | | | | | | | | | | | | | | | | | | |
| specgN | 0.49*** | 0.49*** | 0.38** | 0.51*** | 0.30* | 0.50*** | 0.39*** | 0.68*** | 0.40*** | | | | | | | | | | | | | | | | | | | | | | | | | |
| pC | -0.06 | -0.06 | 0.04 | -0.22 | 0.24* | -0.20 | 0.07 | -0.07 | -0.04 | -0.20 | | | | | | | | | | | | | | | | | | | | | | | | |
| pI | 0.29* | 0.28* | 0.15 | 0.76*** | 0.09 | 0.70*** | 0.19 | 0.02 | 0.22 | 0.36** | -0.38** | | | | | | | | | | | | | | | | | | | | | | | |
| pVT | 0.09 | 0.05 | 0.11 | -0.11 | -0.07 | -0.14 | 0.02 | 0.05 | 0.03 | 0.03 | -0.63*** | -0.19 | | | | | | | | | | | | | | | | | | | | | | |
| pDT | 0.38** | 0.37** | 0.47*** | 0.20 | 0.35** | 0.13 | 0.37** | 0.11 | 0.45*** | 0.20 | -0.18 | -0.06 | 0.02 | | | | | | | | | | | | | | | | | | | | | |
| pGT | 0.14 | 0.19 | 0.07 | 0.15 | 0.12 | 0.23 | 0.01 | -0.07 | 0.09 | 0.04 | 0.03 | 0.20 | -0.20 | -0.03 | | | | | | | | | | | | | | | | | | | | |
| pCA | 0.29* | 0.17 | 0.29* | 0.36** | 0.33** | 0.34** | 0.36** | -0.11 | 0.30* | 0.11 | 0.25* | 0.33** | -0.36** | 0.02 | -0.03 | | | | | | | | | | | | | | | | | | | |
| pIA | 0.24* | 0.24* | 0.18 | 0.40*** | 0.07 | 0.41*** | 0.17 | -0.01 | 0.24* | 0.21 | -0.37** | 0.40*** | -0.03 | 0.30* | 0.19 | 0.16 | | | | | | | | | | | | | | | | | | |
| pVTA | -0.11 | -0.01 | -0.13 | -0.29* | -0.22 | -0.23 | -0.17 | 0.33** | -0.12 | 0.02 | -0.23 | -0.27* | 0.44*** | -0.07 | -0.04 | -0.70*** | -0.23 | | | | | | | | | | | | | | | | | |
| pDA | 0.28* | 0.30* | 0.38** | 0.22 | 0.29* | 0.16 | 0.30* | -0.02 | 0.41*** | 0.14 | 0.11 | 0.04 | -0.26* | 0.59*** | 0.38** | 0.04 | 0.27* | -0.21 | | | | | | | | | | | | | | | | |
| pStC | 0.24* | 0.19 | 0.30* | 0.26* | 0.22 | 0.26* | 0.39*** | 0.29* | 0.18 | 0.14 | 0.07 | 0.20 | -0.09 | 0.07 | 0.13 | 0.17 | 0.18 | -0.08 | 0.15 | | | | | | | | | | | | | | | |
| pAC | 0.28* | 0.29* | 0.18 | 0.11 | 0.35** | 0.20 | 0.17 | 0.36** | 0.23 | 0.48*** | 0.18 | 0.00 | -0.15 | 0.06 | 0.13 | 0.12 | -0.01 | -0.06 | 0.07 | -0.06 | | | | | | | | | | | | | | |
| pCrC | -0.02 | -0.01 | 0.01 | 0.10 | -0.10 | -0.02 | -0.03 | -0.31** | 0.03 | -0.23 | -0.14 | 0.13 | 0.03 | 0.07 | -0.17 | 0.00 | 0.03 | -0.06 | 0.01 | -0.35** | -0.70*** | | | | | | | | | | | | | |
| pStM | 0.12 | 0.11 | 0.17 | 0.05 | 0.13 | 0.05 | 0.19 | 0.31** | 0.01 | -0.03 | -0.09 | -0.03 | 0.22 | -0.01 | 0.01 | -0.08 | 0.09 | 0.25* | -0.03 | 0.59*** | -0.12 | -0.30* | | | | | | | | | | | | |
| pAM | 0.34** | 0.26* | 0.20 | 0.63*** | 0.15 | 0.51*** | 0.19 | 0.11 | 0.25* | 0.39** | -0.27* | 0.80*** | -0.18 | -0.07 | 0.11 | 0.38** | 0.21 | -0.21 | -0.03 | 0.10 | 0.21 | -0.03 | -0.10 | | | | | | | | | | | |
| pCrM | -0.07 | -0.05 | -0.04 | -0.27* | 0.01 | -0.18 | -0.04 | -0.18 | 0.06 | -0.07 | 0.37** | -0.39*** | -0.16 | 0.06 | -0.10 | 0.03 | -0.21 | -0.14 | 0.05 | -0.50*** | 0.16 | 0.23 | -0.75*** | -0.39*** | | | | | | | | | | |
| pSldC | 0.18 | 0.07 | 0.17 | 0.17 | 0.07 | 0.13 | 0.13 | 0.12 | -0.05 | 0.10 | 0.06 | 0.04 | 0.18 | -0.25* | -0.17 | 0.03 | -0.14 | -0.04 | -0.28* | 0.14 | 0.03 | -0.08 | 0.21 | 0.03 | -0.09 | | | | | | | | | |
| pSpC | -0.10 | -0.03 | -0.09 | -0.11 | -0.01 | -0.07 | -0.05 | -0.07 | 0.13 | -0.03 | 0.02 | 0.02 | -0.26* | 0.28* | 0.20 | 0.04 | 0.15 | -0.04 | 0.34** | -0.09 | 0.03 | 0.16 | -0.29* | 0.04 | 0.17 | -0.92*** | | | | | | | | |
| pSpT | -0.07 | -0.02 | -0.08 | -0.01 | -0.05 | 0.02 | -0.04 | -0.07 | 0.13 | 0.00 | -0.08 | 0.18 | -0.25* | 0.24* | 0.23 | 0.07 | 0.25* | -0.06 | 0.32** | -0.02 | -0.02 | 0.18 | -0.24* | 0.17 | 0.06 | -0.89*** | 0.96*** | | | | | | | |
| pSldM | 0.15 | 0.15 | 0.04 | 0.29* | 0.11 | 0.48*** | 0.16 | 0.29* | 0.20 | 0.37** | 0.06 | 0.37** | -0.28* | 0.03 | 0.25* | 0.27* | 0.05 | -0.05 | 0.10 | 0.05 | 0.24 | -0.12 | 0.00 | 0.29* | 0.00 | -0.08 | 0.11 | 0.14 | | | | | | |
| pSpM | 0.23 | 0.20 | 0.14 | 0.38** | 0.11 | 0.29* | 0.09 | 0.16 | 0.15 | 0.32** | -0.22 | 0.40*** | 0.00 | -0.02 | -0.31* | 0.15 | 0.06 | -0.03 | -0.26* | -0.05 | 0.26* | -0.03 | -0.25* | 0.60*** | -0.08 | 0.12 | -0.04 | 0.03 | -0.07 | | | | | |
| pOC | 0.31** | 0.31** | 0.29* | 0.17 | 0.37** | 0.16 | 0.29* | 0.23 | 0.41*** | 0.32** | 0.25* | -0.01 | -0.30* | 0.20 | 0.22 | 0.17 | 0.12 | -0.10 | 0.38** | 0.02 | 0.70*** | -0.49*** | -0.05 | 0.20 | 0.05 | -0.11 | 0.15 | 0.11 | 0.19 | 0.03 | | | | |
| pOI | 0.48*** | 0.43*** | 0.49*** | 0.65*** | 0.41*** | 0.40*** | 0.39*** | -0.11 | 0.46*** | 0.17 | -0.13 | 0.52*** | -0.05 | 0.15 | 0.05 | 0.26* | 0.19 | -0.22 | 0.18 | 0.20 | -0.01 | 0.04 | 0.08 | 0.52*** | -0.35** | 0.21 | -0.17 | -0.09 | -0.09 | 0.43*** | 0.12 | | | |
| pOT | 0.27* | 0.24* | 0.23 | 0.45*** | 0.21 | 0.26* | 0.17 | -0.01 | 0.36** | 0.20 | -0.10 | 0.48*** | -0.21 | 0.11 | 0.22 | 0.17 | 0.27* | -0.13 | 0.24* | 0.06 | 0.29* | -0.16 | -0.02 | 0.57*** | -0.30* | -0.02 | 0.08 | 0.16 | 0.00 | 0.32** | 0.59*** | 0.67*** | | |
| pLOC | 0.19 | 0.18 | 0.11 | 0.13 | 0.16 | 0.13 | 0.10 | 0.35** | 0.21 | 0.51*** | 0.09 | 0.05 | -0.19 | 0.01 | 0.16 | 0.09 | 0.14 | -0.10 | 0.16 | 0.04 | 0.81*** | -0.61*** | -0.09 | 0.24* | 0.07 | 0.00 | 0.05 | 0.04 | 0.15 | 0.15 | 0.80*** | 0.00 | 0.46*** | |
| pLOI | 0.44*** | 0.41*** | 0.43*** | 0.66*** | 0.28* | 0.43*** | 0.30* | 0.07 | 0.40*** | 0.34** | -0.29* | 0.62*** | -0.01 | 0.18 | -0.01 | 0.21 | 0.23 | -0.14 | 0.12 | 0.17 | 0.04 | 0.04 | 0.06 | 0.59*** | -0.36** | 0.19 | -0.14 | -0.05 | -0.01 | 0.51*** | 0.07 | 0.91*** | 0.62*** | 0.07 |

**Table A.11.** Spearman's rank correlation coefficient for reuse-related metrics

| | isApplication | isCore | isOpen | isClosed | isIncomplete | isBranched | isFlat | isInterfaceIntensive | isDelegateIntensive |
|---|---|---|---|---|---|---|---|---|---|
| isApplication | | | | | | | | | |
| isCore | -0.15 | | | | | | | | |
| isOpen | 0.11 | -0.09 | | | | | | | |
| isClosed | -0.02 | 0.13 | -0.41*** | | | | | | |
| isIncomplete | 0.12 | -0.07 | 0.18 | -0.11 | | | | | |
| isBranched | -0.16 | -0.09 | -0.06 | -0.06 | -0.04 | | | | |
| isFlat | 0.39*** | -0.16 | 0.23 | 0.01 | 0.04 | -0.41*** | | | |
| isInterfaceIntensive | 0.12 | 0.08 | -0.04 | -0.11 | 0.27* | -0.11 | 0.04 | | |
| isDelegateIntensive | -0.03 | -0.11 | -0.08 | -0.35** | 0.09 | 0.13 | -0.08 | 0.09 | |
| isEventBased | -0.11 | -0.01 | -0.12 | -0.29* | -0.03 | 0.04 | -0.21 | 0.06 | 0.69*** |

**Table A.12.** Spearman’s rank correlation coefficient for categories

| | types | totalRefT | spbTAbs | spbTRel | totalSpdT |
|---|---|---|---|---|---|
| types | | | | | |
| totalRefT | 0.70*** | | | | |
| spbTAbs | 0.99*** | 0.70*** | | | |
| spbTRel | 0.66*** | 0.95*** | 0.67*** | | |
| totalSpdT | 0.40** | 0.62*** | 0.43** | 0.62*** | |
| totalLbT | 0.22 | 0.49*** | 0.24 | 0.54*** | 0.84*** |

**Table A.13.** Spearman's rank correlation coefficient for actual reuse metrics (numbers)

| | pRefT | pSpbTAbs | pSpbTRel | pSpdTAbs | pSpdTRel | pLBTAbs |
|---|---|---|---|---|---|---|
| pRefT | | | | | | |
| pSpbTAbs | 0.13 | | | | | |
| pSpbTRel | 0.01 | 0.61*** | | | | |
| pSpdTAbs | 0.39** | 0.38* | 0.15 | | | |
| pSpdTRel | 0.26 | 0.46** | 0.17 | 0.94*** | | |
| pLBTAbs | 0.36* | 0.34* | 0.24 | 0.83*** | 0.75*** | |
| pLBTRel | 0.32* | 0.34* | 0.22 | 0.81*** | 0.75*** | 0.99*** |

**Table A.14.** Spearman's rank correlation coefficient for actual reuse metrics (percentage)