Extensions during Software Evolution: Do Objects Meet Their Promise?

Romain Robbes David Röthlisberger Éric Tanter

PLEIAD Laboratory
Computer Science Department (DCC)
University of Chile – Chile
http://pleiad.cl

Abstract. As software evolves, data types have to be extended, possibly with new data variants or new operations. It is now folklore that an object-oriented design supports data extensions well, while a functional design seamlessly supports operation extensions. A large body of programming language research has been devoted to the challenge of properly supporting both kinds of extensions.
While this challenge is well-known from a language design standpoint, it has not been studied empirically. We perform such a study on a large sample of Smalltalk projects (over half a billion lines of code) and their evolution over more than 100,000 committed changes.
Our study of extensions during software evolution finds that extensions are indeed prevalent evolution tasks, and that both kinds of extensions are equally common in object-oriented software. We also discuss findings about the evolution of the kinds of extensions over time, and about the viability of the Visitor pattern as an object-oriented solution to operation extensions. This study suggests that object-oriented design alone is not sufficient, and that practical support for both kinds of program decomposition approaches are in fact needed, either by the programming language or by the development environment.

1 Introduction

Lehman's laws of software evolution [13] tell us that software systems must continuously adapt, or become progressively less useful to their users. Over time, new functionality is added to software systems. Inevitably, some functionality needs to extend existing system components. Depending on the programming paradigm used, different extensions have different consequences.

Extensions can happen along two dimensions: new data variants, or new operations. Object-oriented programming is well-known for seamlessly supporting extensibility of data variants, by introducing new kinds of objects. In contrast, the functional design approach [12]—where the variants of a data type are processed by case-analyzing procedures—is better suited to support additions of new operations, by introducing new procedures. Conversely, supporting new operations for objects requires modifying all object definitions to add new methods,

and adding new data variants in the functional approach implies modifying all existing procedures to handle the new cases.

This complementarity between data types and "procedural data values" (objects) dates back to the work of Reynolds in the 1970s [18] and has been described by other researchers since then (*e.g.* [5, 12, 26]). Supporting both forms of extensions appropriately is a challenge that has a strong practical relevance, because the choice of a programming paradigm (or design approach) greatly influences the kind of extension that is supported in a localized manner, without modifying existing code. For instance, choosing an object-oriented decomposition to implement a system whose evolution predominantly involves operation extensions is like using a hammer to paint a wall: possible, but painful. The object-oriented programming community has in fact designed a solution to handle operation extensions, called the Visitor design pattern [7]. A visitor makes it possible to turn an operation extension scenario into a data extension scenario. But once adopted, the Visitor pattern complicates data extensions. Many programming language constructs have been proposed in order to support both dimensions of extension in a modular (and type safe) manner (*e.g.* [12, 15, 24, 27]).

As a matter of fact, the literature on object-oriented programming very often illustrates the superiority of objects in dealing with software evolution by showcasing data extension scenarios (see Booch [4] and Shalloway Trott [22] for two popular books). However, there is no empirical data on how frequently such extensions do occur, nor is there evidence that data extensions are significantly more common than operation extensions, even in object-oriented software.

The extensibility challenge can be looked at both from the point of view of the implementers of a system—the kinds of extensions that have to be dealt with in the evolution of the system—and from the point of view of black-box third-party extensions (the latter is usually seen as the extensibility/expression problem stricto sensu [26, 24]). This work is concerned with the first part of the question. We study the evolution of open-source object-oriented projects through their commit history, looking at how the implementers of a project add new data variants and operations to their class hierarchies as the system evolves. Even if there is no strong impediment to change existing code in this setting, being able to express these extensions modularly does matter; it is well-known that most of the costs of software development are in maintenance and evolution, not in initial development [6].

Concretely, we seek to answer the following research questions:

Q1: Are extensions prevalent in practice? Looking at the evolution of software, is it really the case that new data variants and operations are frequently added? Or are other kinds of changes (*e.g.* changing the implementation of a method) much more common as to render the point moot?

Q2: Are data extensions more common than operation extensions? If object-oriented programming is really superior in dealing with extensible software, object-oriented projects should showcase a far greater number of data extension cases. Is it really the case? Or conversely, are there much more operation extensions, suggesting that another programming abstrac-

tion would be more adequate? Or, are both kinds of extensions similarly important in practice?

Q3: How do extensions occur over time? Over the lifetime of an object-oriented system, do both kinds of extensions manifest regularly? Or are unanticipated design decisions leading to more problematic extension cases as the system ages?

Q4: Is the Visitor pattern a suitable solution? How much is the Visitor pattern used in practice? In cases where it is adopted, are its benefits clearly observable? Are visitor and visited hierarchies more stable than others?

By observing the evolution of a large number of open source Smalltalk projects, this paper presents elements of answers to these questions. Note that because Smalltalk is a dynamically-typed language, this study does not answer these questions in a typed context. Whether or not static typing has an influence on extensions during software evolution is an open question that future studies should address. Also, because Smalltalk is an object-oriented language, we get to observe how object programmers actually benefit (or not) from working in that paradigm. Studies based on other languages, including those that natively support both decomposition approaches, would be needed to answer the research questions above in general. This study is therefore a first step towards providing substance to the long-running debate that takes place in the programming language research community about different forms of data abstractions.

Structure of the paper. Section 2 briefly reviews background and related work. Section 3 describes the experimental setup, explaining how the data was collected and processed. The next four sections report our findings related to the four research questions stated above. Section 8 discusses threats to the validity of this study, and Section 9 concludes.

2 Background and Related Work

We first explain the different kinds of extensions and how to deal with then in object-oriented programming, including the Visitor design pattern. We then review related studies of object-oriented programming practice.

2.1 Extensibility in OOP

Consider the object-oriented design of a simple programming language of arithmetic expressions (Figure 1a)[1]. Expression subclasses `Num` and `Add` implement their own `eval`uation method. A first kind of extension is *data extension*, which consists in adding new data variants; in that case, a new kind of expression (Figure 1b). Note how the object paradigm makes this extension localized: it

[1] Anticipating the fact that we study Smalltalk code, we present the example in a dynamically-typed class-based setting, using inheritance to define data variants.

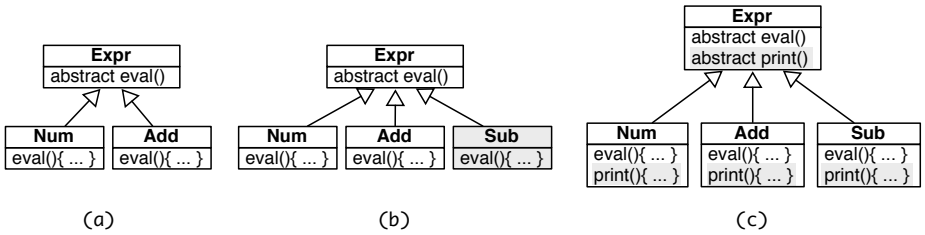

Fig. 1. A class hierarchy (a) and two extensions: data extension (b), and operation extension (c). Changes are highlighted in gray.

is enough to add a new subclass. A second kind of extension is *operation extension*, *e.g.* extending the protocol of expressions, such that they can also be pretty-`print`ed. The object-oriented decomposition is much less suited for this kind of extension, which requires invasive and non-localized modifications of existing classes (Figure 1c).

The Visitor design pattern [7] is the standard way to handle operation extensions in a modular manner in object-oriented programming. It consists of preparing the hierarchy to extend so that it accepts visitor objects (*e.g.* add a method `accept` on each class of the `Expr` hierarchy). A separate hierarchy of visitors is then defined, each for its own operation (*e.g.* `PrintVisitor` extends from `ExprVisitor`). Adding a new operation on the hierarchy is now expressed as adding a new visitor subclass (*e.g.* `TypeVisitor`). Note that applying the Visitor pattern increases the complexity of the system, and that adding a new data variant in the visited hierarchy (*e.g.* `Mult`) implies extending all the visitors.

2.2 Related work

As far as we are aware, there are no empirical studies of the prevalence of extensions during software evolution, nor on comparing the kinds of extensions (data vs. operation) that happen in real world projects. There are however several related studies of characteristics of source code, class hierarchies, and their evolution.

Gîrba *et al.* define a visualization of class hierarchies that incorporates evolutionary metrics, such as age of class, age of inheritance links *etc.* [8]. Based on a study of two open-source systems, they identify several visual patterns to characterize the evolution of the hierarchies. The patterns are however coarse as the unit of granularity is the class, and are aimed to answer general evolution questions, such as the distribution of changes across hierarchies.

A study by van Rysselberghe and Demeyer analyzed hierarchy changes on two Java systems [20]. The exploratory study led to the formulation of 7 hypotheses to be investigated, such as "Hierarchy changes are likely to insert an additional abstraction between the old parent and the center class" and "Inheritance is only rarely replaced by composition". Due to its limited extent this study however only hinted at the answer to the hypotheses; its findings need to be confirmed by a larger-scale study.

Baxter *et al.* performed an empirical study on 16 releases of several Java systems, in order to investigate the distribution of several metrics and whether those followed power laws [3]. Later, Tempero *et al.* focused on the use of inheritance in Java software, using the same corpus (expanded to 93 programs), and a suite of 23 metrics [23]; they found a larger amount of inheritance than they expected: around three-quarters of classes used inheritance (for half of the applications in the corpus). A large-scale survey of programmers by Gorscheck *et al.* [9] found a lack of consensus on what the size of classes and depth of hierarchies should be. A recent large-scale study (2,080 Java programs, found on Sourceforge) by Grechanik *et al.* formulated 32 research questions [10]. Of those, several were related to class hierarchies. They found that almost 50% of the classes are written without using inheritance, and that 71% of the hierarchies had a depth of one. These findings differ somewhat from the ones of Tempero, who found a higher usage of inheritance. However, the metrics used in both studies differ, so comparison is difficult. All of these studies investigate a large number of research questions—trading depth for breadth—, while we focus on the handful of questions that allow us to characterize extensions during the evolution of object-oriented software.

Finally, Aversano *et al.* studied the evolution of several design patterns, including the Visitor pattern, on 3 software systems [2]. They found that classes involved in the Visitor pattern were among the most changed in one of the systems (Eclipse JDT), but that the changes were mostly in the visitor hierarchy, not in the visited hierarchy. The study considers design patterns in general, and is focused on three systems only.

3 Experimental setup

3.1 Data collection

We analyze a large extract of the *Squeaksource*[2] repository for Smalltalk projects written in either Squeak or Pharo (a fork of Squeak). Squeaksource is the foundation for the software ecosystem that the Squeak and Pharo community have built over the years. The majority of Squeak and Pharo developers use Squeaksource as their primary source code repository, making it a nearly complete view of the Squeak and Pharo software ecosystem. The Squeaksource extract we analyze spans 8 years and involves approximately 2,500 projects consisting of more than 95,000 unique classes. Summing all versions of all projects yields nearly 600 million lines of code. Over the course of these 8 years, more than 2,300 developers committed around 110,000 changes to Squeaksource.

To version their source code in Squeaksource, developers use the versioning system *Monticello*. When committing a new version of a project, Monticello stores a snapshot of the entire committed package, without computing the delta to the previous version. Squeaksource is hence a large filesystem directory. With

[2] http://www.squeaksource.com

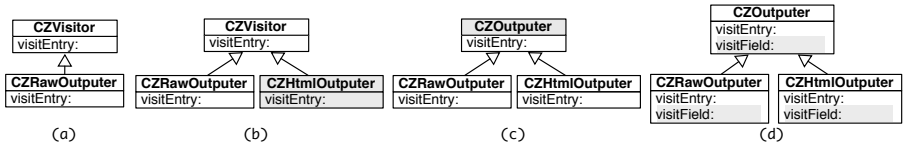

Fig. 2. The Citezen visitor hierarchy: (a) initial version, (b) after a data extension, (c) after renaming the root class, and (d) after an operation extension.

each commit, meta-information is recorded. Monticello versions code at the level of packages, classes, and methods, not at the level of files and lines of code.

To actually analyze the Squeaksource repository we use the Ecco model [19]. Ecco is a lightweight representation of software systems and their versions in an ecosystem. The main unit of abstraction is the system (or software project). For each pair of successive versions of a system, Ecco only keeps the changes between the versions. These changes consist of sets of additions, modifications, and removals of classes and methods in the system. Meta-information such as author, timestamp, and links to one or more ancestors and successors versions are maintained as well. Ecco allows us to effectively and efficiently process and analyze the large set of changes (approx. 13GB of compressed source code) we extracted from Squeaksource.

3.2 Data processing

Before analyzing extension scenarios in the code base, we process the changes from Squeaksource in various steps, described below.

Extension detection. We limit our analysis to the granularity of methods. We do not look into the source code of methods, but stop at the method boundary. Moreover, we only study the additions of classes and methods, but not their modifications. The kinds of extensions can be well quantified by keeping track of additions of classes and methods, since a data extension corresponds to the addition of one or more classes to a hierarchy and an operation extension to the addition of a new method to several classes of a hierarchy.

To detect operation and data extensions, we track the evolution of each class hierarchy in a software project. A real example of such a hierarchy evolution is depicted in Figure 2, which shows the visitor hierarchy of `Citezen`, an application for managing bibtex files on the web. If in a particular change a new class is added to a hierarchy, we consider the addition to be a data extension[3] (shown in Figure 2b where class `CZHtmlOutputer` has been added). The addition of a method with the same name to a least two classes of a hierarchy in a particular change is considered to be an operation extension (*e.g.* Figure 2d, where method `visitField:` is introduced). If a change adds only one method to a class hierarchy, but previous or subsequent changes add methods with that name to the

[3] Adding a new subclass of `Object` is not considered a data extension.

same hierarchy, this single method addition is also considered to be part of an operation extension.

In the case where several classes containing methods with the same name are added to the same hierarchy in the same change, these added methods could actually be detected as operation extensions. For a data extension, we however consider all methods added in the new classes to belong to this data extension, thus no operation extensions are identified in such a scenario.

The root class of any hierarchy can be renamed during the lifetime of a project (*e.g.* root class `CZVisitor` is renamed to `CZOutputer` in Figure 2c). In Monticello, renaming an entity means removing the entity with the old name and adding a new entity with the new name. As we can hence not directly determine a rename of a root class in the changes, we employ an algorithm that tests for every removed root class whether any class newly added in the same change might actually be the renamed version of this root class. For this we compare the set of methods of the removed class with the one of the newly-added class (subclasses of the new and old class are not compared to make the algorithm independent of possible renames to subclasses occurring in the same change). If these sets overlap for at least 80% of the methods, we consider this change as a rename and exchange the old root of the hierarchy with the newly-added class.

Extension weighting. To estimate the effort to realize an extension, it is not enough to compare the number of operation extensions with the number of data extensions. The former is adding just methods, while the latter is adding an entire class to a system. For this reason, we weight a data extension with the number of methods with which the class has been added to the system to obtain a measure for the effort needed to perform an extension. This allows us to compare operation and data extensions in terms of effort, while in the unweighted case we compare the frequency of the two kinds of extensions.

Visitor detection. To detect occurrences of the Visitor pattern and extensions to them, we search for methods whose name is starting with `accept` or `visit`. What follows this prefix is usually the name of the class being visited, *e.g.* `visitField:` typically accepts an instance of class `Field` (or subclasses). The visitor hierarchy is the class hierarchy in which one or more methods following this name pattern are located, while the visited hierarchy is the hierarchy containing the visited class (*e.g.* `Field`). We also support the case when a visitor is visiting various hierarchies, or when a visited hierarchy is visited by several independent visitors.

Aggregation. Beyond class hierarchies, we are also interested in how our analysis translates to the level of *projects*. Recent work by Posnett *et al.* shows that findings at one level of abstraction do not necessarily translate to finer or coarser levels—a phenomenon known as the *ecological fallacy* [17]. For our study we expect that the proportion of projects featuring extensions is higher than the same proportion for class hierarchies. Since the extensions could also be concentrated on a few, possibly large projects, the project level analysis is important to reveal how extensions are distributed over the projects.

Classification. To ease the analysis of the data, we classify the changes, that is, the commits to the projects in three categories: (i) initial, (ii) large, and (iii) selected commits. (i) The first commit to a project reflects the initial development of a project. (ii) Large commits consist of more than 50 added classes and methods. Note that initial commits are often also large commits. (iii) Selected commits are all commits neither classified as initial nor large. This classification is necessary because initial commits carry no change information, and large commits can hardly be meaningfully analyzed because they contain too many changes and are therefore considered as noise [28].

We also classify class hierarchies and projects in two categories: (i) all and (ii) large hierarchies or projects. A large hierarchy has a size of more than five classes. A large project is one with more than 50 classes. This classification is interesting because the impact of an operation extension is arguably more critical in large cases.

Filtering. A large and publicly accessible repository like Squeaksource typically also contains many toy or abandoned projects that would add undesired noise to our analysis. Hence we only take into account class hierarchies that have been changed at least five times and that contain at least two classes (one root and one subclass). Except for the first measurement of Section 4, we only analyze selected commits.

3.3 Basic statistics in Squeaksource

Processing the dataset as discussed gives us the following information to be analyzed in detail in subsequent sections: We start with 111,071 commits; of those, 10,718 commits are classified as *large* or *initial commits*, leaving us with 100,353 *selected* commits. The 95,662 classes are organized in 48,595 hierarchies. Of those, 20,046 have more than one class. This means that 28,550 of the 95,662 classes (29.84%) do not use inheritance, a figure that concords with that reported by Tempero *et al.* [23]. Out of these hierarchies, we select 10,390 satisfying our thresholds of size (at least 2 classes) and activity (at least 5 changes); these are the focus of our analysis. Of these 10,390 class hierarchies, 2,879 have at least an operation or a data extension in selected commits. Also, 2,360 of these 10,390 class hierarchies are classified as large (more than 5 classes). We analyze 2505 projects, of which 569 are classified as large (more than 50 classes); 1036 of the projects feature either operation or data extensions in selected commits.

In a single commit, the largest operation extension we found added 40 methods to the hierarchy, whereas 36 classes were added to the same hierarchy in a single commit. This excludes large and initial commits, including several legitimate operation extensions. These large values lead us to investigate the distribution of the metrics.

Distribution of metrics. Figure 3 shows the distribution of our metrics of interest across projects and class hierarchies. None of the distribution follows the characteristic "bell shape" of a normal distribution. Instead, the overwhelming majority

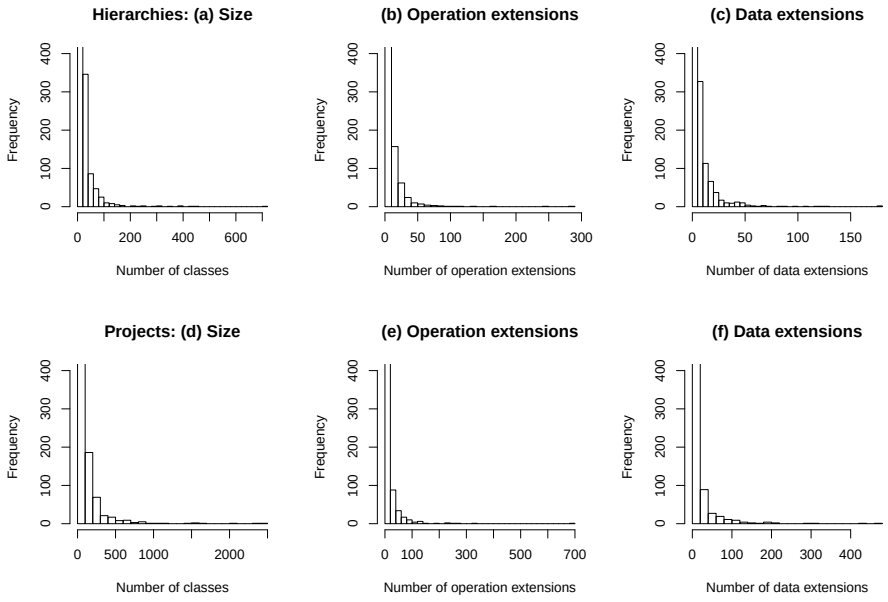

Fig. 3. Histograms showing the distribution of size and extension metrics across hierarchies and projects. Note that the first bar is cut as it would be too tall otherwise.

of observations has very low metric values, and a minority has high values. This observation and the presence of outliers on the tail end of the distributions, leads us to use robust descriptors when characterizing the distributions, *i.e.* the median instead of the mean, and boxplots (showing percentiles) as a visual summary of the distributions. In Section 5.2 we discuss which statistical tests can be used to analyze the data even though its distribution strongly departs from normality and thus breaks the assumptions of most parametric tests.

4 Are extensions prevalent?

We first estimate the prevalence of extensions by looking at the frequency of extension changes in commits. In a second step, we study the frequency of extensions in hierarchies and projects.

4.1 Frequency of extensions in commits

Intuitively, the frequency of extension events across commits tells us how often developers need to perform extensions over time: if extensions are extremely rare, then the challenge of dealing with both kinds of extensions is interesting from a theoretical standpoint, but has little practical impact. Figure 4 shows the proportion of commits featuring operation and data extensions versus commits that

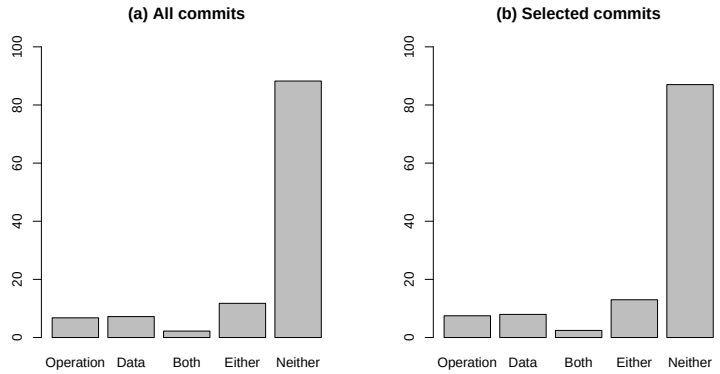

Fig. 4. Percentages of commits featuring extensions. (a) all commits; (b) selected commits

feature neither of these[4]. Of the 111,071 commits we analyzed, 13,063 (11.76%) feature either an operation or a data extension. If we only consider selected commits (that is, we filter out both large and initial commits), the proportion rises to 12.99%. This means that in more than 1/8 of the commits, developers perform either a data or an operation extension in the system they work on.

An extension is problematic in an object-oriented program as soon as an operation extension is needed. We can see that operation extensions occur in 6.77% of all commits (7.48% of selected commits).

4.2 Frequency of extensions in class hierarchies and projects

To view the problem from another angle, we also measure the proportion of hierarchies that feature extensions at any given point in their life. This gives the proportion of hierarchies for which a developer will be expected to perform extensions. Figure 5 shows the proportions of hierarchies featuring at least one extension during their lifetime, versus hierarchies that do not. Out of the 10,390 hierarchies we observed, 27.70% (2,879) become subject of extensions sooner or later. Clearly, a large portion of hierarchies need refinements over time. Importantly, 19.35% of all class hierarchies are subject to operation extensions, which are not modularly supported by objects.

Intuitively, extensions are more problematic for larger hierarchies, where the complexity is higher. We measured the proportion of large hierarchies that are subject to extensions. We find that an overwhelming majority (1,883 out of 2,360, *i.e.* 79.81%) of these large hierarchies feature extensions. Also, more than 62.48% of these hierarchies are subject to operation extensions. Across large, more complex hierarchies, the modularity issue to express extensions is no longer a minority case; it is the norm.

[4] We discuss the relative prevalence of both kinds of extensions in Section 5.

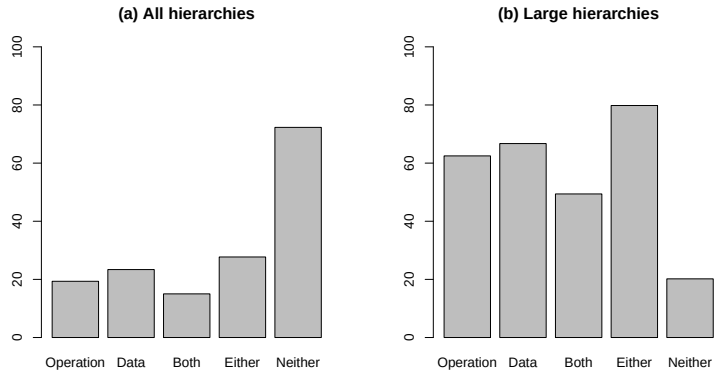

Fig. 5. Percentages of hierarchies featuring extensions. (a) all hierarchies; (b) large hierarchies only (5 or more classes).

For space reasons, we do not provide graphs for projects featuring extensions. In brief, 41% of all projects feature extensions (data extensions: 37.16%; operation extensions: 33.35%; both: 29.06%). Almost all (95%) large projects (*i.e.* projects with more than 50 classes) have to deal with extensions (data extensions: 87.07 %; operation extensions: 89.11%; both: 81.63%).

4.3 Executive Summary

Extensions regularly occur in practice: one out of eight commits (13%) features an extension. Further, a fifth of all class hierarchies have to be extended with new operations; this rate increases to over 60% for large hierarchies. We can conclude that developers are often confronted with extensions that are not modularly supported by object-oriented design. Moreover, for large hierarchies—where one can suppose the impact is more severe—the problem is all the more prevalent.

Far from being a theoretical curiosity, properly supporting both kinds of extensions is of practical concern for software developers, and hence effectively deserves the attention of the community.

5 Comparing data and operation extensions

Having established that extensions are prevalent, we now focus on the distribution of the extension cases across the two categories of extensions. Underlying the research question is the intuition that if the object-oriented paradigm is well-suited for most kinds of evolutions, we expect data extensions to be *much more* common than operation extensions.

5.1 Frequency of both kinds of events

We have already seen in the previous section that both kinds of extensions happen in practice. Looking back at Figure 4 and Figure 5, we notice that both

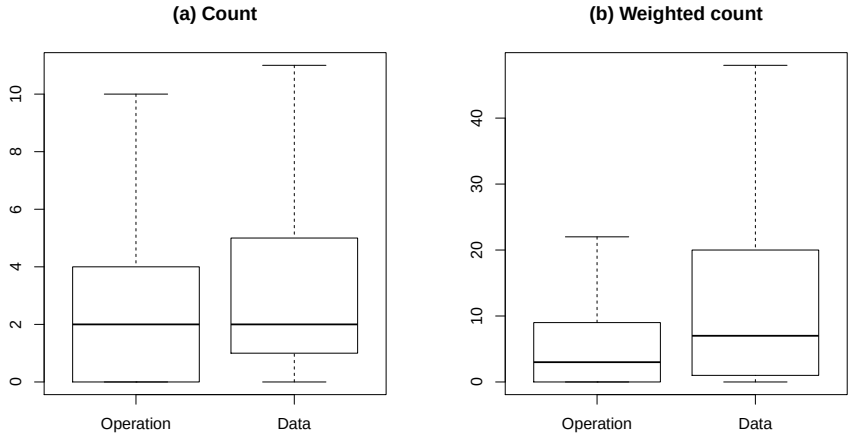

Fig. 6. boxplots of distribution of extensions for hierarchies featuring them. (a) unweighted; (b) weighted.

types of extensions happen with a somewhat similar frequency (*i.e.* 7–8% of all commits, 60–65% of large hierarchies, etc.) and routinely overlap[5]. This gives us a first impression that operation extensions are actually *not* uncommon; rather, they seem to occur with relatively the same frequency as data extensions.

To investigate the problem more closely, we look at the distributions of both kinds of events, for the subset of hierarchies which experience these events. In order to evaluate the problem beyond frequency, we also look at the distributions of the weighted coefficients we introduced earlier—where the weight of an extension is defined by the number of methods it contains—, which gives us a more accurate metric of the effort involved in each kind of extension.

Figure 6 shows the distributions of both kinds of events as box-and-whiskers plots, for both unweighted—to evaluate frequency—and weighted—to evaluate effort—distributions, for the 27% of hierarchies that feature at least one of the two events during their life time. The thick line across each box represents the median value of the distribution, the boxes delimit the interquartile range (the 50% of the values that are between the 25th and the 75th percentile), while the whiskers depict the 5th and the 95th percentile; outliers are not displayed.

If only frequency is considered (Figure 6a), we see that in both cases, the median value (2) is identical. This tells us that the distributions are very similar in terms of frequency. The impression is reinforced by the significant overlap of the boxes. All in all, both kinds of extensions seem to happen with the same regularity, with data extensions being only slightly more common.

[5] Cases where both kinds of extensions overlap in the same hierarchy are especially interesting because they correspond to scenarios that no single data abstraction mechanism would be able to handle properly.

If we consider effort (Figure 6b), a different picture emerges. The median effort in introducing data extensions is higher (7 methods) than the effort involved in introducing operation extensions (3 methods). However, the boxes still overlap significantly: the 75th percentile of operation extensions is higher than the median of data extensions. If operation extensions need more effort, the difference is not so high that one can ignore operation extensions altogether.

Due to space limitations, we do not provide the graphs for commits and projects. These however feature the same pattern of an extremely large overlap in the unweighted case, and a still large overlap in the weighted case—with the upper quartile of weighted operation extensions above the median of weighted data extensions. For commits, the weighted data extension median is 2, while the weighted operation extension's 3rd quartile is 3; for projects, we have 20 and 25, respectively.

5.2 Quantifying the difference between kinds of extension

A visual inspection of the distributions of extensions shows that the distribution of the two kinds of extensions largely overlap in terms of frequency, and still overlap significantly when effort is taken into account. In this section, we seek to quantify the difference.

Given the large size of our sample, a statistical test such as the non-parametric Mann-Whitney U-test would almost certainly find a difference in the values of the distributions, and being in favor of data extensions. However, such a test would not tell us anything about the magnitude of the difference. As such, we measure the effect size of the differences in the distributions.

The most well-known effect-size metric is Cohen's d; however, it is not robust to departures from normality. As such, we opted for a non-parametric effect size, Vargha and Delaney's $\hat{A}_{12}$ [25]. This effect size measure was recommended by Arcuri and Briand in the case of algorithms whose performance follow geometric distributions which strongly depart from normality [1]. $\hat{A}_{12}$ ranges from 0 to 1, and measures the probability that a value taken at random from the first sample is higher than a value taken at random from the second sample.

In the case of unweighted frequencies of both kinds of extensions, we obtain an $\hat{A}_{12}$ value of 0.5554 in favor of data extensions, *i.e.* there is a 55% probability that a randomly chosen frequency of data extension is higher than a randomly chosen frequency of operation extension. This is very close to 50%, where the effect would be null. Since Cohen's d has well-accepted thresholds for effect sizes, we computed an estimate of the equivalent Cohen's d for this value. Our estimation of Cohen's d gives us 0.03, an effect that is considered as *trivial*, barely worth mentioning[6].

If we weight the measurements by effort, the picture is somewhat different. The advantage towards data extensions increases, with $\hat{A}_{12}$ being 0.6197: A

[6] Cohen's d varies from -1 to 1; the commonly accepted thresholds for effect size are 0.2 (small), 0.5 (medium), and 0.8 (strong). Negative values of d indicate an effect in the opposite direction, and have identical thresholds.

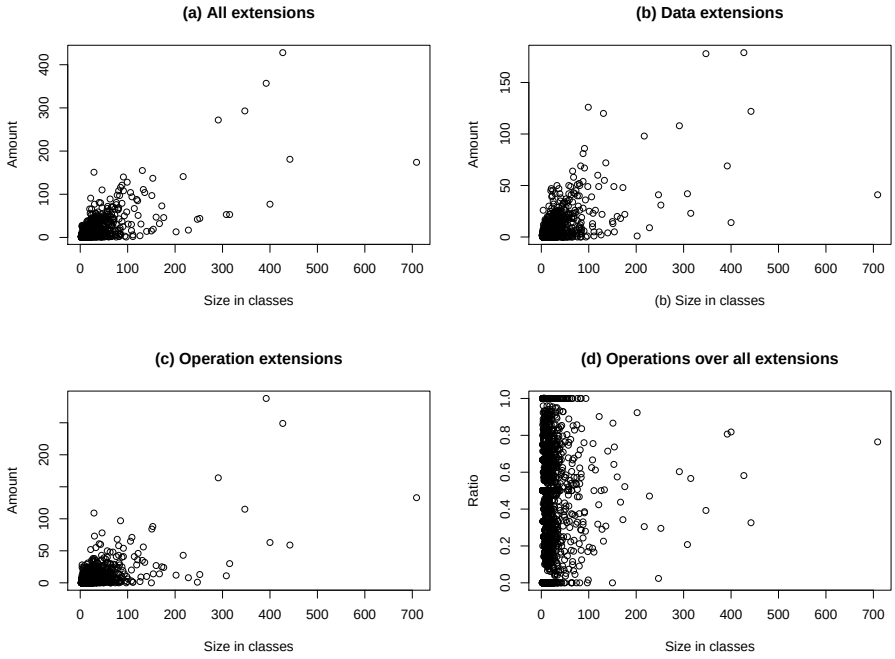

Fig. 7. Effect of size of hierarchies on kinds of extensions.

randomly-chosen weighted data extension count has a 62% probability of being larger than a randomly picked weighted operation extension count. If this higher probability seems reassuring, we do not know how to interpret that value. We again computed an estimate of the equivalent Cohen's d for this probability; we obtained a value of 0.25, which gives us a *small* effect. In other words, if data extensions are more common (barely), and involve more effort (somewhat), a large part of the extensions still are done by adding operations. For the object-oriented paradigm to be most suited for most extensions, we would have expected a much larger advantage in favor of data extensions, with at least a *medium*, if not a *large* effect size.

We quantified the effect size at the level of projects, where we obtained nearly identical results ($\hat{A}_{12}$: 0.5514 (unweighted) and 0.6307 (weighted); estimate of d: 0.05 and 0.25). These findings show that in practice, both kinds of extensions are needed in object-oriented programs; as such, adequate means to express both kinds of extensions are required in order to assist developers.

5.3 Relationship with size of hierarchies

We now look at how the size of hierarchies affects the number of extensions and their kinds. The scatterplots in Figure 7 show the relationship between size of hierarchies and: all extensions (a); data extensions (b); operation extensions (c); and ratio of operation extensions over all extensions (d).

To quantify the relationships, we measure the Spearman correlation between the number of extensions and the size of the hierarchies. Correlation ranges from 1 (perfectly correlated), to -1 (perfect inverse correlation), with 0 being uncorrelated. Spearman's ρ is a rank-based, non-parametric correlation, and as such it is less sensitive to outliers than alternatives (*e.g.* Pearson's product-moment correlation). Commonly-used thresholds for correlation are: 0.1 (small), 0.3 (medium), and (0.5) strong. We also report the statistical significance of the correlations we encounter, using the common threshold of $p < 0.05$ for significance. All the correlations below are highly statistically significant: in all cases, $p \lll 0.01$.

We start with both kinds of extensions taken together (Figure 7a). We see an upward trend (large hierarchies have more extensions) and find a strong correlation ($\rho = 0.67$). This corroborates our findings in Section 4.2, where we found that 80% of the hierarchies with five or more classes had extensions.

Figure 7b shows the relation between the size of hierarchies and the number of data extensions. We see an upward trend as well, giving us the impression that overall larger-sized hierarchies have more data extensions. The Spearman correlation yields a value of $\rho = 0.48$, which qualifies for a *medium* correlation.

The same situation holds with respect to the relationship between operation extensions and size, as shown in Figure 7c. Surprisingly, we observe a higher correlation between size and number of operation extensions, passing the *strong* threshold, with $\rho = 0.55$. If we weight the observations, we see an increase in the correlation for operation extensions ($\rho = 0.59$), and a decrease in the correlation for data extensions ($\rho = 0.42$). We have seen previously that both kinds of extensions are prevalent, with a small advantage for data extensions; here, operation extensions tend to increase more with the size of the hierarchies.

Having observed that operation extensions seem to "take the edge" in large hierarchies, we investigated if this behavior extends to the proportion of extensions. We computed the ratio of operation extensions over all extensions, and investigated its relationship to size. However, as Figure 7d shows, we found no visible relationship: hierarchies are nearly evenly spread across the ratio spectrum. Since the overall difference in correlation was not very large, the relationship practically disappears when the ratio is taken into account. Clearly, there are other factors at play also influencing the relationship between the two variables, as we see next.

In the interest of completeness, we mention that relationships taken at the project level exhibit a similar behavior, having significant, medium-to-strong correlations in the first three cases (a, b, and c).

5.4 Executive summary

Analyzing the frequency and the effort invested in each kind of extension, we see overall that data extensions are slightly more frequent than operation extensions. However, this difference is very small: operation extensions are mostly as frequent as data extensions, and only somewhat smaller. If the extension mechanisms of object-oriented programming was adequate in most cases, the proportion of data extensions would be much larger.

To make matters worse, while both kinds of extensions are unsurprisingly correlated with the size of hierarchies, we find that operation extensions are actually slightly more correlated with size. Large hierarchies seem to necessitate more operation extensions.

6 Extensions and Evolution

In the previous section, we have seen that even if both kinds of extensions are correlated with the size of hierarchies, the ratio of operation extensions over both extensions was not obviously correlated with size. However, there may be other factors influencing this ratio. In particular, Lehman's laws of software evolution [13] say that software systems tend to decay over time, if no effort is undertaken to prevent that. Thus it seems reasonable to think that over time, unanticipated design decisions lead to more extensions that do not fit the class hierarchy, and as such need to be done via operation extensions. Hence, we analyze the proportion of operation extensions out of all extensions over time.

6.1 Introducing periods

To answer this question we split the evolution of class hierarchies in periods. We gather all the commits affecting a candidate hierarchy, sort them according to time, and split the resulting list in 50 slices, each representing one period of the evolution. If a hierarchy was changed less than 50 times, we distribute the changes across the periods as close to being equidistant as possible. Since there is considerable variation in the number of changes between hierarchies, this ensures a uniform distribution of the changes over the 50 periods[7].

We then aggregate all the changes of all the hierarchies that belong to the same period. For each of these sets of changes, we sum the number of operation and data extensions, and compute the ratio of data extensions over all extensions, resulting in a proportion between 0 and 1 for all periods.

We also investigate the phenomenon at the level of projects; there, the only difference is that we gather all the changes related to a project before splitting the history in 50 periods. If a hierarchy is added later in a project, its changes will be distributed across the later periods of the project evolution only.

6.2 Evolution of the ratio of operation extensions

Figure 8 plots the evolution of the proportion of operation extensions among all extensions over time, considering both hierarchies (a) and projects (b). To highlight the overall trend, a smoothed fitted curve is added to the scatterplots.

[7] We contemplated splitting the sets of changes in equal time periods, instead of equal number of commits per period. However, determining the time periods involves computing the time interval based on the first and the last change of the hierarchies. This introduces a bias in the earlier and later periods (more changes are found in the very first and very last periods), hence we discarded that idea.

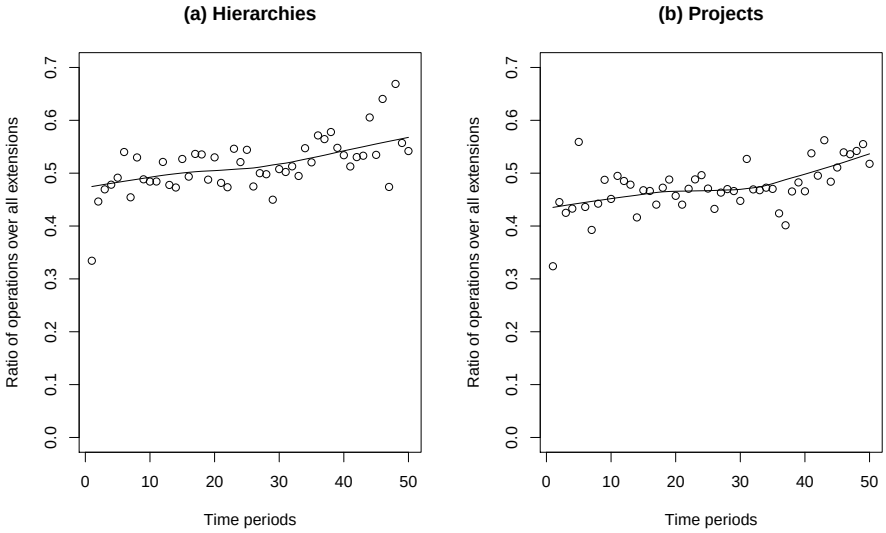

Fig. 8. Proportion of operation extensions among all extensions over time. (a) hierarchies; (b) projects

In both cases, there is clearly an increase of the ratio of operation extensions over all extensions over time. The effect is more pronounced when hierarchies are considered on their own, which is not surprising: a possible reason being that new hierarchies may be added to projects later on. These hierarchies will then be "younger" and for a while offset the upward trend. It is interesting that the smoothed curve on the project scatterplot rises more sharply in the last periods: a possible explanation is that by then, the "young" hierarchies have begun to also become older, and seen their ratio increase, in turn impacting the project.

After a visual check, we quantify the relationship. The Spearman correlation indicates for both cases a significant relationship, which also confirms the visual impression that the effect is more pronounced for hierarchies in isolation than it is for projects. We find a Spearman correlation of $\rho = 0.60$ ($p \lll 0.01$) for hierarchies, and of $\rho = 0.51$ ($p \lll 0.01$) for projects.

If we take weighting into account (not shown in the figure), the relationship—unsurprisingly—drops. It however stays significant. The correlation of the weighted ratio with time for hierarchies is $\rho = 0.38$ ($p = 0.007$), and for projects, $\rho = 0.33$ ($p = 0.018$, less than the usual 0.05 threshold).

Of course, these correlations are not very strong; nor should we expect them to be. There are many more factors, beyond mere time passing, that could explain why a given hierarchy may need more of a certain kind of extension than others.

6.3 Executive summary

If we consider the higher ratio of operation extensions as a sign of decay of object-oriented software, these results certainly confirm Lehman's observations

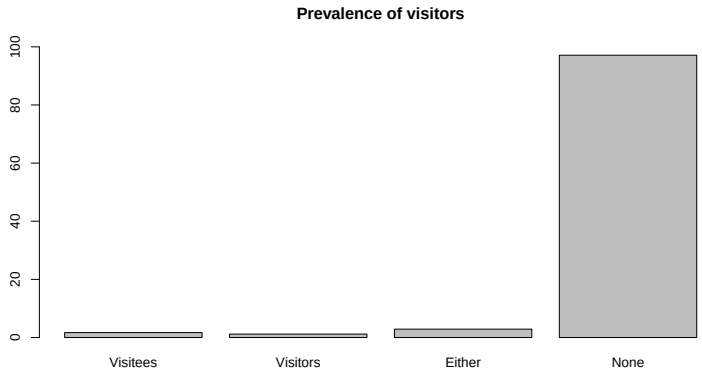

Fig. 9. Proportion of visitors, visited hierarchies, across all hierarchies.

that software systems decay over time. We have found a moderate, yet significant, relationship between the ratio of operation extensions over all extensions and the age (as changes per periods), for both hierarchies and projects.

In our overall analysis, this adds evidence towards the emerging trend that more complex hierarchies (*i.e.* larger, older, etc.) are more confronted with extensions that do not fit the paradigm than other ones. Further, they seem to require proportionally more operation extensions than data extensions. These results cement the relevance of supporting both kinds of extensions adequately, as the most problematic hierarchies are the ones that need solutions the most.

7 Is the Visitor pattern a suitable solution?

The well-known solution to operation extensions in object-oriented software is the Visitor pattern [7], as briefly described in Section 2.1. Is the Visitor pattern enough? We first analyze the prevalence of visitors in our data set, and then look at how both visitor hierarchies and the hierarchies they visit are themselves subject to operation extensions.

7.1 Prevalence of the Visitor pattern

Figure 9 shows the results of our categorization of the hierarchies according to our visitor detection algorithm (Section 3.2). One can clearly see that a minority of classes are involved as either visitors or visitees. Out of the 2,879 hierarchies that experienced at least an extension (shown in the figure), 34 are visitors, and 49 are visitees, corresponding to a total of 2.88% of these hierarchies. In all the 10,271 hierarchies, we find 57 visitor hierarchies, and 62 visited hierarchies, for an even smaller proportion of 1.16% [8].

[8] The discrepancy in number is because there may not be a 1-to-1 mapping between visitors and visited hierarchies

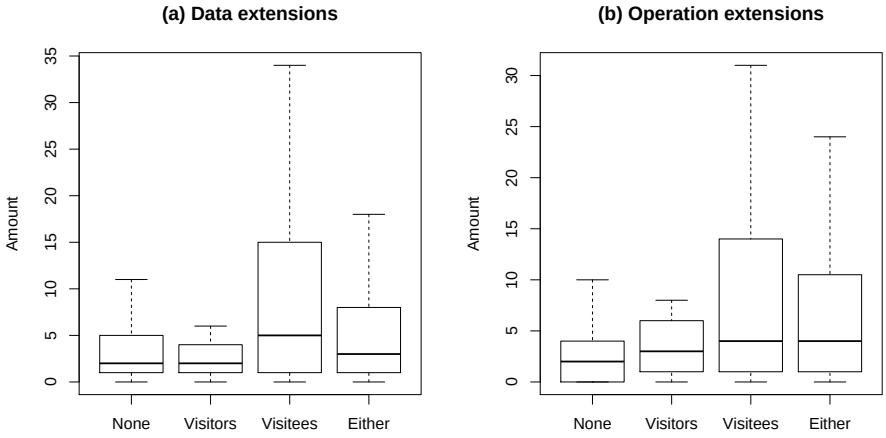

Fig. 10. Distribution of data and operation extensions by role in the Visitor pattern.

All in all, usages of the Visitor pattern are few and far between. If it alleviates the issue of dealing with operation extensions, it cannot do so on a large scale, either because it cannot cover all cases, because few programmers have knowledge of the pattern (which, considering the popularity of design patterns, seems somewhat unlikely), or because the adoption cost of the pattern is judged too high. We also notice that the proportion of visitor and visited classes that experience extensions (83 out of 119, or 69.7%) is much higher than the proportion of hierarchies overall (2,879 out of 10,271, or 28%). This seems to indicate that classes involved in the visitor pattern are extended more than other classes. This warrants further investigation.

7.2 How are visitors and visitees extended?

Considering the documented drawbacks of the Visitor pattern (adding a new class in the visited hierarchy impacts all the visitors), we would expect the uses of the Visitor pattern to follow the Gang of Four's recommendations, and be applied to *stable* visited hierarchies only [7]. This means that visited hierarchies should feature less data extensions, and the corresponding visitor hierarchies should undergo less operation extensions.

Figure 10 shows the distribution of extension metrics, contrasting normal hierarchies, visitor hierarchies, visited hierarchies, and the last two kinds of hierarchies taken together. What we see contradicts our intuition. First, in Figure 10a, visitors seem to exhibit the same number or less data extensions than normal hierarchies, and visited hierarchies seem to feature considerably more! In Figure 10b, we see that visitors seem to feature slightly more operation extensions, and visitees considerably more.

There is, however, an important confounding factor: size. If a hierarchy has many data extensions, it is by definition large. Examining the data, we found that, indeed, an overwhelmingly large proportion of visited hierarchies are large. Thus we account for size in our statistical tests to assess whether the observed effect is significant.

Since there is no statistical test to determine the impact of a confounding factor [16], we employ an alternative strategy. We test for the statistical significance of differences in the number of data and operation extensions, for both visitor and visited hierarchies compared to normal hierarchies, using the Mann-Whitney U-test (the non-parametric equivalent to Student's t-test), under the null hypothesis $H0$ that there is no difference between the quantities of data and operation extensions. In case we find a significant difference and hence reject $H0$, we control for the size factor by doing a subsequent Mann-Whitney test, but this time comparing only the hierarchies with more than five classes.

After doing this procedure, the only significant differences below the $p = 0.05$ threshold were the number of data and operation extensions of visited hierarchies compared to normal hierarchies. However, most visited hierarchies actually have more than ten classes (44 out of 49). Repeating the same procedure but with a threshold of ten classes, both relationships lose their significance[9].

All in all, we can safely assume that any relationship between role (*i.e.* visitor, visitee) in the Visitor pattern and number of extensions is primarily due to the confounding factor of size. This makes classes involved in the Visitor pattern no worse, but also no better, than regular classes, for both roles and both metrics. Overall this suggests that the GoF advice of using the Visitor pattern on stable hierarchies may not be followed in practice. We have observed several examples of operation extensions in visitors that were performed to retrofit the visitors to data extensions in the visited hierarchies.

7.3 Executive summary

We find that the Visitor pattern is not used very often in our dataset. Further, visitor and visited hierarchies seem to feature the same rate of extensions as other hierarchies (when accounting for size). We can conclude that the Visitor pattern is a viable solution only for a subset of all the extension cases. In addition, we noticed that visited hierarchies still suffer from operation extensions, which should normally be handled in the visitors. Finally, the results show that the GoF advice—the Visitor pattern should be applied only to stable hierarchies—is hardly followed in practice. This differs from Aversano's study, which found that visited hierarchies were stable, albeit on three systems only [2].

8 Threats to validity

In this section we report on the threats to validity of our study. We distinguish between (i) construct validity, that is, threats due to how we operationalized

[9] The p-value of 0.06 for operation extensions is close to significance; however, raising the number of classes further eliminates this tenuous relationship.

the measures, (ii) internal validity, that is, threats affecting the measured cause-effect relationship, and (iii) external validity, which refers to threats concerning the generalization of the experiment results.

8.1 Construct validity

By weighting each data extension with the number of methods added along with the new class, we might not correctly represent the severity of a data extension. For instance, after the initial addition of the class in a particular commit, the class might be extended with more methods in subsequent commits, methods that should also be considered when weighting this data extension.

The various thresholds we impose during data analysis (*e.g.* only class hierarchies with more than two classes and that have been changed more than five times are studied), have an influence on how many data and operation extensions we measure. However, we carefully selected these thresholds empirically, that is, by experimenting with different threshold values. The currently selected thresholds are most reasonable given the analyzed data. In the case of the threshold for large commits (addition of more than 50 entities in a commit), we observed that some genuine operation extensions were actually above that threshold; for instance, a polymorphic method was added on 61 classes of the same hierarchy in a single commit.

Since we do not analyze the source code inside methods, we do not account for methods that perform an explicit dispatch based on the type of an object in a functional design manner (*e.g.* `anObject isFoo ifTrue: [...] ...`). These methods are in fact operation extensions in disguise, for which the developer did not adopt the object-oriented paradigm in order to avoid having to add methods in scattered places. As such, we may under-estimate the amount of operation extensions that are performed.

Another source of underestimation of operation extensions is that we do not consider the *class extension* mechanism of Smalltalk. Class extensions are methods added to existing classes in one project by another project. For instance, the class `Object` in the kernel of Pharo Smalltalk has several dozens methods defined by other projects. These are excluded from our analysis, and may reflect potential operation extensions. We counted the number of methods defined as class extensions, and found that they represented 3.79% of all methods (2,732,618 out of 72,028,070 method definitions across all versions). As such, they are unlikely to influence our results.

We only study additions of methods and classes, not their modifications. If we considered modifications as well, we may find a higher proportion of changes related to data and operation extensions, for instance because such extensions tend to trigger more modifications than other additions.

8.2 Internal validity

Squeaksource contains a considerable amount of code duplication, since projects are stored several times in the repository, for instance once as an individual

project and once embedded in another project. In a recent study, we found that 10-15% of the code in Squeaksource is duplicated [21]. This aligns with the code duplication rate found in the literature [11, 14]. The effect of the presence of code duplication on the results of our study is hard to predict. We assume that duplicated projects do not stand out regarding data or operation extensions and hence expect the effect of code duplication to be minimal.

If the same method is added to two unrelated siblings, we count this as an operation extension, even if all other classes in the hierarchy do not either define or inherit the method. Such a case may either be an incomplete operation extension, two unrelated single-method extensions, a bug, or an incremental step towards a consistent extension. In a dynamically-typed language, it is hard to tell whether this scenario corresponds to an operation extension or not, unless we rely on human judgment. This is because object interfaces are totally implicit in such languages. In a statically-typed language, object interfaces are explicit and the type system ensures that an extension of the interface is consistently implemented.

The detection of renames of root classes in a hierarchy is not perfect and might not detect some renames. In such a case we end up with having an old, obsolete hierarchy in our dataset to which we cannot relate any subsequent changes and thus not detect operation or data extensions affecting such a hierarchy. We however expect such cases to be rare and could not find a single false-negative case while overviewing most of the very large hierarchies in Squeaksource.

The visitor detection heuristic we implemented is also not perfect. However, we validated each identified visitor manually and did not find any false-positives, thus the detection algorithm yields a precision of 100%. The recall is not assessable though, our algorithm might not detect all visitors, thus we possibly underestimate the presence of visitors and visited hierarchies. Since we search for variations in terminology (*e.g.* `accept` and `visit` for visitor methods), we expect the recall to be fairly high.

We took dispositions against the ecological fallacy [17]—incorrectly assuming that observations holding at a level of abstraction holds at another level—by systematically verifying that findings we found at the level of class hierarchies also applied at the level of projects, when it was pertinent to do so.

8.3 External validity

The generalization of our study is dependent on how representative the analyzed projects are for object-oriented software projects in general. As Squeaksource is a very large repository containing more than 2,500 projects to which more than 2,300 developers contributed, we expect that very different programming styles and flavors have been applied in these projects, making the analyzed projects well representative of object-oriented software.

A possible bias is that our sample of project contains only open-source software systems. Practices in the industry may differ and limit the generalization of our results. However, access to a large sample of closed-source software systems is notoriously difficult.

Smalltalk is a dynamically-typed programming language. In a statically-typed language, data and operation extensions might be employed differently, following different rules and patterns. It is very hard to assess whether one or both type of extensions are more or less frequent in a statically-typed languages than in its dynamic pendant. Also, we cannot claim that the results we found for Smalltalk also hold for other dynamically-typed object-oriented languages, although we expect to find similar patterns. It would be very interesting to replicate our study for *e.g.* Java and Ruby, to assess the use of data and operation extensions in other object-oriented languages.

Smalltalk is an object-oriented language. The extensibility challenge we studied is a general problem that occurs with other abstraction mechanisms as well. We cannot claim that the results related to the kinds of extensions that occur in Smalltalk projects also apply to other mechanisms. Studying programs written in languages with different mechanisms (*e.g.* ML, Haskell), including combinations of objects and others (*e.g.* Scala, Racket), would be extremely interesting to shed more light on this topic.

9 Conclusions

Reconciling the two kinds of extensions to data types has been a subject of interest for years, if not decades; we assessed the prevalence of this challenge with a large-scale empirical study. Our empirical study of the Squeaksource ecosystem analyzed more than half a billion lines of code, distributed over 2,505 projects and 111,071 commits. Thousands of contributors performed these commits over the course of 8 years.

We found the following:

1. Extensions do occur: one out of eight commits introduces an operation or a data extension; large projects and large hierarchies are more prone to extensions. More than half of the large class hierarchies have to be extended with new operations.
2. Both kinds of extensions happen with roughly the same frequency. When effort is measured, data extensions take a small advantage. However, the margin is very small, so the data-extension friendly mechanism of objects needs supplementation for operation extensions.
3. Over time, projects and hierarchies tend to need more operation extensions, as the new extensions were not envisioned by the initial design. These larger, older hierarchies need better extensibility support all the more.
4. The Visitor pattern, the de-facto solution to modularly support operation extensions in object-oriented software, is not applied frequently. Furthermore, classes involved in the pattern still need operation extensions: in visited classes when the extensions do not fit well the Visitor pattern, and in visitor classes to react to data extensions in the visitees.

We see these findings as a call to the community to continue investigation on this topic, and, perhaps more crucially, to propose solutions to practitioners.

If the first can be done with novel languages, perhaps tool support is best to assist practitioners working on existing systems. For instance, IDEs could provide programmers with a way to switch between a data-centric view and an operation-centric view of the program. The seed of such tool support already exists in the venerable Smalltalk class browser, which is able to display all the implementors of a polymorphic method in a single, editable view. As for the extensibility problem stricto sensu, further studies are needed to see if our results also reflect black-box third-party extension scenarios.

Acknowledgments. We thank the ECOOP reviewers for their helpful comments. R. Robbes and É. Tanter are partially funded by FONDECYT Projects 11110463 and 1110051, respectively. D. Röthlisberger is funded by the Swiss National Science Foundation, SNF Project No. PBBEP2 135018.

References

1. Andrea Arcuri and Lionel C. Briand. A practical guide for using statistical tests to assess randomized algorithms in software engineering. In *Proceedings of ICSE 2011*, pages 1–10, 2011.
2. Lerina Aversano, Gerardo Canfora, Luigi Cerulo, Concettina Del Grosso, and Massimiliano Di Penta. An empirical study on the evolution of design patterns. In *Proceedings of ESEC/SIGSOFT FSE 2007*, pages 385–394, 2007.
3. 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 *Proceedings of OOPSLA 2006*, pages 397–412, 2006.
4. Grady Booch. *Object-Oriented Analysis and Design with Applications*. Addison-Wesley, 1994. 2nd edition.
5. William R. Cook. Object-oriented programming versus abstract data types. In *Proceedings of the REX Workshop/School on the Foundations of Object-Oriented Languages*, volume 173 of *LNCS*. Springer-Verlag, 1990.
6. Len Erlikh. Leveraging legacy system dollars for e-business. *IT Professional*, 2(3):17–23, May 2000.
7. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Professional Computing Series. Addison-Wesley, October 1994.
8. Tudor Gîrba, Michele Lanza, and Stéphane Ducasse. Characterizing the evolution of class hierarchies. In *Proceedings of CSMR 2005*, pages 2–11, 2005.
9. Tony Gorschek, Ewan D. Tempero, and Lefteris Angelis. A large-scale empirical study of practitioners' use of object-oriented concepts. In *Proceedings of ICSE 2010*, pages 115–124, 2010.
10. Mark Grechanik, Collin McMillan, Luca DeFerrari, Marco Comi, Stefano Crespi, Denys Poshyvanyk, Chen Fu, Qing Xie, and Carlo Ghezzi. An empirical investigation into a large-scale java open source code repository. In *Proceedings of ESEM 2010*, pages 11:1–11:10, 2010.
11. Cory J. Kapser and Michael W. Godfrey. Supporting the analysis of clones in software systems: A case study. *Journal of Software Maintenance and Evolution: Research and Practice*, 18, 2006.

12. Shriram Krishnamurthi, Matthias Felleisen, and Daniel P. Friedman. Synthesizing object-oriented and function design to promote reuse. In Eric Jul, editor, *Proceedings of ECOOP 98*, volume 1445 of *LNCS*, pages 91–113, Brussels, Belgium, July 1998. Springer-Verlag.
13. Manny Lehman and Les Belady. *Program Evolution: Processes of Software Change*. London Academic Press, London, 1985.
14. J. Mayrand, C. Leblanc, and E.M. Merlo. Experiment on the automatic detection of function clones in a software system using metrics. In *Proceedings of Software Maintenance 1996*, pages 244 –253, nov 1996.
15. Bruno C. d. S. Oliveira. Modular visitor components: A practical solution to the expression families problem. In Sophia Drossopoulou, editor, *Proceedings of ECOOP 2009*, number 5653 in LNCS, pages 269–293, Genova, Italy, july 2009. Springer-Verlag.
16. Judea Pearl. Why there is no statistical test for confounding, why many think there is, and why they are almost right. Technical report, Department of Statistics, University of California, Los Angeles, 1998.
17. Daryl Posnett, Vladimir Filkov, and Premkumar Devanbu. Ecological inference in empirical software engineering. In *Proceedings of ASE 2011*, pages 362–371, 2011.
18. John C. Reynolds. User-defined types and procedural data structures as complementary approaches to data abstraction. In *New Advances in Algorithmic Languages*, pages 157–168, 1975.
19. Romain Robbes and Mircea Lungu. A study of ripple effects in software ecosystems. In *Proceedings of ICSE 2011, NIER Track*, pages 904–907, Honolulu, Hawaii, USA, May 2011. ACM Press.
20. Filip Van Rysselberghe and Serge Demeyer. Studying versioning information to understand inheritance hierarchy changes. In *Proceedings of MSR 2007*, page 16, 2007.
21. Niko Schwarz, Mircea Lungu, and Romain Robbes. On how often code is cloned across repositories. In *Proceedings of ICSE 2012, NIER Track*, 2012.
22. Alan Shalloway and James R. Trott. *Design Patterns Explained: A New Perspective on Object-Oriented Design*. Addison-Wesley, 2004. 2nd edition.
23. Ewan D. Tempero, James Noble, and Hayden Melton. How do java programs use inheritance? an empirical study of inheritance in java software. In *Proceedings of ECOOP 2008*, pages 667–691, 2008.
24. Mads Torgersen. The expression problem revisited (four new solutions using generics). In Martin Odersky, editor, *Proceedings of ECOOP 2004*, number 3086 in LNCS, pages 123–146, Oslo, Norway, June 2004. Springer-Verlag.
25. A. Vargha and H. D. Delaney. A critique and improvement of the cl common language effect size statistics of McGraw and Wong. *Journal of Educational and Behavioral Statistics*, 25(2):101–132, 2000.
26. Philip Wadler. The expression problem. Mail to the java-genericity mailing list, 1998.
27. Matthias Zenger and Martin Odersky. Independently extensible solutions to the expression problem. In *Proceedings of FOOL 2005*, Long Beach, USA, January 2005.
28. Thomas Zimmermann, Peter Weißgerber, Stephan Diehl, and Andreas Zeller. Mining version histories to guide software changes. *IEEE Transactions on Software Engineering*, 31(6):429–445, 2005.