Contracts in Practice*

H.-Christian Estler, Carlo A. Furia, Martin Nordio,
Marco Piccioni, and Bertrand Meyer

Chair of Software Engineering, Department of Computer Science, ETH Zurich, Switzerland
firstname.lastname@inf.ethz.ch

Abstract. Contracts are a form of lightweight formal specification embedded in the program text. Being executable parts of the code, they encourage programmers to devote proper attention to specifications, and help maintain consistency between specification and implementation as the program evolves. The present study investigates how contracts are used in the practice of software development. Based on an extensive empirical analysis of 21 contract-equipped Eiffel, C#, and Java projects totaling more than 260 million lines of code over 7700 revisions, it explores, among other questions: 1) which kinds of contract elements (preconditions, postconditions, class invariants) are used more often; 2) how contracts evolve over time; 3) the relationship between implementation changes and contract changes; and 4) the role of inheritance in the process. It has found, among other results, that: the percentage of program elements that include contracts is above 33% for most projects and tends to be stable over time; there is no strong preference for a certain type of contract element; contracts are quite stable compared to implementations; and inheritance does not significantly affect qualitative trends of contract usage.

1 Introduction

Using specifications as an integral part of the software development process has long been advocated by formal methods pioneers and buffs. While today few people question the value brought by formal specifications, the software projects that systematically deploy them are still a small minority. What can we learn from these adopters about the practical usage of specifications to support software development?

In this paper, we answer this question by looking into *contracts*, a kind of lightweight formal specification in the form of executable assertions (preconditions, postconditions, and class invariants). In the practice of software development, contracts support a range of activities such as runtime checking, automated testing, and static verification, and provide rigorous and unambiguous API documentation. They bring some of the advantages of "heavyweight" formal methods while remaining amenable to programmers without strong mathematical skills: whoever can write Boolean expressions can also write contracts. Therefore, learning how contracts are used in the projects that use them can shed light on how formal methods can make their way into the practice of software development.

* Work supported by Gebert-Ruf Stiftung, by ERC grant CME # 291389, and by SNF grant ASII # 200021-134976.

C. Jones, P. Pihlajasaari, and J. Sun (Eds.): FM 2014, LNCS 8442, pp. 230–246, 2014.

The empirical study of this paper analyzes 21 projects written in Eiffel, C#, and Java, three major object-oriented languages supporting contracts, with the goal of studying how formal specifications are written, changed, and maintained as part of general software development. Eiffel has always supported contracts natively; the Java Modeling Language (JML [16]) extends Java with contracts written as comments; and C# has recently added support with the Code Contracts framework [8]. Overall, our study analyzed more than 260 million lines of code and specification distributed over 7700 revisions. To our knowledge, this is the first extensive study of the practical evolving usage of simple specifications such as contracts over project lifetimes.

The study's specific **questions** target various aspects of how contracts are used in practice: Is the usage of contracts quantitatively significant and uniform across the various selected projects? How does it evolve over time? How does it change with the overall project? What kinds of contracts are used more often? What happens to contracts when implementations change? What is the role of inheritance?

The main **findings** of the study, described in Section 3, include:

- The projects in our study make a *significant usage* of contracts: the percentages of routines and classes with specification is above 33% in the majority of projects.
- The usage of specifications tends to be *stable over time*, except for the occasional turbulent phases where major refactorings are performed. This suggests that contracts evolve following design changes.
- There is *no strong preference* for certain *kinds* of specification elements (preconditions, postconditions, class invariants); but preconditions, when they are used, tend to be larger (have more clauses) than postconditions. This indicates that different specification elements are used for different purposes.
- Specifications are quite *stable* compared to implementations: a routine's body may change often, but its contracts will change infrequently. This makes a good case for a fundamental software engineering principle: stable interfaces over changing implementations [21].
- *Inheritance* does not significantly affect the qualitative findings about specification usage: measures including and excluding inherited contracts tend to correlate. This suggests that the abstraction levels provided by inheritance and by contracts are largely complementary.

As a supplemental contribution, we make all data collected for the study available online as an SQL database image [3]. This provides a treasure trove of data about practically all software projects of significant size publicly available that use contracts.

Positioning: What this Study is *Not*. The term "specification" has a broad meaning. To avoid misunderstandings, let us mention other practices that might be interesting to investigate, but which are *not* our target in this paper. We do not consider formal specifications in forms other than executable contracts. We do not look for formal specifications in *generic* software projects: it is well-known [22] that the overwhelming majority of software does not come with formal specifications (or any specifications). Instead, we pick our projects among the minority of those actually using contracts, to study how the few adopters use formal specifications in practice. We do not study *applications* of contracts; but our analysis may serve as a basis to follow-up studies targeting applications. We do not compare different methodologies to design and write contracts; we just observe the results of programming practices.

Extended Version. For lack of space, we can only present the most important facts; an extended version [7] provides more details on both the analysis and the results.

2 Study Setup

Our study analyzes contract specifications in Eiffel, C#, and Java, covering a wide range of projects of different sizes and life spans developed by professional programmers and researchers. We use the terms "contract" and "specification" as synonyms.

Data Selection. We selected 21 open-source projects that use contracts and are available in public repositories. Save for requiring a minimal amount of revisions (at least 30) and contracts (at least 5% of elements in the latest revisions), we included all open-source projects written in Eiffel, C# with CodeContracts, or Java with JML we could find when we performed this research. Table 1 lists the projects and, for each of them, the total number of REVisions, the life span (AGE, in weeks), the size in lines of code (LOC) at the latest revision, the number of DEVelopers involved (i.e., the number of committers to the repository), and a short description.

Table 1. List of projects used in the study. "AGE" is in weeks, "#LOC" is lines of code.

#	PROJECT	LANG.	# REV.	AGE	# LOC	# DEV.	DESCRIPTION
1	AutoTest	Eiffel	306	195	65'625	13	Contract-based random testing tool
2	EiffelBase	Eiffel	1342	1006	61'922	45	General-purpose data structures library
3	EiffelProgramAnalysis	Eiffel	208	114	40'750	8	Utility library for analyzing Eiffel programs
4	GoboKernel	Eiffel	671	747	53'316	8	Library for compiler interoperability
5	GoboStructure	Eiffel	282	716	21'941	6	Portable data structure library
6	GoboTime	Eiffel	120	524	10'840	6	Date and time library
7	GoboUtility	Eiffel	215	716	6'131	7	Library to support design patterns
8	GoboXML	Eiffel	922	285	163'552	6	XML Library supporting XSL and XPath
9	Boogie	C#	766	108	88'284	29	Program verification system
10	CCI	C#	100	171	20'602	3	Library to support compilers construction
11	Dafny	C#	326	106	29'700	19	Program verifier
12	LabsFramework	C#	49	30	14'540	1	Library to manage experiments in .NET
13	Quickgraph	C#	380	100	40'820	4	Generic graph data structure library
14	Rxx	C#	148	68	55'932	2	Library of unofficial reactive LINQ extensions
15	Shweet	C#	59	7	2352	2	Application for messaging in Twitter style
16	DirectVCGen	Java	376	119	13'294	6	Direct Verification Condition Generator
17	ESCJava	Java	879	366	73'760	27	An Extended Static Checker for Java (version 2)
18	JavaFE	Java	395	389	35'013	18	Front-end parser for Java byte and source code
19	Logging	Java	29	106	5'963	3	A logging framework
20	RCC	Java	30	350	10'872	7	Race Condition Checker for Java
21	Umbra	Java	153	169	15'538	8	Editor for Java bytecode and BML specifications
	Total		7'756	6'392	830'747	228	

Measures. The raw measures produced by include: the number of classes, the number of classes with invariants, the average number of invariant clauses per class, and the number of classes modified compared to the previous revision; the number of routines (public and private), the number of routines with non-empty precondition, with non-empty postcondition, and with non-empty specification (that is, precondition, postcondition, or both), the average number of pre- and postcondition clauses per routine, and the number of routines with modified body compared to the previous revision.

Measuring precisely the *strength* of a specification (which refers to how constraining it is) is hardly possible as it requires detailed knowledge of the semantics of classes and establishing undecidable properties in general. In our study, we *count* the number of specification clauses (elements *and*ed, normally on different lines) as a proxy for specification strength. The number of clauses is a measure of *size* that is interesting in its own right. If some clauses are changed,[1] just counting the clauses may measure strength incorrectly. We have evidence, however, that the error introduced by measuring strengthening in this way is small. We manually inspected 277 changes randomly chosen, and found 11 misclassifications (e.g., strengthening reported as weakening). Following [17, Eq. 5], this implies that, with 95% probability, the errors introduced by our estimate (measuring clauses for strength) involve no more than 7% of the changes.

3 How Contracts Are Used

Our study targets the following main *questions*, addressed in the following subsections.

Q1. Do projects make a significant *usage* of contracts, and how does usage evolve over time?
Q2. How does the usage of contracts change with projects growing or shrinking in *size*?
Q3. What *kinds* of contract elements are used more often?
Q4. What is the typical *size* and *strength* of contracts, and how does it change over time?
Q5. Do *implementations* change more often than their *contracts*?
Q6. What is the role of *inheritance* in the way contracts change over time?

Table 2 shows the essential quantitative data we discuss for each project; Table 3 shows sample plots of the data for four projects. In the rest of the section, we illustrate and summarize the data in Table 2 and the plots in Table 3 as well as much more data and plots that, for lack of space, are available elsewhere [3,7].

3.1 Writing Contracts

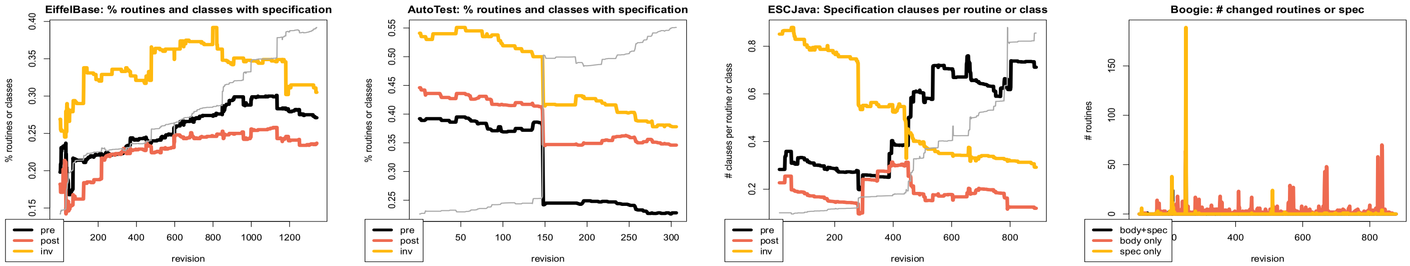
In the majority of projects in our study, developers devoted a considerable part of their programming effort to writing specifications for their code. While we specifically target projects with *some* specification (and ignore the majority of software that doesn't use contracts), we observe that most of the projects achieve *significant* percentages of routines or classes with specification. As shown in column % ROUTINES SPEC of Table 2, in 7 of the 21 analyzed projects, on average 50% or more of the public routines have some specification (pre- or postcondition); in 14 projects, 35% or more of the routines have specification; and only 3 projects have small percentages of specified routines (15% or less). Usage of class invariants (column % CLASSES INV in Table 2) is more varied but still consistent: in 9 projects, 33% or more of the classes have an invariant; in 10 projects, 12% or less of the classes have an invariant. The standard deviation of these percentages is small for 11 of the 21 projects, compared to the average value over

[1] We consider all concrete syntactic changes, that is all textual changes.

Table 2. Specification overall statistics with non-flat classes. For each project, we report the number of classes and of public routines (# CLASSES, # ROUTINES); the percentage (1 is 100%) of classes with non-empty invariant (% CLASSES INV); of routines with non-empty specification (% ROUTINES SPEC) and more specifically with non-empty precondition (PRE) and postcondition (POST); the mean number of clauses of routine preconditions (AVG ROUTINES PRE) and of postconditions (POST). For each measure, the table reports minimum (**m**), median (μ), maximum (**M**), and standard deviation (σ) across all revisions.

	# CLASSES				% CLASSES INV				# ROUTINES				% ROUTINES SPEC				% ROUTINES PRE				% ROUTINES POST				AVG ROUTINES PRE				AVG ROUTINES POST			
Project	**m**	μ	**M**	σ	**m**	μ	**M**	σ	**m**	μ	**M**	σ	**m**	μ	**M**	σ	**m**	μ	**M**	σ	**m**	μ	**M**	σ	**m**	μ	**M**	σ	**m**	μ	**M**	σ
AutoTest	98	220	254	66	0.38	0.43	0.55	0.06	352	1053	1234	372	0.47	0.49	0.61	0.06	0.23	0.25	0.4	0.07	0.34	0.36	0.45	0.04	1.73	1.76	1.85	0.03	1.19	1.22	1.28	0.03
EiffelBase	93	184	256	36	0.24	0.34	0.39	0.03	545	1984	3323	696	0.26	0.4	0.44	0.04	0.17	0.27	0.3	0.03	0.14	0.24	0.26	0.03	1.43	1.6	1.7	0.05	1.2	1.46	1.51	0.06
EiffelProgramAnalysis	0	179	221	30	0	0.04	0.05	0	0	828	1127	199	0	0.25	0.27	0.02	0	0.14	0.16	0.02	0	0.15	0.16	0.01	0	1.23	1.25	0.09	0	1.13	1.17	0.08
GoboKernel	0	72	157	38	0	0.11	0.13	0.04	0	168	702	155	0	0.6	1	0.17	0	0.3	0.4	0.09	0	0.51	1	0.19	0	2.1	2.91	0.59	0	1.32	1.86	0.25
GoboStructure	42	75	109	17	0.19	0.33	0.39	0.06	122	372	483	88	0.18	0.29	0.41	0.07	0.07	0.19	0.28	0.06	0.16	0.23	0.32	0.05	1.45	1.82	1.93	0.13	1.17	1.44	1.49	0.1
GoboTime	0	22	47	10	0	0.12	0.28	0.09	0	176	333	53	0	0.63	0.66	0.06	0	0.28	0.33	0.03	0	0.58	0.6	0.06	0	1.62	1.7	0.15	0	2.28	2.53	0.25
GoboUtility	3	25	43	10	0	0.22	0.5	0.08	1	90	185	55	0	0.9	0.98	0.14	0	0.58	0.83	0.12	0	0.58	0.67	0.11	0	1.8	2.07	0.24	0	1.29	1.52	0.25
GoboXML	0	176	859	252	0	0.38	0.48	0.07	0	883	5465	1603	0	0.35	0.44	0.05	0	0.23	0.35	0.03	0	0.23	0.33	0.06	0	1.43	1.55	0.14	0	1.2	1.36	0.07
Boogie	9	606	647	181	0.24	0.34	0.58	0.06	80	3542	3748	1055	0.49	0.52	0.81	0.09	0.28	0.3	0.74	0.13	0.08	0.32	0.38	0.04	1.6	1.73	1.76	0.03	1	1.02	1.02	0.01
CCI	45	60	108	15	0.01	0.04	0.06	0.01	160	210	302	50	0	0.03	0.05	0.01	0	0.03	0.04	0.01	0	0	0.01	0	1	1.33	1.6	0.22	0	0	1	0.49
Dafny	11	148	184	25	0.04	0.47	0.52	0.06	25	375	551	85	0.16	0.64	0.74	0.07	0.16	0.57	0.64	0.06	0	0.18	0.22	0.03	1	2.29	2.36	0.18	0	1.04	1.05	0.14
Labs	47	58	75	8	0.35	0.38	0.42	0.02	351	413	518	29	0.38	0.47	0.5	0.03	0.28	0.38	0.42	0.03	0.1	0.13	0.21	0.03	1.34	1.37	1.58	0.08	1.13	1.17	1.28	0.05
Quickgraph	228	260	336	27	0	0.02	0.04	0.01	1074	1262	1862	179	0	0.16	0.22	0.07	0	0.15	0.21	0.07	0	0.01	0.02	0.01	0	1.71	2.1	0.71	0	1.18	1.36	0.46
Rxx	0	145	189	53	0	0.42	0.44	0.08	0	1358	1792	494	0	0.7	0.97	0.11	0	0.6	0.93	0.13	0	0.62	0.81	0.08	0	2.1	2.24	0.18	0	1.03	1.12	0.1
Shweet	0	28	36	13	0	0	0	0	0	57	85	33	0	0.1	0.4	0.07	0	0.1	0.4	0.07	0	0.01	0.07	0.02	0	1.6	2	0.77	0	1	1	0.49
DirectVCGen	13	55	82	17	0	0	0.03	0	74	440	582	115	0.06	0.15	0.37	0.04	0.06	0.15	0.37	0.04	0.02	0.1	0.35	0.05	1	1	1.33	0.05	1	1	1	0
ESCJava	66	161	308	80	0.11	0.17	0.26	0.05	233	585	3079	853	0.16	0.36	0.74	0.21	0.14	0.27	0.69	0.2	0.06	0.12	0.2	0.03	1.07	1.27	1.66	0.21	1.21	1.52	1.88	0.12
JavaFE	107	124	641	29	0.12	0.47	0.62	0.04	499	589	1081	125	0.34	0.43	0.8	0.15	0.26	0.34	0.74	0.14	0.13	0.18	0.31	0.04	1.2	1.54	1.61	0.12	1.26	1.48	1.82	0.09
Logging	20	22	23	1	0.04	0.09	0.09	0.01	154	171	173	6	0.32	0.49	0.54	0.04	0.14	0.33	0.35	0.04	0.21	0.28	0.33	0.02	1.39	1.43	1.5	0.04	1.58	1.75	2	0.08
RCC	48	142	144	42	0.08	0.1	0.11	0.01	359	441	447	35	0.06	0.56	0.59	0.24	0.03	0.07	0.1	0.02	0.04	0.52	0.54	0.23	1.21	1.28	1.36	0.04	1	1.04	1.05	0.02
Umbra	23	41	77	16	0	0.06	0.1	0.03	36	122	332	78	0	0.02	0.05	0.02	0	0.01	0.03	0.01	0	0.02	0.04	0.01	0	1	1	0.49	0	1	1	0.47

Table 3. Selected plots for projects EiffelBase, AutoTest, ESCJava, and Boogie. Each graph from left to right represents the evolution over successive revisions of: (1) and (2), percentage of routines with precondition (*pre* in the legend), with postcondition (*post*), and of classes with invariant (*inv*); (3), average number of clauses in contracts; (4), number of changes to implementation and specification (*body+spec*), to implementation only (*body only*), and change to *spec*ification *only*. When present, a thin gray line plots the total number of routine in the project (scaled). Similar plots for all projects are available [7,3].



all revisions: the latter is at least five times larger. suggesting that deviations from the average are normally small. Section 3.2 gives a quantitative confirmation of this hint about the stability of specification amount over time.

The EiffelBase project—a large standard library used in most Eiffel projects—is a good "average" example of how contracts may materialize over a project's lifetime. After an initial fast growing phase (see the first plot in Table 3), corresponding to a still incipient design that is taking shape, the percentages of routines and classes with specification stabilize around the median values with some fluctuations that—while still significant, as we comment on later—do not affect the overall trend or the average percentage of specified elements. This two-phase development (initial mutability followed by stability) is present in several other projects of comparable size, and is sometimes extreme, such as for Boogie, where there is a widely varying initial phase, followed by a very stable one where the percentages of elements with specification is practically constant around 30%. Analyzing the commit logs around the revisions of greater instability showed that wild variations in the specified elements coincide with major reengineering efforts. For Boogie, the initial project phase coincides with the porting of a parent project written in Spec# (a dialect of C#), and includes frequent alternations of adding and removing code from the repository; after this phase, the percentage of routines and classes with specification stabilizes to a value close to the median.

There are few outlier projects where the percentage of elements with specification is small, not kept consistent throughout the project's life, or both. Quickgraph, for example, never has more than 4% of classes with an invariant or routines with a postcondition, and its percentage of routines with precondition varies twice between 12% and 21% in about 100 revisions (see complete data in [7]).

In two thirds of the projects, on average $1/3$ *or more of the routines have* ***some*** *specification (pre- or postconditions).*

Public vs. Private Routines. The data analysis focuses on contracts of *public* routines. To determine whether trends are different for *private* routines, we visually inspected the plots [3] and computed the correlation coefficient[2] τ for the evolution of the percentages of specified public routines against those of private routines. The results suggest to partition the projects into three categories. For the 9 projects in the first category—AutoTest, EiffelBase, Boogie, CCI, Dafny, JavaFE, Logging, RCC and Umbra—the correlation is positive ($0.51 \leq \tau \leq 0.94$) and highly significant. The 2 projects in the second category—GoboStructure and Labs—have negative ($\tau \leq -0.47$) and also significant correlation. The remaining 10 projects belong to the third category, characterized by correlations small in absolute value, positive or negative, or statistically insignificant. This partitioning seems to correspond to different approaches to interface design and encapsulation: for projects in the first category, public and private routines always receive the same amount of specification throughout the project's life; projects in the second category show negative correlations that may correspond to changes to the visibility status of a significant fraction of the routines; visual inspection of projects in the third category still suggests positive correlations between public and private routines with

[2] All correlation measures in the paper employ Kendall's rank correlation coefficient τ.

specification, but the occasional redesign upheaval reduces the overall value of τ or the confidence level. In fact, the confidence level is typically small for projects in the third category; and it is not significant ($p = 0.418$) only for EiffelProgramAnalysis which also belongs to the third category. Projects with small correlations tend to be smaller in *size* with fewer routines and classes; conversely, large projects may require a stricter discipline in defining and specifying the interface and its relations with the private parts, and have to adopt consistent approaches throughout their lives.

In roughly half of the projects, the amounts of contracts in ***public*** *and in* ***private*** *routine correlate; in the other half, correlation vanishes due to redesign changes.*

3.2 Contracts and Project Size

The correlation between the number of routines or classes with some specification and the total number of routines or classes (with or without specification) is consistently strong and highly significant. Looking at routines, 10 projects exhibit an almost perfect correlation with $\tau > 0.9$ and $p \sim 0$; only 3 projects show medium/low correlations (Labs and Quickgraph with $\tau = 0.48$, and Logging with $\tau = 0.32$) which are however still significant. The outlook for classes is quite similar: the correlation between number of classes with invariants and number of all classes tends to be high. Outliers are the projects Boogie and JavaFE with the smaller correlations $\tau = 0.28$ and $\tau = 0.2$, but visual inspection still suggests that a sizable correlation exists for Boogie (the results for JavaFE are immaterial since it has only few invariants overall). In all, the absolute number of elements with specification is normally synchronized to the overall size of a project, confirming the suggestion of Section 3.1 that the percentage of routines and classes with specification is *stable* over time.

Having established that, in general, specification and project size have similar trends, we can look into finer-grained variations of specifications over time. To estimate the *relative* effort of writing specifications, we measured the correlation between *percentage* of specified routines or classes and *number* of all routines or all classes.

A first large group of projects, almost half of the total whether we look at routines or classes, show weak or negligible correlations ($-0.35 < \tau < 0.35$). In this majority of projects, the relative effort of writing and maintaining specifications evolves largely independently of the project size. Given that the overall trend is towards stable percentages, the high variance often originates from initial stages of the projects when there were few routines or classes in the system and changes can be momentous. GoboKernel and DirectVCGen are specimens of these cases: the percentage of routines with contracts varies wildly in the first 100 revisions when the system is still small and the developers are exploring different design choices and styles.

Another group of 3 projects (AutoTest, Boogie, and Dafny) show strong *negative* correlations ($\tau < -0.75$) both between percentage of specified routines and number of routines and between percentage of specified classes and number of classes. The usual cross-inspection of plots and commit logs points to two independent phenomena that account for the negative correlations. The first is the presence of large merges of project branches into the main branch; these give rise to strong irregularities in the absolute and relative amount of specification used, and may reverse or introduce new specification styles and policies that affect the overall trends. As evident in the second plot of Table 3, AutoTest epitomizes this phenomenon, with its history clearly partitioned into two parts separated by a large merge at revision 150. The second phenomenon that may account for negative correlations is a sort of "specification fatigue" that kicks in as a project becomes mature and quite large. At that point, there might be diminishing returns for supplying more specification, and so the percentage of elements with specification gracefully decreases while the project grows in size. (This is consistent with Schiller et al.'s suggestion [27] that annotation burden limits the extent to which contracts are used.) The fatigue is, however, of small magnitude if present at all, and may be just be a sign of reached maturity where a solid initial design with plenty of specification elements pays off in the long run to the point that less relative investment is sufficient to maintain a stable level of maintainability and quality.

The remaining projects have significant *positive* correlations ($\tau > 0.5$) between either percentage of specified routines and number of routines or between percentage of specified classes and number of classes, but not both. In these special cases, it looks as if the fraction of programming effort devoted to writing specification tends to increase with the absolute size of the system: when the system grows, proportionally more routines or classes get a specification. However, visual inspection suggests that, in all cases, the trend is ephemeral or contingent on transient phases where the project size changes significantly in little time. As the projects mature and their sizes stabilize, the other two trends (no correlation or negative correlation) emerge in all cases.

*The fraction of routines and classes with some specification is quite **stable** over time. Local exceptions are possible when major redesign changes take place.*

3.3 Kinds of Contract Elements

Do programmers prefer preconditions? Typically, one would expect that preconditions are simpler to write than postconditions (and, for that matter, class invariants): postconditions are predicates that may involve two states (before and after routine execution). Furthermore, programmers have immediate benefits in writing preconditions as opposed to postconditions: a routine's precondition defines the valid input; hence, the stronger it is, the fewer cases the routine's body has to deal with.

Contrary to this common assumption, the data in our study (columns % ROUTINES PRE and POST in Table 2) is not consistently lopsided towards preconditions. 2 projects show no difference in the median percentages of routines with precondition and with postcondition. 10 projects do have, on average, more routines with precondition than routines with postcondition, but the difference in percentage is less than 10% in 5 of those projects, and as high as 39% only in one project (Dafny). The remaining 9 projects even have more routines with postcondition than routines with precondition, although the difference is small (less than 5%) in 5 projects, and as high as 45% only in RCC.

On the other hand, in 17 projects the percentage of routines with some specification (precondition, postcondition, or both) is higher than both percentages of routines with precondition and of routines with postcondition. Thus, we can partition the routines of most projects in three groups of comparable size: routines with only precondition, routines with only postcondition, and routines with both. The 4 exceptions are CCI, Shweet, DirectVCGen, and Umbra where, however, most elements have little specification. In summary, many exogenous causes may concur to determine the ultimate reasons behind picking one kind of contract element over another, such as the project domain and the different usage of different specification elements. Our data is, however, consistent with the notion that programmers choose which specification to write according to context and requirements, not based on a priori preferences. It is also consistent with Schiller et al.'s observations [27] that contract usage follows different patterns in different projects, and that programmers are reluctant to change their preferred usage patterns—and hence patterns tend to remain consistent within the same project.

A closer look at the projects where the difference between percentages of routines with precondition and with postcondition is significant (9% or higher) reveals another interesting pattern. All 6 projects that favor preconditions are written in C# or Java: Dafny, Labs, Quickgraph, Shweet, ESCJava (third plot in Table 3, after rev. 400), and JavaFE; conversely, the 3 of 4 projects that favor postconditions are in Eiffel (AutoTest, GoboKernel, and GoboTime), whereas the fourth is RCC written in Java. A possible explanation for this division involves the longer time that Eiffel has supported contracts and the principal role attributed to Design by Contract within the Eiffel community.

Preconditions and postconditions are used ***equally frequently*** *across most projects.*

Class Invariants. Class invariants have a somewhat different status than pre- or postconditions. Since class invariants must hold between consecutive routine calls, they define object consistence, and hence they belong to a different category than pre- and postconditions. The percentages of classes with invariant (% CLASSES INV in Table 2) follow similar trends as pre- and postconditions in most projects in our study. Only 4 projects stick out because they have 4% or less of classes with invariant, but otherwise make a significant usage of other specification elements: Quickgraph, EiffelProgramAnalysis, Shweet, and DirectVCGen.[3] Compared to the others, Shweet has a short history and EiffelProgramAnalysis involves students as main developers rather than professionals. Given that the semantics of class invariants is less straightforward than that of pre- and postconditions—and can become quite intricate for complex programs [1]—this might be a factor explaining the different status of class invariants in these projects. A specific design style is also likely to influence the usage of class invariants, as we further comment on in Section 3.4.

Kinds of Constructs. An additional classification of contracts is according to the constructs they use. We gathered data about constructs of three types: expressions involving checks that a reference is **Void** (Eiffel) or **null** (C# and Java); some form of finite quantification (constructs for $\forall/\exists$ over containers exist for all three languages); and **old** expressions (used in postconditions to refer to values in the pre-state). **Void/null** checks are by far the most used: in Eiffel, 36%–93% of preconditions, 7%–62% of postconditions, and 14%–86% of class invariants include a **Void** check; in C#, 80%–96% of preconditions contain **null** checks, as do 34%–92% of postconditions (the only exception is CCI which does not use postconditions) and 97%–100% of invariants (exceptions

[3] While the projects CCI and Umbra have few classes with invariants (4%–6%), we don't discuss them here because they also only have few routines with preconditions or postconditions.

are Quickgraph at 20% and Shweet which does not use invariants); in Java, 88%–100% of preconditions, 28%–100% of postconditions, and 50%–77% of class invariants contain **null** (with the exception of Umbra which has few contracts in general). **Void/null** checks are simple to write, and hence cost-effective, which explains their wide usage; this may change in the future, with the increasing adoption of static analyses which supersede such checks [19,4]. The predominance of simple contracts and its justification have been confirmed by others [27].

At the other extreme, quantifications are very rarely used: practically never in pre- or postconditions; and very sparsely (1%–10% of invariants) only in AutoTest, Boogie, Quickgraph, ESCJava, and JavaFE's class invariants. This may also change in the future, thanks to the progresses in inferring complex contracts [11,30,29], and in methodological support [24].

The usage of **old** is more varied: C# postconditions practically don't use it, Java projects rarely use it (2%–3% of postconditions at most), whereas it features in as many as 39% of postconditions for some Eiffel projects. Using **old** may depend on the design style; for example, if most routines are side-effect free and return a value function solely of the input arguments there is no need to use **old**.

*The overwhelming majority of contracts involves **Void/null** checks.*
In contrast, quantifiers appear very rarely in contracts.

3.4 Contract Size and Strength

The data about specification *size* (and strength) partly vindicates the intuition that preconditions are more used. While Section 3.3 showed that routines are not more likely to have preconditions than postconditions, preconditions have more clauses on average than postconditions in all but the 3 projects GoboTime, ESCJava, and Logging. As shown in columns AVG ROUTINES PRE and POST of Table 2, the difference in favor of preconditions is larger than 0.5 clauses in 9 projects, and larger than 1 clause in 3 projects. CCI never deploys postconditions, and hence its difference between pre- and postcondition clauses is immaterial. GoboTime is a remarkable outlier: not only do twice as many of its routines have a postcondition than have precondition, but its average postcondition has 0.66 more clauses than its average precondition. ESCJava and Logging also have larger postconditions on average but the size difference is less conspicuous (0.25 and 0.32 clauses). We found no simple explanation for these exceptions, but they certainly are the result of deliberate design choices.

The following two facts corroborate the idea that programmers tend to do a better job with preconditions than with postconditions—even if they have no general preference for one or another. First, the default "trivial" precondition *true* is a perfectly reasonable precondition for routines that compute total functions—defined for every value of the input; a trivial postcondition is, in contrast, never satisfactory. Second, in general, "strong" postconditions are more complex than "strong" preconditions [24] since they have to describe more complex relations.

Class invariants are not directly comparable to pre- and postconditions, and their usage largely depends on the design style. Class invariants apply to all routines and attributes of a class, and hence they may be used extensively and involve many clauses;

conversely, they can also be replaced by pre- and postconditions in most cases, in which case they need not be complex or present at all. In the majority of projects (15 out of 21), however, class invariants have more clauses on average than pre- and postconditions. We might impute this difference to the traditional design principles for object-oriented contract-based programming, which attribute a significant role to class invariants [18,5,25] as the preferred way to define valid object state.

In over eighty percent of the projects, the average ***preconditions*** *contain* ***more clauses*** *than the average postconditions.*

Section 3.1 observed the prevailing stability over time of routines with specification. Visual inspection and the values of standard deviation point to a qualitatively similar trend for specification size, measured in number of clauses. In the first revisions of a project, it is common to have more varied behavior, corresponding to the system design being defined; but the average strength of specifications typically reaches a plateau, or varies quite slowly, in mature phases.

Project Labs is somewhat of an outlier, where the evolution of specification strength over time has a rugged behavior (see [7] for details and plots). Its average number of class invariant clauses has a step at about revision 29, which corresponds to a merge, when it suddenly grows from 1.8 to 2.4 clauses per class. During the few following revisions, however, this figure drops quickly until it reaches a value only slightly higher than what it was before revision 29. What probably happened is that the merge mixed classes developed independently with different programming styles (and, in particular, different attitudes towards the usage of class invariants). Shortly after the merge, the developers refactored the new components to make them comply with the overall style, which is characterized by a certain average invariant strength.

One final, qualitative, piece of data about specification strength is that in a few projects there seems to be a moderate increase in the strength of postconditions towards the latest revisions of the project. This observation is however not applicable to any of the largest and most mature projects we analyzed (e.g., EiffelBase, Boogie, Dafny).

The average ***size*** *(in number of clauses) of specification elements is stable over time.*

3.5 Implementation vs. Specification Changes

Contracts are *executable* specifications; normally, they are checked at runtime during debugging and regression testing sessions (and possibly also in production releases, if the overhead is acceptable, to allow for better error reporting from final users). Specifically, most applications and libraries of our study are actively used and maintained. Therefore, their contracts cannot become grossly misaligned with the implementation.

A natural follow-up question is then whether contracts change more often or less often than the implementations they specify. To answer, we compare two measures in the projects: for each revision, we count the number of routines with changed body and changed specification (pre- or postcondition) and compare it to the number of routines with changed body and unchanged specification. These measures aggregated over all revisions determine a pair of values (c_P, u_P) for each project P: c_P characterizes the frequency of changes to implementations that also caused a change in the contracts, whereas u_P characterizes the frequencies of changes to implementations only. To avoid that few revisions with very many changes dominate the aggregate values for a project, each revision contributes with a binary value to the aggregate value of a project: 0 if no routine has undergone a change of that type in that revision, and 1 otherwise.[4] We performed a Wilcoxon signed-rank test comparing the c_P's to the u_P's across all projects to determine if the median difference between the two types of events (changed body with and without changed specification) is statistically significant. The results confirm with high statistical significance ($V = 0$, $p = 9.54 \cdot 10^{-7}$, and large effect size—Cohen's $d > 0.99$) that specification changes are quite infrequent compared to implementation changes for the same routine. Visual inspection also confirms the same trend: see the last plot in Table 3 about Boogie. A similar analysis ignoring routines with trivial (empty) specification leads to the same conclusion also with statistical significance ($V = 29$, $p = 4.78 \cdot 10^{-3}$, and medium effect size $d > 0.5$).

When specifications do change, what happens to their *strength* measured in number of clauses? Another Wilcoxon signed-rank test compares the changes to pre- and postconditions and class invariants that added clauses (suggesting strengthening) against those that removed clauses (suggesting weakening). Since changes to specifications are in general infrequent, the results were not as conclusive as those comparing specification and implementation changes. The data consistently points towards strengthening being more frequent than weakening: $V = 31.5$ and $p < 0.02$ for precondition changes; $V = 29$ and $p < 0.015$ for postcondition changes; $V = 58.5$ and $p = 0.18$ for invariant changes. The effect sizes are, however, smallish: Cohen's d is about 0.4, 0.42, and 0.18 for preconditions, postconditions, and invariants. In all, the effect of strengthening being more frequent than weakening seems to be real but more data is needed to obtain conclusive evidence.

*The **implementation** of an average routine changes much more frequently than its **specification**.*

3.6 Inheritance and Contracts

Inheritance is a principal feature of object-oriented programming, and involves contracts as well as implementations; we now evaluate its effects on the findings previously discussed.

We visually inspected the plots and computed correlation coefficients for the percentages and average strength of specified elements in the flat (explicitly including all routines and specification of the ancestor classes) and non-flat (limited to what appears in the class text) versions of the classes. In the overwhelming majority of cases, the correlations are high and statistically significant: 16 projects have $\tau \geq 0.54$ and $p < 10^{-9}$ for the percentage of routines with specification; 17 projects have $\tau \geq 0.66$ and $p \sim 0$ for the percentage of classes with invariant; 12 projects have $\tau \geq 0.58$ and $p < 10^{-7}$ for the average precondition and postcondition strength (and 7 more projects still have

[4] Using other "reasonable" aggregation functions (including exact counting) leads to qualitatively similar results.

$\tau \geq 0.33$ and visually evident correlations); and 15 projects have $\tau \geq 0.45$ and $p \sim 0$ for the average invariant strength. The first-order conclusion is that, in most cases, ignoring the inherited specification does not preclude understanding qualitative trends.

What about the remaining projects, which have small or insignificant correlations for some of the measures in the flat and non-flat versions? Visual inspection often confirms the absence of significant correlations, in that the measures evolve along manifestly different shapes in the flat or non-flat versions; the divergence in trends is typically apparent in the revisions where the system size changes significantly, where the overall design—and the inheritance hierarchy—is most likely to change. To see if these visible differences invalidate some of the findings discussed so far, we reviewed the findings against the data for *flat* classes. The big picture was not affected: considering inheritance may affect the measures and offset or bias some trends, but the new measures are still consistent with the same conclusions drawn from the data for non-flat classes. Future work will investigate whether this result is indicative of a mismatch between the semantics of inheritance and how it is used in practice [28,26]. (See the extended version [7] for details.)

*Qualitative trends of measures involving contracts do **not** change significantly whether we consider or ignore **inherited** contracts.*

4 Threats to Validity

Construct Validity. Using the number of clauses as a proxy for the strength of a specification may produce imprecise measures; Section 2, however, estimated the imprecision and showed it is limited, and hence an acceptable trade-off in most cases (also given that computing strength exactly is infeasible). Besides, the number of clauses is still a valuable size/complexity measure in its own right (Section 3.4).

Internal Validity. Since we targeted object-oriented languages where inheritance is used pervasively, it is essential that the inheritance structure be taken into account in the measures. We fully addressed this major threat to internal validity by analyzing all projects twice: in non-flat and flat version (Section 3.6).

External Validity. Our study is restricted to three formalisms for writing contract specifications. While other notations for contracts are similar, we did not analyze other types of formal specification, which might limit the generalizability of our findings. In contrast, the restriction to open-source projects does not pose a serious threat to external validity in our study, because several of our projects are mainly maintained by professional programmers (EiffelBase and Gobo projects) or by professional researchers in industry (Boogie, CCI, Dafny, and Quickgraph).

An important issue to warrant external validity involves the *selection of projects.* We explicitly targeted projects that make a non-negligible usage of contracts (Section 2), as opposed to the overwhelming majority that only include informal documentation or no documentation at all. This deliberate choice limits the generalizability of our findings, but also focuses the study on understanding how contracts can be seriously used in practice. A related observation is that the developers of several of the study's projects are supporters of using formal specifications. While this is a possible source of bias it also contributes to reliability of the results: since we are analyzing good practices and success stories of writing contracts, we should target competent programmers with sufficient experience, rather than inexpert novices. Besides, Schiller et al.'s independent analysis [27] of some C# projects using CodeContracts also included in our study suggests that their developers are hardly fanatic about formal methods, as they use contracts only to the extent that it remains inexpensive and cost-effective, and does not require them to change their programming practices.

Nevertheless, to get an idea of whether the programmers we studied really have incomparable skills, we also set up a small *control group*, consisting of 10 projects developed by students of a software engineering course involving students from universities all around the world. In summary (see [7] for details), we found that several of the trends measured with the professional programmers were also present in the student projects—albeit on the smaller scale of a course project. This gives some confidence that the big picture outlined by this paper's results somewhat generalizes to developers willing to spend some programming effort to write contracts.

5 Related Work

To our knowledge, this paper is the first quantitative empirical study of specifications in the form of contracts and their evolution together with code. Schiller et al. [27] study C# projects using CodeContracts (some also part of our study); while our and their results are not directly comparable because we take different measures and classify contract usage differently, the overall qualitative pictures are consistent and nicely complementary. In the paper we also highlighted a few points where their results confirm or justify ours. Schiller et al. do not study contract *evolution*; there is evidence, however, that other forms of documentation—e.g., comments [9], APIs [13], or tests [32]—evolve with code.

A well-known problem is that specification and implementation tend to diverge over time; this is more likely for documents such as requirements and architectural designs that are typically developed and stored separately from the source code. Much research has targeted this problem; specification refinement, for instance, can be applied to software revisions [10]. Along the same lines, some empirical studies analyzed how requirements relate to the corresponding implementations; [12], for example, examines the co-evolution of certain aspects of requirements documents with change logs and shows that topic-based requirements traceability can be automatically implemented from the information stored in version control systems.

The information about the usage of formal specification by programmers is largely anecdotal, with the exceptions of a few surveys on industrial practices [2,31]. There is, however, some evidence of the usefulness of contracts and assertions. [15], for example, suggests that increases of assertions density and decreases of fault density correlate. [20] reports that using assertions may decrease the effort necessary for extending existing programs and increase their reliability. In addition, there is evidence that developers are more likely to use contracts in languages that support them natively [2]. As the technology to infer contracts from code reaches high precision levels [6,30], it is natural to compare automatically inferred and programmer-written contracts; they turn out to be, in general, different but with significant overlapping [23].

6 Concluding Discussion and Implications of the Results

Looking at the big picture, our empirical study suggests a few actionable remarks. *(i)* The effort required to make a quantitatively significant usage of lightweight specifications is sustainable consistently over the lifetime of software projects. This supports the practical applicability of methods and processes that rely on *some* form of rigorous specification. *(ii)* The overwhelming majority of contracts that programmers write in practice are short and simple. This means that, to be practical, methods and tools should make the best usage of such simple contracts or acquire more complex and complete specifications by other means (e.g., inference). It also encourages the usage of simple specifications early on in the curriculum and in the training of programmers [14]. *(iii)* In spite of the simplicity of the contracts that are used in practice, developers who commit to using contracts seem to stick to them over an entire project lifetime. This reveals that even simple specifications bring a value that is worth the effort: a little specification can go a long way. *(iv)* Developers often seem to adapt their contracts in response to changes in the design; future work in the direction of facilitating these adaptations and making them seamless has a potential for a high impact. *(v)* A cornerstone software engineering principle—stable interfaces over changing implementations—seems to have been incorporated by programmers. An interesting follow-up question is then whether this principle can be leveraged to improve not only the reusability of software components but also the *collaboration* between programmers in a development team. *(vi)* Somewhat surprisingly, inheritance does not seem to affect most qualitative findings of our study. The related important issue of how *behavioral subtyping* is achieved in practice [26] belongs to future work, together with several other follow-up questions whose answers can build upon the foundations laid by this paper's results.

Acknowledgments. Thanks to Sebastian Nanz for comments on a draft of this paper; and to Todd Schiller, Kellen Donohue, Forrest Coward, and Mike Ernst for sharing a draft of their paper [27] and comments on this work.

References

1. Barnett, M., Fähndrich, M., Leino, K.R.M., Müller, P., Schulte, W., Venter, H.: Specification and verification: the Spec# experience. Comm. ACM 54(6), 81–91 (2011)
2. Chalin, P.: Are practitioners writing contracts? In: Butler, M., Jones, C.B., Romanovsky, A., Troubitsyna, E. (eds.) Fault-Tolerant Systems. LNCS, vol. 4157, pp. 100–113. Springer, Heidelberg (2006)
3. http://se.inf.ethz.ch/data/coat/
4. Dietl, W., Dietzel, S., Ernst, M.D., Muslu, K., Schiller, T.W.: Building and using pluggable type-checkers. In: ICSE, pp. 681–690. ACM (2011)
5. Drossopoulou, S., Francalanza, A., Müller, P., Summers, A.J.: A unified framework for verification techniques for object invariants. In: Vitek, J. (ed.) ECOOP 2008. LNCS, vol. 5142, pp. 412–437. Springer, Heidelberg (2008)
6. Ernst, M.D., Perkins, J.H., Guo, P.J., McCamant, S., Pacheco, C., Tschantz, M.S., Xiao, C.: The Daikon system for dynamic detection of likely invariants. Sci. Comput. Program. 69, 35–45 (2007)
7. Estler, H.C., Furia, C.A., Nordio, M., Piccioni, M., Meyer, B.: Contracts in practice (2013), extended version with appendix http://arxiv.org/abs/1211.4775

8. Fähndrich, M., Barnett, M., Logozzo, F.: Embedded contract languages. In: SAC, pp. 2103–2110. ACM (2010)
9. Fluri, B., Würsch, M., Gall, H.: Do code and comments co-evolve? on the relation between source code and comment changes. In: WCRE, pp. 70–79. IEEE (2007)
10. García-Duque, J., Pazos-Arias, J., López-Nores, M., Blanco-Fernández, Y., Fernández-Vilas, A., Díaz-Redondo, R., Ramos-Cabrer, M., Gil-Solla, A.: Methodologies to evolve formal specifications through refinement and retrenchment in an analysis-revision cycle. Requirements Engineering 14, 129–153 (2009)
11. Henkel, J., Reichenbach, C., Diwan, A.: Discovering documentation for Java container classes. IEEE Trans. Software Eng. 33(8), 526–543 (2007)
12. Hindle, A., Bird, C., Zimmermann, T., Nagappan, N.: Relating requirements to implementation via topic analysis. In: ICSM (2012)
13. Kim, M., Cai, D., Kim, S.: An empirical investigation into the role of API-level refactorings during software evolution. In: ICSE, pp. 151–160. ACM (2011)
14. Kiniry, J.R., Zimmerman, D.M.: Secret ninja formal methods. In: Cuellar, J., Sere, K. (eds.) FM 2008. LNCS, vol. 5014, pp. 214–228. Springer, Heidelberg (2008)
15. Kudrjavets, G., Nagappan, N., Ball, T.: Assessing the relationship between software assertions and faults: An empirical investigation. In: ISSRE, pp. 204–212 (2006)
16. Leavens, G.T., Baker, A.L., Ruby, C.: JML: A notation for detailed design. In: Behavioral Specifications of Businesses and Systems, pp. 175–188. Kluwer Academic Publishers (1999)
17. Martin, J.K., Hirschberg, D.S.: Small sample statistics for classification error rates II. Tech. rep., CS Department, UC Irvine (1996), http://goo.gl/Ec8oD
18. Meyer, B.: Object Oriented Software Construction, 2nd edn. Prentice Hall PTR (1997)
19. Meyer, B., Kogtenkov, A., Stapf, E.: Avoid a Void: the eradication of null dereferencing. In: Reflections on the Work of C.A.R., pp. 189–211. Springer (2010)
20. Müller, M.M., Typke, R., Hagner, O.: Two controlled experiments concerning the usefulness of assertions as a means for programming. In: ICSM, pp. 84–92 (2002)
21. Parnas, D.L.: On the criteria to be used in decomposing systems into modules. Commun. ACM 15(12), 1053–1058 (1972)
22. Parnas, D.L.: Precise documentation: The key to better software. In: The Future of Software Engineering, pp. 125–148. Springer (2011)
23. Polikarpova, N., Ciupa, I., Meyer, B.: A comparative study of programmer-written and automatically inferred contracts. In: ISSTA, pp. 93–104 (2009)
24. Polikarpova, N., Furia, C.A., Pei, Y., Wei, Y., Meyer, B.: What good are strong specifications? In: ICSE, pp. 257–266. ACM (2013)
25. Polikarpova, N., Tschannen, J., Furia, C.A., Meyer, B.: Flexible invariants through semantic collaboration. In: Jones, C., Pihlajasaari, P., Sun, J. (eds.) FM 2014. LNCS, vol. 8442, pp. 505–520. Springer, Heidelberg (2014)
26. Pradel, M., Gross, T.R.: Automatic testing of sequential and concurrent substitutability. In: ICSE, pp. 282–291. ACM (2013)
27. Schiller, T.W., Donohue, K., Coward, F., Ernst, M.D.: Writing and enforcing contract specifications. In: ICSE. ACM (2014)
28. Tempero, E., Yang, H.Y., Noble, J.: What programmers do with inheritance in Java. In: Castagna, G. (ed.) ECOOP 2013. LNCS, vol. 7920, pp. 577–601. Springer, Heidelberg (2013)
29. Wasylkowski, A., Zeller, A.: Mining temporal specifications from object usage. Autom. Softw. Eng. 18(3-4), 263–292 (2011)
30. Wei, Y., Furia, C.A., Kazmin, N., Meyer, B.: Inferring better contracts. In: ICSE, pp. 191–200 (2011)
31. Woodcock, J., Larsen, P.G., Bicarregui, J., Fitzgerald, J.: Formal methods: Practice and experience. ACM CSUR 41(4) (2009)
32. Zaidman, A., Van Rompaey, B., Demeyer, S., van Deursen, A.: Mining software repositories to study co-evolution of production and test code. In: ICST, pp. 220 –229 (2008)