Empir Software Eng (2015) 20:745–782
DOI 10.1007/s10664-013-9298-0

Object-oriented software extensions in practice

Romain Robbes · David Róthlisberger · Éric Tanter

Published online: 29 January 2014

Abstract As software evolves, data types have to be extended, possibly with new data variants or new operations. Object-oriented design is well-known to support data extensions well. In fact, most popular books showcase data extensions to illustrate how objects adequately support software evolution. Conversely, operation extensions are typically better supported by a functional design. 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 130,000 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; the viability of the Visitor pattern as an object-oriented solution to operation extensions; the change-proneness of extensions; and the prevalence of extensions by third parties. 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.

Keywords Object-oriented programming · Software evolution · Data extensions · Operation extensions · Empirical studies · Mining software repositories

Communicated by: Arie van Deursen

R. Robbes (✉) · É. Tanter
PLEIAD Laboratory, Computer Science Department (DCC), University of Chile, Santiago, Chile
e-mail: rrobbes@dcc.uchile.cl

É. Tanter
e-mail: etanter@dcc.uchile.cl

D. Röthlisberger
School of Informatics and Telecommunications, Universidad Diego Portales, Santiago, Chile
e-mail: roethlis@mail.udp.cl

1 Introduction

Lehman's laws of software evolution (Lehman and Belady 1985) 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 (Krishnamurthi et al. 1998)—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 (Reynolds 1975) and has been described by other researchers since then (e.g., Cook 1990; Krishnamurthi et al. 1998; Wadler 1998). 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 more complicated than using an appropriate tool, such as a brush. The object-oriented programming community has in fact designed a solution to handle operation extensions, called the Visitor design pattern (Gamma et al. 1994). 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, since a new data type implies the modification of all existing visitor classes. Many programming language constructs have been proposed in order to support both dimensions of extension in a modular (and type safe) manner (e.g. Krishnamurthi et al. 1998; Oliveira 2009; Torgersen 2004; Zenger and Odersky 2005).

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 1994 and Shalloway Trott 2004 for two popular books). Meyer also claims that "One of the most frequent forms of extension to a system will be the addition of new types [...] This is where object technology shines in its full glory" (Meyer 2009). However, there is no empirical data on how frequently data 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 investigated 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 (Wadler 1998; Torgersen 2004). This work is concerned with both viewpoints, and studies the evolution of open-source object-oriented projects through their commit history. For the first perspective, we look 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 (Erlikh 2000). For the third-party extension viewpoint, we also look at how programmers extend external libraries, contrasting the findings of both settings.

Concretely, we seek to answer the following research questions:[1]

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, as is commonly said in the literature, object-oriented programming is really superior in dealing with extensible software, we would expect object-oriented projects to showcase a greater number of data extension cases. Is it really the case? Or conversely, are there much more operation extensions, suggesting that another programming abstraction 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?

Q5: How stable are extensions over time? Once an extension is performed, is it frequent to update its definition? More precisely, for operation extensions, are the various implementations of an operation uniformly subject to changes, thereby making their non-modular implementation problematic in practice?

Q6: Are third-party extensions special? If we only consider extensions by third-party—the expression problem stricto sensu—do we observe similar prevalence results with respect to both kinds of extensions? How much is a mechanism like open classes (known in Smalltalk as "class extensions") really used to perform third-party operation extensions?

Why Study Smalltalk? By observing the evolution of a large number of open source Smalltalk projects, this paper presents elements of answers to these questions. We chose to study Smalltalk for two reasons. First, Smalltalk is one of the languages that supports third-party extensions, allowing us to investigate our last research question. Second, we did previous work on the same corpus (Callaú et al. 2012; Robbes et al. 2012a) allowing us to reuse our expertise about the systems we investigate, the data we gathered, and part of the tool support we implemented.

Smalltalk is a dynamically-typed object-oriented language, like JavaScript, Python and Ruby, which are increasingly popular. Therefore, this study does not answer the research questions above 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 object oriented, we get to observe how object programmers actually benefit (or not) from working in that paradigm. Studies based on other languages, including

[1]This paper extends our previous conference publication (Robbes et al. 2012b) with two new research questions, one related to the stability of extensions (Q5) and the other related to the analysis of third-party extensions (Q6).

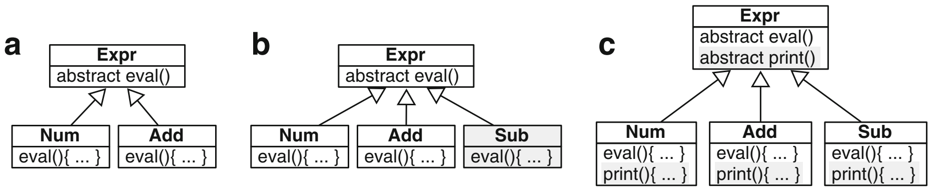

Fig. 1 A class hierarchy (**a**) and two extensions: data extension (**b**), and operation extension (**c**). Changes are highlighted in *gray*

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 (Cook 2009).

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 six sections report our findings related to the six research questions stated above. Section 10 discusses threats to the validity of this study, and Section 11 concludes.

2 Background and Related Work

We first explain the different kinds of extensions and how to deal with them 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 (Fig. 1a).[2] 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 (Fig. 1b). Note how the object paradigm makes this extension localized: it 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 (Fig. 1c).

One could also decide to implement the different variants of the operation in separate objects (e.g. `NumPrinter`, `AddPrinter`). The superclass `Expr` can declare a `Printer` field and implement `print` so that it delegates to these objects. In terms of modularity, each subclass still needs to initialize the printer field adequately, therefore the scattering related to the operation remains.

Finally, in certain scenarios, the new operation is independent of the particular subclass, in which case it can be defined as a single method in the superclass. It is then inherited in all subclasses, without having to modify them. Such an extension is therefore implementable

[2]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.

modularly with an object-oriented decomposition; but it would also be implementable modularly with a functional decomposition. For this reason, we do not focus on such cases in this paper.

The Visitor design pattern (Gamma et al. 1994) is the standard way to handle operation extensions in a modular manner in object-oriented programming. It is illustrated on Fig. 2. 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 (Fig. 2a). A separate hierarchy of visitors is then defined, each for its own operation (e.g. `EvalV` extends from `ExprVisitor`). Adding a new operation on the hierarchy is now expressed as adding a new visitor subclass, e.g. `PrintV` (Fig. 2b). 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. `Sub`) implies extending all the visitors with new `visit` methods (Fig. 2c).

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. (Gîrba et al. 2005). 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 (Van Rysselberghe and Demeyer 2007). 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 (Baxter et al. 2006). 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 (Tempero et al. 2008); 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

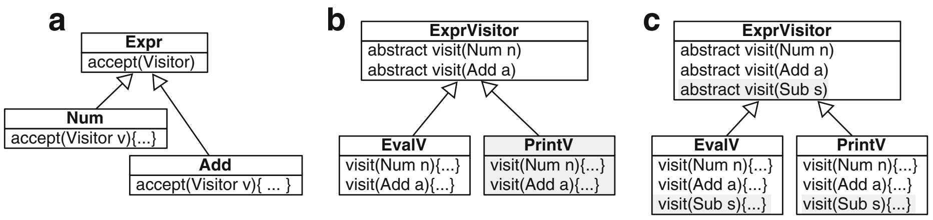

Fig. 2 A class hierarchy prepared to accept visitors (**a**). The hierarchy of visitors is extended with a visitor for printing (**b**). Adding a new data type implies modifying all the visitor hierarchy (**c**). Changes are highlighted in gray

large-scale survey of programmers by Gorschek et al. (2010) 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 (Grechanik et al. 2010). 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.

Parnin et al. (2012) investigated how 20 open-source Java projects adopted the Java Generics, and found that there was less Generics usage than they expected, and that most cases were due to type-safe collection traversals.

Finally, Aversano et al. studied the evolution of several design patterns, including the Visitor pattern, on three software systems (Aversano et al. 2007). 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

Squeaksource We analyze a large extract of the *Squeaksource*[3] 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 130,000 changes to Squeaksource. Figure 3 shows an evolution plot of the Squeaksource repository illustrating how the number of committed versions grows from 2003 to July 2011, marking the end of the period we analyze in this paper. The graph shows sustained activity for the entire period, with thousands of monthly commits in the later periods.

Monticello 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 each commit, meta-information is recorded. Monticello has the particularity that it is a language-aware version control system: it was designed for the specific purpose of versioning Smalltalk code. As such, Monticello versions code at the level of packages, classes, and methods, not at the level of files and lines of code. This allows the analyses (differencing, merging, conflict detection, etc) to

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

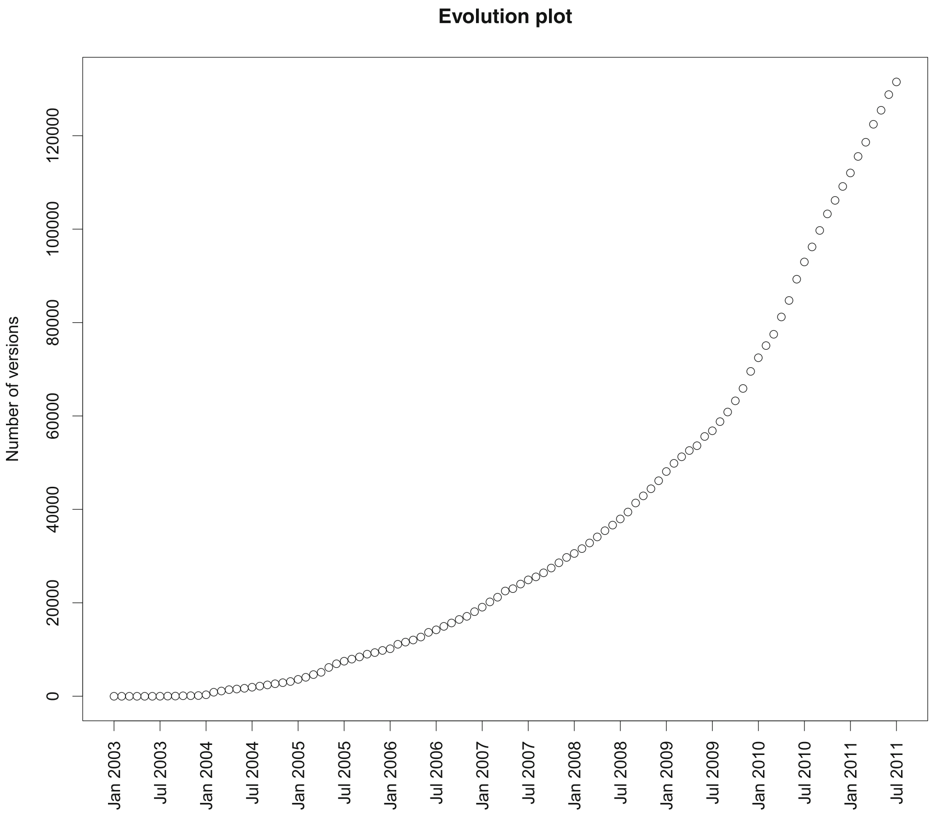

Fig. 3 The evolution of the Squeaksource repository from January 2003 to July 2011 in terms of number of versions contributed

be done at the level of methods and classes, instead of simply lines of code, providing a clearer view of the conflicts. A further interesting aspect of Monticello is that is supports class extensions: a method can be defined and versioned in another package than the class it belongs to. This feature allows a package to extend the behavior of an existing class; we will inspect that particular feature in Section 9. All these advantages do not come for free: a language-aware versioning system such as Monticello can not be used to version resources beyond source code (e.g. documentation, graphics, etc) (Robbes and Lanza 2005).

Ecco To actually analyze the Squeaksource repository we use the Ecco model (Robbes and Lungu 2011). 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; the model allows multiple links between ancestors and successors to accommodate forks and merge operation.

Ecco allows us to effectively and efficiently process and analyze the large set of changes (approximately. 13GB of compressed source code) we extracted from Squeaksource. Ecco

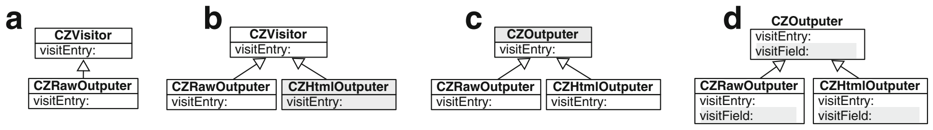

Fig. 4 The Citezen visitor hierarchy: **a** initial version, **b** after a data extension, **c** after renaming the root class, and **d** after an operation extension

acts as a front-end for Monticello, abstracting its functionality to the level of the analysis. Each package version in Ecco is associated to a Monticello package version; this allows us to get detail on demand about the projects in the system when it is deemed necessary, accessing Monticello's APIs for that. Likewise, we use Monticello's difference engine to extract the changes between the versions of the system; the actual change analysis specific to this study is described in the next section.

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 Fig. 4, 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[4] (shown in Fig. 4b 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. Fig. 4d, 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 same hierarchy, this single method addition is also considered to be part of an operation extension. Additions of methods being unique in the hierarchy in current, previous and subsequent versions are not considered in this study because such additions are local by definition and hence do not impose a problem from an extensibility point of view.

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 Fig. 4c). In Monticello, renaming an

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

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 While simply counting the number of extensions gives us an idea of their relative frequency, we want to study the phenomenon further. We also weight the extensions by the number of methods involved in them, in order to have an approximation of the size of each kind of extensions.

The reason we measure the number of methods, and not for instance the number of lines of code, is that the difference between the two kinds of extension is one of modularity. In the case of a data extension, the set of methods implementing the new behavior is added in a single module. In the case of an operation extension however, the set of methods is *scattered* over a set of modules. The complexity of each individual method would be very similar in each decomposition, but the factor that varies is their location in the system. Adding an operation as a set of methods in several classes implies identifying the classes that need to change, and understand each changing class enough to add the new method. One could argue that such a process requires more effort due to the multiple context switches involved. However, this is hard to measure. In the absence of that, we believe the number of methods contained in the extension is a good indicator of both the size and the scattered-ness of the extension, and use that metric in the rest of the paper.

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* (Posnett et al. 2011). 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. The threshold value of 50 additions of classes and methods is motivated by the fact that the distribution of commit size has an inflection point near 50 additions, meaning that the number of commits per number of additions is from a peak at one addition rapidly decreasing to 50 additions per commit while from there on its slowly and steadily decreasing, hence only outlier commits include more than 50 additions. (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 (Zimmermann et al. 2005).

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. The selected thresholds approximately represent the third quartile of the distributions, that is, only 25 % of all hierarchies have more than five classes while only 25 % of the projects have more then 45 classes. To be in sync with the threshold of 50 additions for large commits, we have opted to use a 50 classes threshold for large projects instead of the 45 classes representing the third quartile.

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 (Fig. 5): We start with 131,544 commits; of those, 13,148 commits are classified as *large* or *initial commits*, leaving us with 118,396 *selected* commits. The 95,662 classes are organized in 48,595 hierarchies. Of those, 20,045 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. (2008).

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 6 shows the distribution of our metrics of interest across projects and class hierarchies, applying a logarithmic scale on the y-axis to account for the relative large number of hierarchies and projects with no extensions. None of the distribution follows the characteristic “bell shape” of a normal distribution. Instead, they seem to follow exponential or power-law distributions, where the overwhelming majority of observations has very low metric values, and a minority has high values. Previous work confirmed that

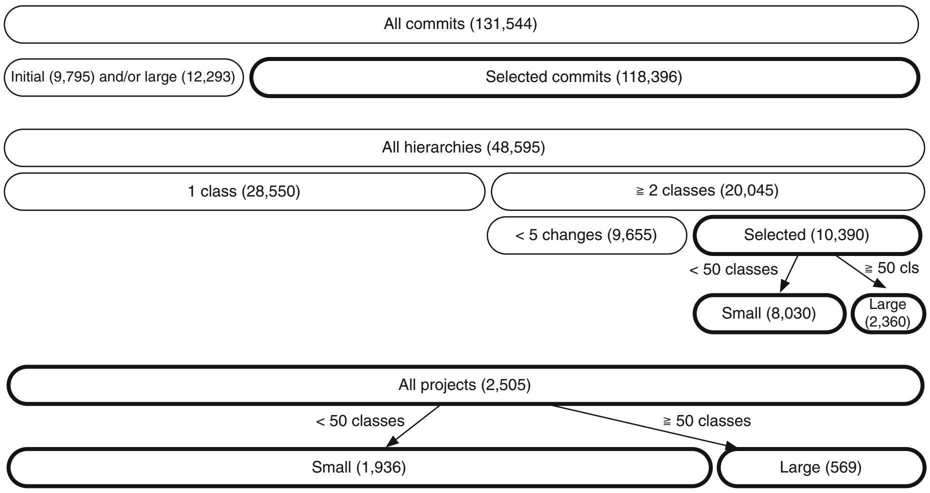

Fig. 5 Diagram showing number of selected commits, hierarchies and projects after filters have been applied

power laws are common in software, be it in Java source code (Baxter et al. 2006), or even in a variety of metrics, including source code metrics, but also dependencies between libraries (Louridas et al. 2008). 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 either boxplots (showing percentiles) or violin plots as a visual summary of the distributions.

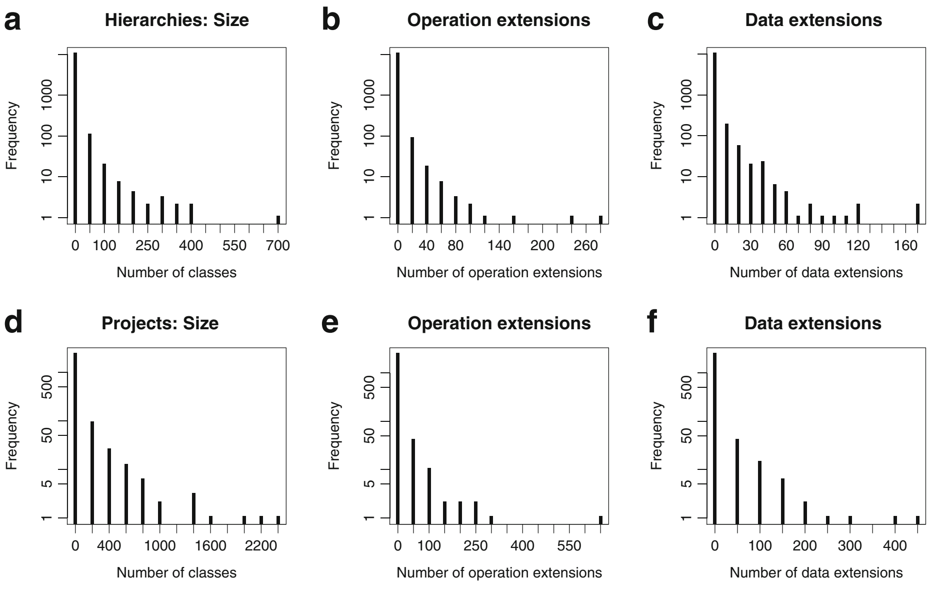

Fig. 6 Histograms showing the distribution of size and extension metrics across hierarchies and projects. Note that a logarithmic scale is used for the frequency (y-axis)

Statistical Tests For the analysis of the data, for instance regarding differences in distributions of the two kinds of extensions, we cannot rely on parametric tests as the distribution of the data strongly departs from normality and thus breaks the assumptions of most parametric tests. However, given the large size of our sample, a non-parametric test such as the Mann-Whitney U-test would almost certainly find a statistically significant 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, i.e. whether the difference is practically significant. 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 $\hat{A}_{12}$ (Vargha and Delaney 2000). 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 (Arcuri and Briand 2011). $\hat{A}_{12}$ 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. We take use of the $\hat{A}_{12}$ effect size measure in Section 5.2 when analyzing the difference between data and operation extensions.

To quantify relationships in the data between different variables, for instance between size of hierarchies and number of extensions (e.g. in Section 5.3), we compute the Spearman correlation. 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.

To ensure the replicability of this study, we make publicly available all data extracted from Squeaksource in a comma-separated format along with all R scripts we developed to visualize and statistically analyze this data, in order to give other researchers the possibility to reproduce and extend our study. All necessary files are contained in this archive: http://tinyurl.com/p6usq9c.

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 7 shows the proportion of commits featuring operation and data extensions versus commits that feature neither of these.[5] Of the 131,544 commits we

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

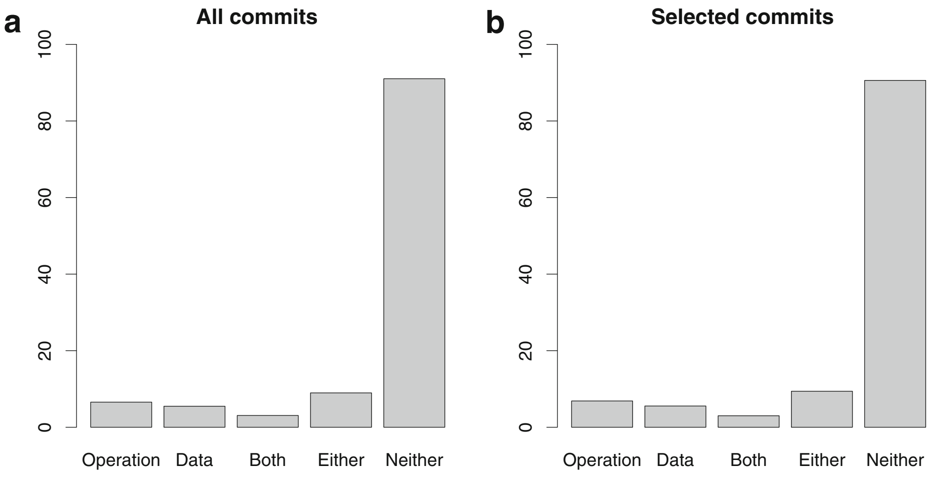

Fig. 7 Percentages of commits featuring extensions. **a** all commits; **b** selected commits

analyzed, 11,802 (8.97 %) feature either an operation or a data extension (commits with data extensions: 5.49 %; commits with operation extensions: 6.56 %). If we only consider selected commits (that is, we filter out both large and initial commits), the proportion rises to 9.41 % (data extension: 5.54 %; operation extensions: 6.86 %). This means that in nearly one out of 10 commit, developers perform either a data or an operation extension in the system they work on.

Operation extensions are potentially more problematic than data extensions, at least they are considered to not align with the "clean case" of extending object-oriented software. We can see that operation extensions occur in 6.56 % of all commits (6.87 % 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 8 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. As expected and highlighted in Fig. 9 the number of extensions increase with the size of the hierarchy, for both data and operation extensions. Computing the Spearman correlation measure reveals a fairly strong

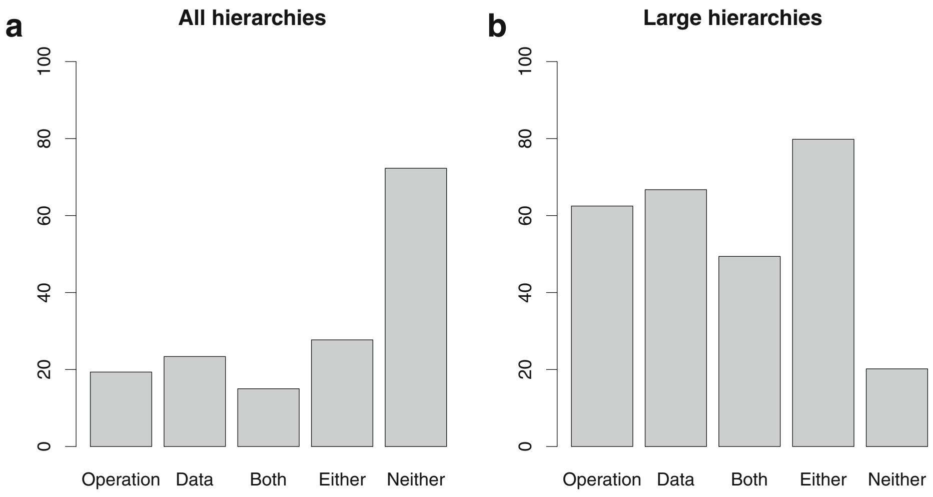

Fig. 8 Percentages of hierarchies featuring extensions. **a** all hierarchies; **b** large hierarchies only (5 or more classes)

correlation between number of extensions and size of hierarchy: $\rho = 0.48$ for data and $\rho = 0.55$ for operation extensions.

Figure 10 highlights the percentages of projects featuring extensions. In brief, 41 % of all projects feature extensions (data extensions: 37.16 %; operation extensions: 33.35 %; both: 29.06 %). A very large majority (84.02 %) of large projects (i.e. projects with more than 50 classes) have to deal with extensions (data extensions: 79.20 %; operation extensions: 78 %; both: 73.19 %). The larger a project the more extensions it features as depicted in Fig. 11: The number of both data and operation extensions steadily grows with the number of classes in a project. The Spearman correlation between number of extensions and number of classes in a project is strong: $\rho = 0.60$ for data and $\rho = 0.61$ for operation extensions.

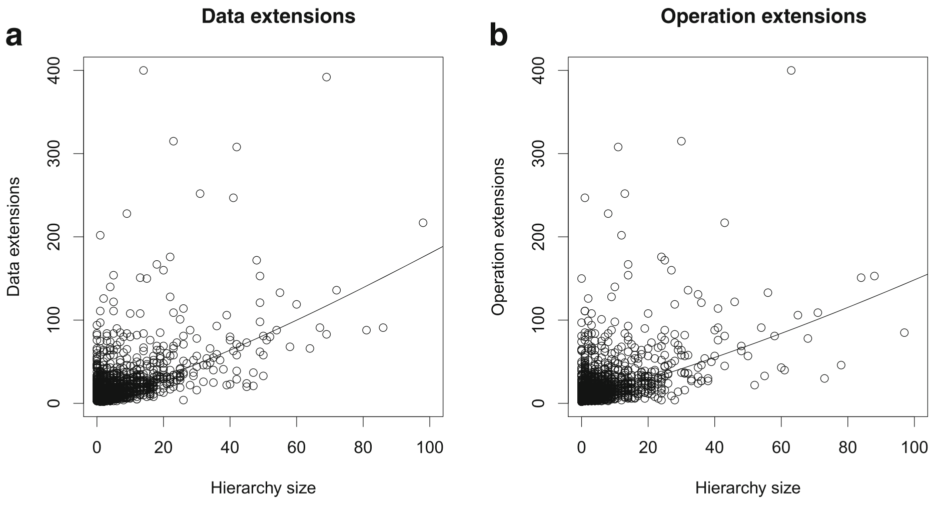

Fig. 9 Effect of size of hierarchy on number of extensions. **a** data extensions; **b** operation extensions

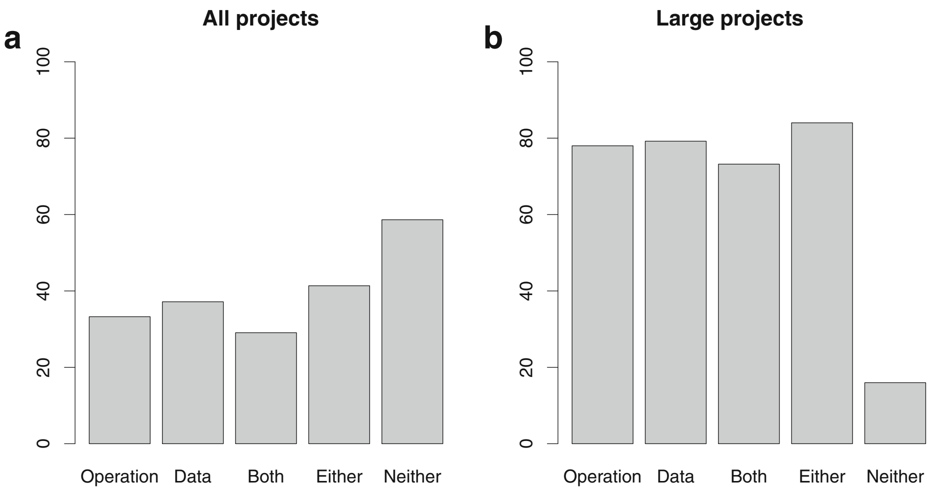

Fig. 10 Percentages of projects featuring extensions. **a** all projects; **b** large projects only (50 or more classes)

4.3 Summary

Extensions regularly occur in practice: one out of ten commits (9.41 %) 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.

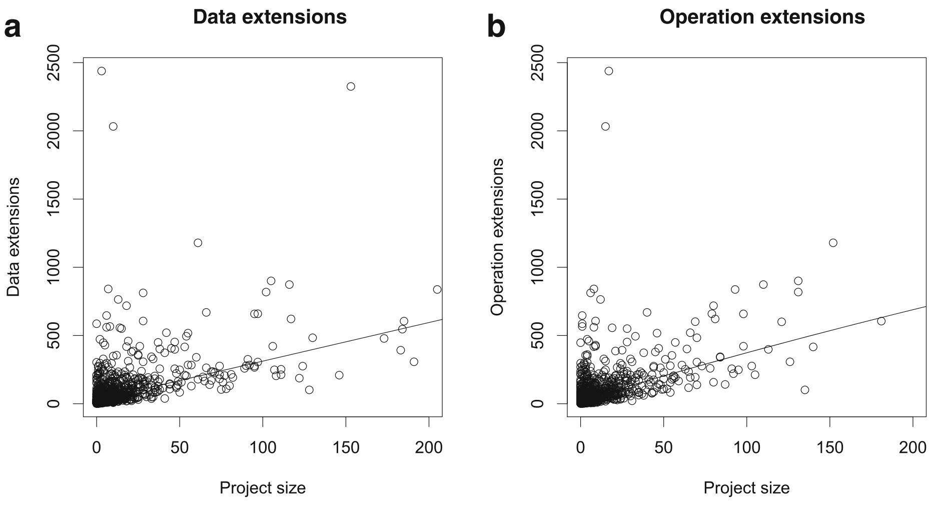

Fig. 11 Effect of project size on number of extensions. **a** data extensions; **b** operation extensions

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, present in the literature, 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 Figs. 7 and 8, we notice that both types of extensions happen with a somewhat similar frequency (i.e. 5.5 to 6.5 % of all commits, 60–65 % of large hierarchies, etc.) and routinely overlap.[6] 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.

Figure 12 shows the distributions of both kinds of events as box-and-whiskers plots, for both unweighted—to evaluate frequency—and weighted—to evaluate scattering and size—distributions, for the 28 % of hierarchies that feature at least one of the two events during their life time.

If only frequency is considered (Fig. 12a), we see that hierarchies featuring extensions have an identical median value of two extensions for both kinds of extensions. 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.

In the weighted case, (Fig. 12b), a different picture emerges. The median number of methods to introduce a data extension is higher (7) than the number of methods to introduce an operation extensions (3). However, the boxes still overlap significantly: the 75th percentile of operation extensions is higher than the median of data extensions. If operation extensions are larger, the difference is not so high that one can ignore operation extensions altogether. Further, the number of methods involved in operation extensions shows that some of the operations are very scattered.

Figures 13 and 14 show the distributions of the commits and projects, respectively, that contain at least one kind of extension. The box plots illustrate that the commit and project distributions share with the distribution of extensions for hierarchies the same pat-

[6] 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.

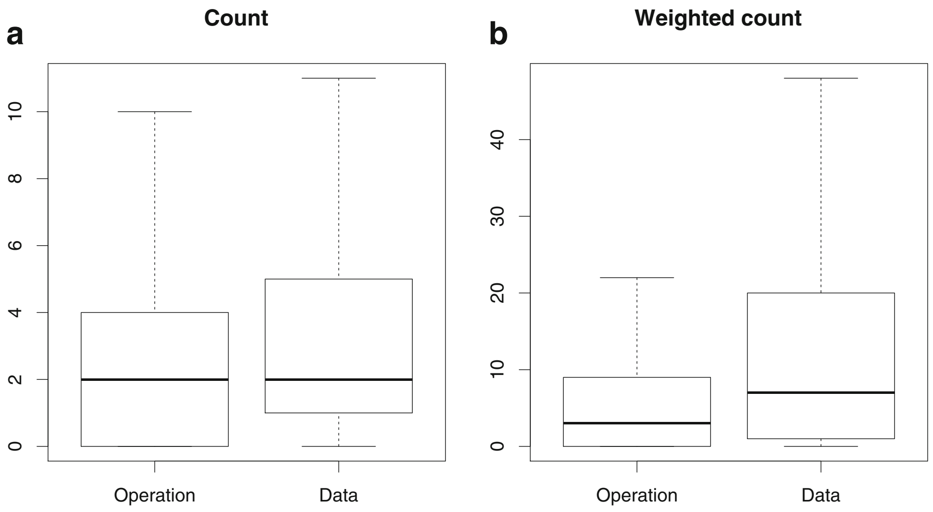

Fig. 12 Boxplots of distribution of extensions for hierarchies featuring them. **a** unweighted; **b** weighted

tern 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 1, while the weighted operation extension's 3rd quartile is 4; 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

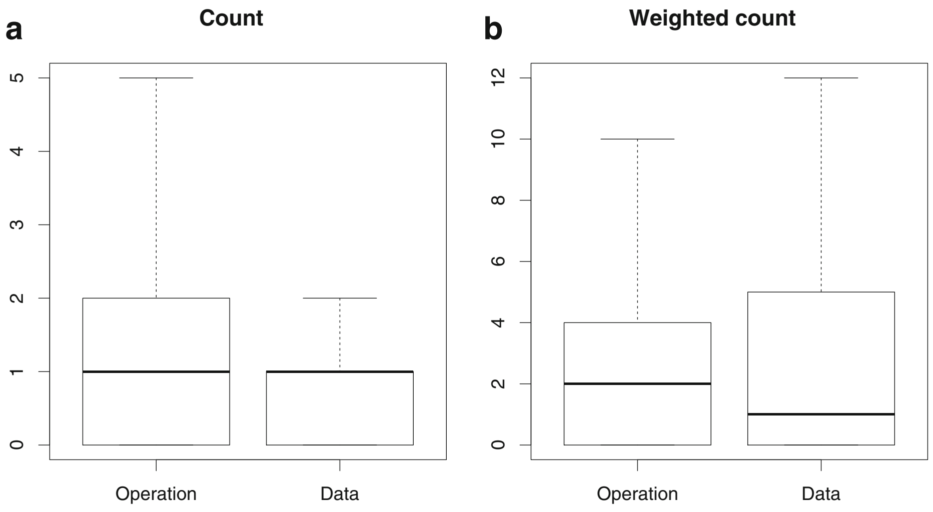

Fig. 13 Boxplots of distribution of extensions for commits featuring them. **a** unweighted; **b** weighted

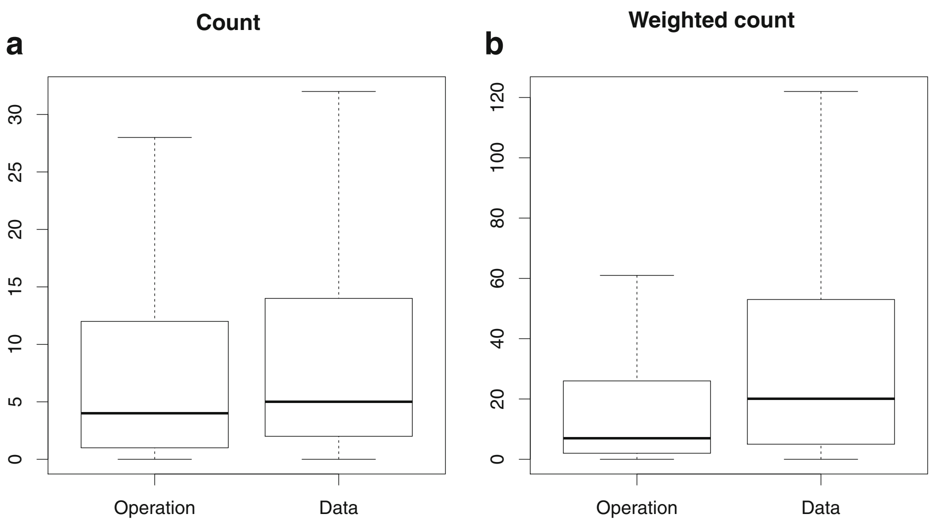

Fig. 14 Boxplots of distribution of extensions for projects featuring them. **a** unweighted; **b** weighted

when weighting is applied. In this section, we seek to quantify the difference by computing the effect size measure $\hat{A}_{12}$ as described in Section 3.3.

In the case of unweighted frequencies of both kinds of extensions, we obtain for the effect size 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.[7]

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

[7]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.

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 Fig. 15 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 (see Section 3.3) between the number of extensions and the size of the hierarchies. Note that all the correlations below are highly statistically significant: in all cases, $p \lll 0.01$.

We start with both kinds of extensions taken together (Fig. 15a). 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 15b 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 Fig. 15c. 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

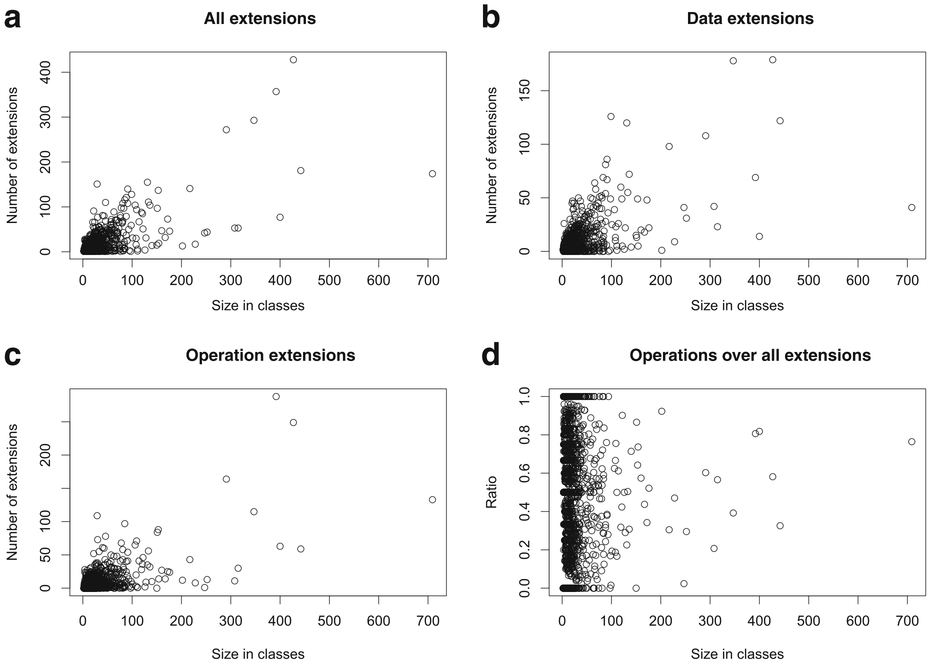

Fig. 15 Effect of size of hierarchies on kinds of extensions

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 Fig. 15d 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 Summary

Analyzing the frequency and the number of methods present 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 (Lehman and Belady 1985) 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.[8]

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 16 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.

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 Summary

If we consider a high ratio of operation extensions as a sign for developers frequently not following the easier case of object-oriented software development, these results confirm Lehman's observations 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.

[8]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.

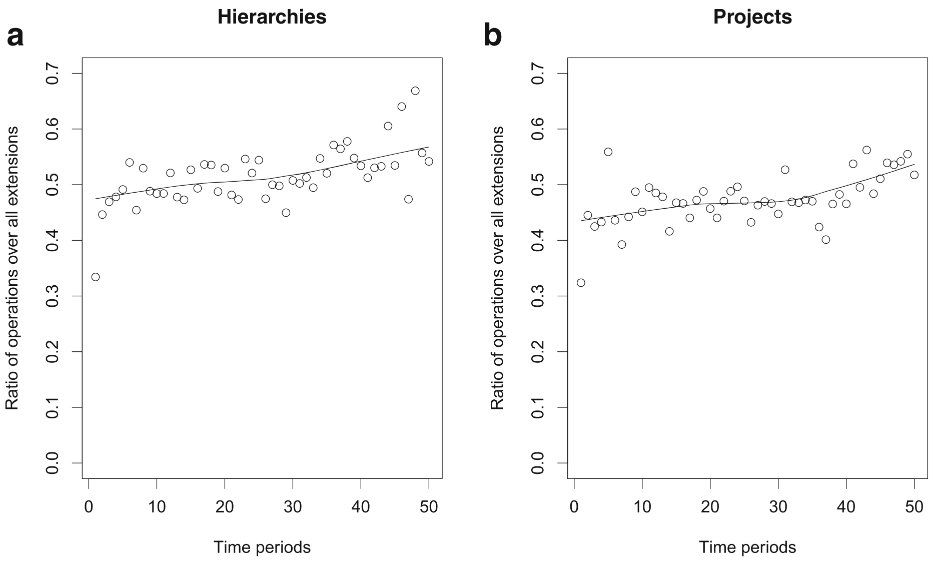

Fig. 16 Proportion of operation extensions among all extensions over time. **a** hierarchies; **b** projects

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 (Gamma et al. 1994), 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

Our visitor detection algorithm (Section 3.2) reveals that a minority of classes are involved as either visitors or visitees. Out of the 2,879 hierarchies that experienced at least an extension, 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 %.[9]

[9]The discrepancy in number of hierarchies is because there may not be a one-to-one mapping between visitors and visited hierarchies.

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 (Gamma et al. 1994). This means that visited hierarchies should feature less data extensions, and the corresponding visitor hierarchies should undergo less operation extensions.

Figure 17 shows the distribution of extension metrics, normalized by hierarchy size measured in number of classes, contrasting normal hierarchies, visitor hierarchies, visited hierarchies, and the last two kinds of hierarchies taken together. First, in Fig. 17a, visitors and visited hierarchies seem to exhibit less data extensions than normal hierarchies, in proportion to the size of the hierarchy. This is particularly noticeable in for visitor hierarchies, where the difference is statistically significant ($p < 0.02$). Second, in Fig. 17b, we see that both visitor and visited hierarchies seem to feature around the same number of extensions than normal hierarchies.

More operation extensions, while visitor hierarchies take around the same number of operation extensions than normal hierarchies.

Besides the statistically significant difference in data extensions between visitor and normal hierarchies, all other relationships were found to not be significant (with p-values in the range of 0.2 to 0.4). This makes classes involved in the Visitor pattern no worse, but also

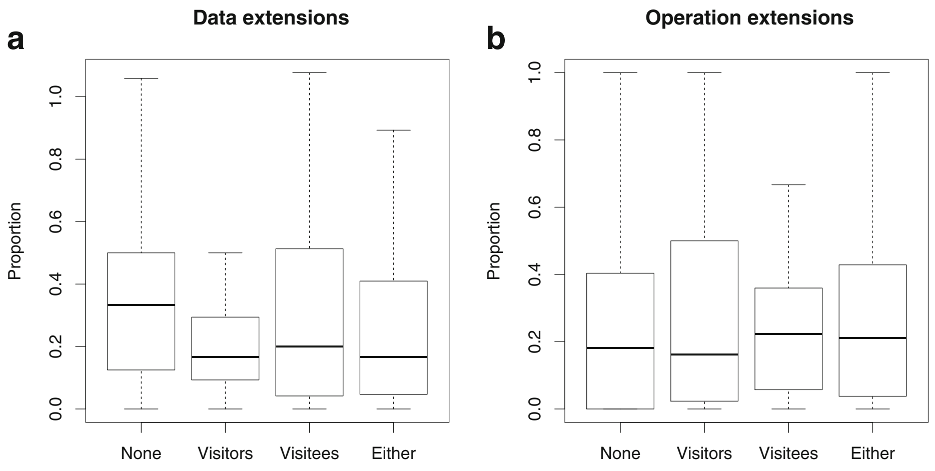

Fig. 17 Distribution of data and operation extensions by role in the Visitor pattern

no better, than regular classes, for both roles and both metrics, except for visitor hierarchies and data extensions. In that particular case, we found that visitor hierarchies had comparatively less data extensions than normal hierarchies, meaning that comparatively less visitors were added after the initial introduction of the hierarchy.

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 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 (Aversano et al. 2007).

8 How Stable are Extensions?

The modular implementation of stable extensions is less crucial than that of unstable ones: occasional crosscutting changes are less harmful than frequent ones. Furthermore, for unstable extensions, the distribution of the changes across source code locations—also known as *entropy*—matters: if the location that changes is predictable, it is less problematic than if changes can potentially occur anywhere.

8.1 Data Processing for Extension Stability

To investigate the stability of extensions over time, we extend our analysis to take into account *changes* to extensions, instead of only looking at the additions of new extensions as we did before.

We define the *change proneness* of a given method as the probability of it changing in a revision since it is present in the system; more formally, the change proneness is the ratio of the number of times the method was changed, over the number of times the method was present in a revision (excluding the revision when it was added to the system). This gives us a value ranging from 0 for a method that has been introduced but never changed, to 1 for a method that has been introduced and then changed in every subsequent revision. We restrict the analysis to methods that were present in at least 5 versions.

Note that unlike in the previous analysis, we do not apply any filtering on the type of commits: the large commits and initial version commits that were filtered previously contained a large number of additions that were constituting noise, but this is not true for the modifications. Each method is then tagged by how it was introduced in the ecosystem, either as a method introduced as part of a *data* extension, as an *operation* extension, or as part of the *normal* development process. In some cases a method can be added several times; if it is added as both a data and an operation extension, it is counted as both.

8.2 Change Proneness by Kinds of Extensions

Figure 18 shows a violin plot for the distribution of the change proneness of all the methods, according to their category of introduction. The violin plot also shows the frequency of values in a distribution on the horizontal axis. The wider the "violin", the more data points with the same value exist. The white dot represents the median, the black box the interquartile range (50 % of the values are between the 25th and the 75th percentile), similar as in a box plot. The vertical lines leaving the black box depict the 5th and the 95th percentile, respectively, like the whiskers in a box plot. In Fig. 18 we can see that all the distributions have essentially the same shape, with a large majority of methods having a low change proneness, and few outliers that have a higher change proneness. The medians for each distribution are: 2.9 % (operation), 3.3 % (data), and 4.8 % (normal), with the first quartiles at 1.2 %, 1.2 %, and 1.7 %, and the third quartiles at 7.9, 9.1, and 13.2 %.

As such the data shows that methods introduced via operation and data extensions are actually somewhat less prone to change than other methods. The figure also seems to indicate that methods introduced via operation extensions are slightly less change-prone than methods introduced via data extensions.

The Wilcoxon rank-sum tests find significant differences between all the distributions (with $p < 10^{-4}$ in all cases), however some of it is due to the large sample sizes; computing the $\hat{A}_{12}$ effect sizes shows that the differences between methods introduced by data and operation extensions are trivial ($\hat{A}_{12} = 0.514$, for an equivalent $d = 0.04$). On the other hand, differences between normal and data extensions ($\hat{A}_{12} = 0.571, d = 0.19$), and normal and operation extensions ($\hat{A}_{12} = 0.586$, $d = 0.23$), are small, but genuine. This reflects what is visible in the figure.

To conclude, methods introduced by each kind of extension are slightly less change-prone than methods introduced by regular software development. However, the difference is small, and the two kinds of extensions are virtually indistinguishable.

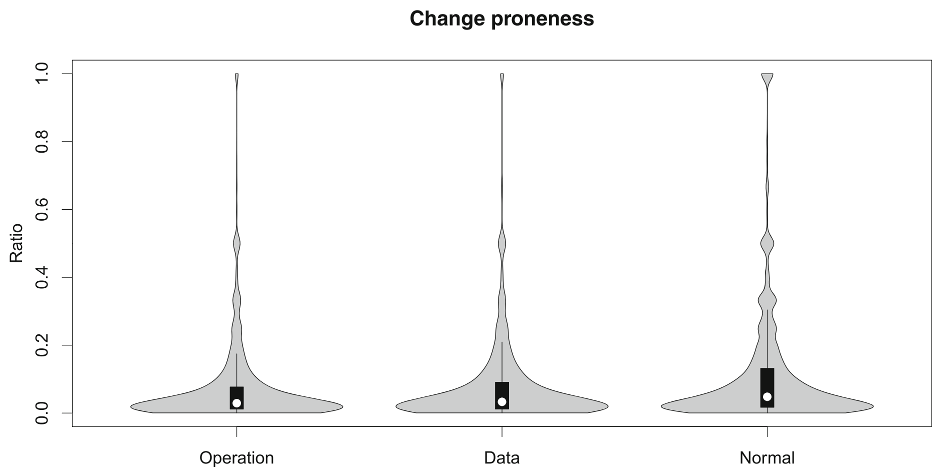

Fig. 18 Distribution of change proneness of operation and data extensions compared to normal methods

8.3 Entropy of Operation Extensions

In order to understand the impact of non-modular extensions on software maintenance scenarios, it is informative to study how the changes are distributed across the multiple implementations of the same message.[10] In object-oriented terminology, methods are invoked by sending messages to objects. The name of the invoked method is equal to the message sent (e.g. `visitField:` in Section 3.2 is both the name of message and method). Each message (or operation) can be implemented by several methods with that name but in different classes. The concretely invoked method is depending on the object receiving the message send at runtime.

Indeed, if the changes are consistently applied to a single implementor of a message, then the situation is arguably better than if the changes are equally likely to be introduced to all implementors of a message. In the latter scenario, developers are much more likely to be forced to inspect all the implementors of a given message in the case it has to be modified.

More precisely, we use the concept of *normalized entropy* as defined by Shannon (1948) and used in defect prediction by Hassan (2009). The entropy characterizes the predictability of the distribution of values across categories, so that the entropy is lowest (0) when all the values belong to one of the categories, and highest (1) when the values are equally distributed over the possible categories. More formally, for a probability distribution p, where $\sum_{k=1}^{n} p_k = 1$, the normalized entropy is defined as:

$$entropy = -\sum_{k=1}^{n} p_k log_2(p_k) \tag{1}$$

In our case, we compute the entropy of a given polymorphic message based on the change proneness of the methods that implement the message. The change proneness of message `m` will be 0, for instance, if it has several implementations in different classes, but only one of them ever changes. It will be at its maximum value of 1 when the probability that each implementor changes is equal. We performed this process for all the messages that were introduced via operation extensions.

Figure 19a shows the distribution of the entropy of the change proneness of all the implementors of messages added as operation extensions. What we can see is the shape of a bimodal distribution, with a peak at the minimum entropy of 0, and another at the maximum entropy of 1, with a lower density between the two extremes. The peak at 0 is larger (indeed the median is at 0), which is good news at first glance: the majority of operation extensions are changed constantly, rather than unpredictably. However, the second peak is accompanied with the 3rd quartile (which is at 0.91), which means that 25 % of all messages have an entropy higher than 0.9. This needs to be inspected more closely, as a significant minority of message implementations are changing in an unpredictable manner.

To have a clearer picture of these high-entropy messages, we set up finer-grained classifications of the operation extensions, first according to their change proneness—Fig. 19b, c, and d—, and then according to the number of implementors of the message—Fig. 19e, f, and g.

[10]Note that because data extensions are by definition modular in an object-oriented decomposition, it is unnecessary to study the distribution of their changes.

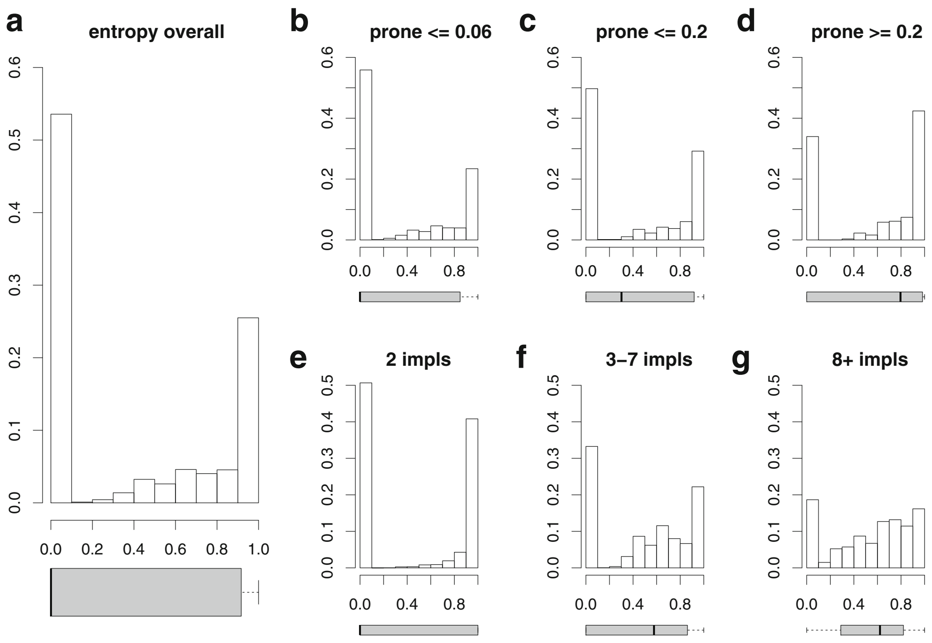

Fig. 19 Distribution of entropy of changes performed to operation extensions (x-axis: entropy; y-axis: percentage). **a** Overall, **b–d** depending on change proneness of the operation extension, **e–g** depending on number of implementors of the extension

We decided to define bins for both change proneness and number of implementors (low, median and high proneness, few, median and many implementors) because when we examined the scatterplots of extension entropy and these metrics, we realized that their relationships are not linear which makes an overall analysis regarding the correlation of proneness/implementors and entropy unsuited. Hence, we investigated variations of the distributions across different bins. We determined the thresholds separating the bins based on the distributions of the change proneness and number of implementors, noting in particular the presence of early peaks and long tails in the distributions. In each case, we used two thresholds: one for the majority of values that is situated before the peak (change proneness of 0.06 in one case, 2 implementors in the other), and one separating the values after the peak and the extreme values at the end of the tail (change proneness of 0.2 in one case, and 7 implementors in the other).

Change Proneness The rationale for the classification based on change proneness is that if the messages that change in an unpredictable manner are the ones that do not change often, then it is less problematic than if the reverse is true. We ordered the operation extensions based on their overall change proneness (i.e. the change proneness of a given message is obtained by averaging the change proneness of each implementor), and divided them in three categories or bins, as described above.

As illustrated in Fig. 19b–d, the results unfortunately indicate that messages who are change-prone (Fig. 19d), have a higher entropy than the other categories; the median entropy among the operation extensions with high change proneness is 0.80, whereas it is 0 in the low case, and 0.30 in the medium case. The third quartile is also higher (0.98, versus 0.92 for medium change proneness, and 0.85 for low change proneness).

We also find statistically significant differences: $\hat{A}_{12}$ for the probability that the entropy of a high change proneness is higher than of a medium change proneness equals to 0.58, while $\hat{A}_{12}$ for high change proneness compared to low change proneness is 0.63, translating to a d of 0.35 and 0.52, or small and medium effects. As such, we find that the overall change proneness of a message has an effect on its overall entropy: messages with a higher change proneness are more likely to have a higher entropy.

Number of Implementors The rationale for classifying changes according to the number of implementors of a message is simple: if the entropy of a message is high, but the numbers of implementors is low, then the programmer has few locations to check before doing a change to a polymorphic message. Conversely, messages with a higher number of implementors with a high entropy are more problematic, since the effort involved is higher. We split the operation extensions in three groups, as described above and obtain the results depicted in Fig. 19e–g.

We see a similar behavior as with change proneness: as we progress among the categories of number of implementors, there is a tendency towards higher entropy. More specifically, the distribution when there are two implementors only is strongly bimodal, with either very high or very low entropy values, whereas for the other categories, the entropy values are much more spread out. The median message in the low category (2 implementors) has an entropy of 0; it rises to 0.58 for the medium category, and 0.62 for the high category. As far as effect sizes are concerned, the $\hat{A}_{12}$ metric is of 0.59 between low and medium number of implementors ($d = 0.43$), and of 0.63 between small and large number of implementors ($d = 0.59$). The difference in entropy between medium and large number of implementors is small, with an $\hat{A}_{12}$ of 0.52 ($d = 0.09$).

8.4 Summary

Summing up all our conclusions on the topic of change proneness, we can conclude that if methods introduced as part of operation extensions are about as prone to change as those introduced as data extensions, and less change-prone than normal methods, this does not mean that their maintenance is necessarily easier.

As a measure of the difficulty of maintaining a set of scattered methods, we chose the entropy measurement, which describes how predictable or unpredictable the location of the changes over time are.

According to our entropy measurement, we find that a relatively large minority of operation extensions have a large entropy in their change proneness, meaning that all implementors of the extension are nearly equally likely to be changed. This leads to unpredictable modifications of these extensions, and potentially crosscutting modifications across several classes.

To make matters worse, we found that the messages with a higher average change proneness were likely to have a higher entropy, i.e., to have less predictable change patterns.

Furthermore, we found that messages with a larger number of implementors, i.e., a larger number of locations that can change, were more likely to have a higher entropy. When modifications are hard to predict, it is very likely that the number of possibilities is large, making the effort to verify all the locations that much harder.

As such, far from altering the conclusions of our previous work, our study of change proneness leads us to conclude that operation extensions can pose a maintenance problem. When they do, the problem is likely to be in the most complex cases; this supports our conclusions that more satisfactory means to deal with operation extensions are needed.

9 The Case of Third-Party Extensions

As we mentioned earlier, there are two points of view from which one can study extensions. The one we have investigated so far refers to *local extensions* made by the implementers of the system themselves while it evolves. The second point of view, also known as the expression problem proper, refers to *third-party extensions*, i.e. extensions to existing libraries. The problem there lies in the fact that modifying third-party code is usually difficult as the code is not accessible, or needs to be recompiled.

Smalltalk is an interesting data point in this context as well, because Smalltalk is one of the rare languages where it is possible to modify third-party code quite easily. Smalltalk supports a mechanism of *class extensions*, that is, methods a project can add to classes of another project or to core classes such as `Object`. This mechanism is also known as open classes (Clifton et al. 2000), or inter-type declarations in AspectJ (Kiczales et al. 2001), and is similar to partial classes in C#, mixins in Ruby or mixin composition of traits in Scala. Methods introduced through class extensions have access to the instance variables of the class, just like other methods. The version control system used by Squeaksource, Monticello, supports class extensions and can easily export them as part of a project. In short, the class extension mechanism of Smalltalk can be used by programmers to define *third-party operation extensions*.

The Smalltalk corpus hence gives us an idea of how many third-party operation extensions would occur in other programming languages, if programmers were given the opportunity to take advantage of them. This allows language designers to make informed decisions about the necessity to incorporate an extension mechanism in their languages as well. As another data point, we contrast third-party operation extensions, available in few languages such as Smalltalk, MultiJava and AspectJ, with *third-party data extensions*—i.e., subclassing a class from an existing hierarchy defined outside of the project—a mechanism that exists in all object-oriented languages.

9.1 Frequency of Third-Party Extensions

To characterize the prevalence of third-party extensions, we first analyze how many commits and projects actually encompass third-party extensions. Out of the 131,544 commits we analyzed, 8.81 % contain third-party operation extensions, 13.08 % third-party data extensions, 3.07 % both, 19.29 % either of the two, and 80.71 % no third-party extensions (see Fig. 20a). If we remove initial commits, and large commits that add more than 50 classes and methods, and limit the analysis to the 118,396 remaining commits, we reveal that still 6.87 % of the smaller commits contain third-party operation extensions, 7.73 % third-party data extensions, 0.87 % both, 13.84 % either of the two, and 86.15 % no third-party extensions, as illustrated in Fig. 20b.

If we compare this data to the one in Fig. 7, we note that the figures are in the same range of frequency. However third-party extensions appear to be slightly more common than local extensions. It appears more common to perform third-party data extensions, but the data shows that large and inital commits do it much more often (hence the drop by half when

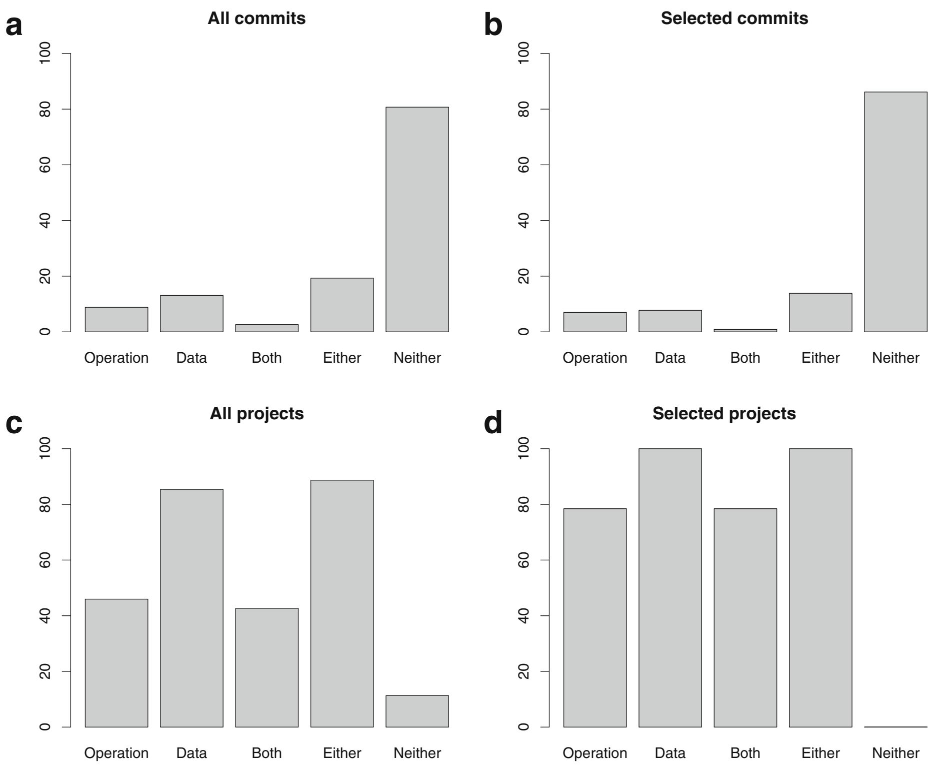

Fig. 20 Percentages of commits featuring third-party operation and data extensions. **a** All commits; **b** selected commits

they are filtered out). As such, we share the same conclusion: third party extensions are not happening extremely frequently, but they are common enough that they can be considered among the concepts that developers are bound to use at some point. This applies to third-party operations in general, but also to the specific case of third-party operation extensions.

Another interesting fact is that there are much less overlap between the kind of extensions: less than 1 % of the selected commits feature both of them.

We now analyze the frequency of third-party extensions in projects.[11] Figure 20c shows that out of all the projects, 45.96 % use third-party operation extensions, 85.40 % make use of third-party data extensions, 42.66 % use both, and 88.69 % use either data or operation extensions (11.31 % of all projects do not use any third-party extensions). When only taking into account large projects, the percentages are even higher: Out of the remaining projects, 78.45 % use third-party operation extensions, and *all* projects use third-party data extensions (see Fig. 20d).

Comparing the prevalence of third-party extensions to local extensions (Section 4), on a project basis, we can see that third-party extensions are considerably more prevalent, as 85.40 % of all projects feature third-party extensions but only 41.2 % local extensions.

[11]Unlike earlier, we are not able to do the analysis at the level of hierarchies, as it is not always clear to which hierarchy a third-party operation extension belongs. This issue is discussed in more details in Section 10.

Hence we can conclude that third-party extensions are more prevalent than operation and data extensions.

If we look at third-party operation extensions in particular, we can see that they are extremely common, especially in larger projects. Third-party operation extensions may be used in only 7 % of commits, but they end up in more than three out of four large projects. We can conclude that Smalltalk programmers do use the possibility of extending library classes when they need it, and tend to need it often.

9.2 Distribution of Third-Party Extensions

In order to characterize third-party extensions more accurately, we investigate beyond the presence/absence of third-party extensions, and examine the size of these extensions. Similarly to the previous sections, we weight the third-party extensions in terms of methods: each third-party operation extension has a weight of 1, while each data extension has a weight equal to the number of methods in the new class.

Extensions in Commits The box plots presented in Fig. 21a illustrate the distribution of third-party operation and third-party data extensions per commit (commits not adding any third-party extension are not included in the box plots). The median number of third-party operation extensions per commit is 2 while there is a median number of 8 third-party data extension per commit (considering the weighting). The first quartile of third-party operation and third-party data extensions are at 1 and 3 extensions, while the third quartiles are at 5 and 23. This shows that when considering the size of the extensions, third-party data extensions are much larger than third-party operation extensions. This finding is unsurprisingly statistically significant, with a large effect size ($\hat{A}_{12} = 0.73$ in favor of data extensions).

Extensions in Projects For the distribution of third-party operation and third-party data extensions per project, we see a similar picture in Fig. 21b (projects without any third-party

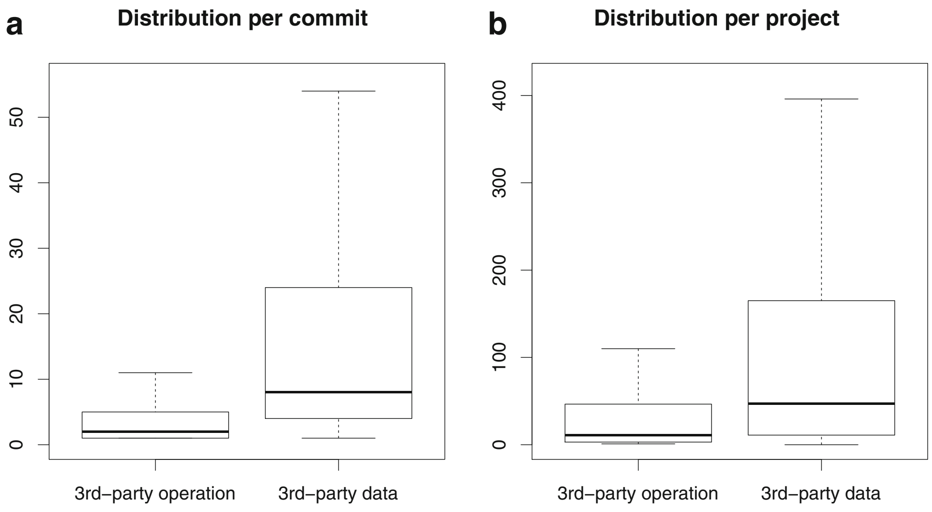

Fig. 21 Distribution of the number of third-party operation extensions and third-party data extensions per commit (**a**) and project (**c**), and proportion of third-party extensions compared to the total number of methods and classes added, (**b**) per commit and (**d**) per project

extension omitted): Third-party operation extensions are considerably smaller than third-party data extensions (median: 11 versus 47; first quartile: 3 versus 11; third quartile: 46.5 versus 165). As before, the difference is statistically significant, with a smaller, but still consequent, effect size ($\hat{A}_{12} = 0.69$ in favor of weighted data extensions).

9.3 Summary

Considering the weight of third-party extensions somewhat dampens our previous result about the comparable frequencies of third-party data and operation extensions. It seems that, all in all, third-party data extensions are as common as third-party operation extensions, but are overall much larger.

This leads us to conclude that, when given a language mechanism for defining third-party operation extensions, programmers tend to use it: 78 % of selected projects use it, and its use appears in 7 % of selected commits. Research providing ways to seamlessly extend third-party library is hence justified by an actual need in practice. However, the comparable frequency of third-party data extension (8 % of selected commits, 100 % of selected projects), and their much larger size goes to show that the classical third-party extension mechanisms—subclassing a framework—is alive and well: the bulk of the third-party extensions are performed in this way. Of course, similar studies in other languages supporting third-party extensions would be needed to confirm these findings.

10 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 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.

10.1 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 analyzing the distribution of the variables and 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.

Our analysis focuses on the addition of methods, and therefore does not identify field additions and other modifications that may occur as part of an extension. This means for instance that the object composition approach to operation extensions described in

Section 2.1 would not be detected. Consequently, we may be under-estimating the negative impact of operation extensions in practice.

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 number of operation extensions that are performed.

While we study the prevalence of the *class extension* mechanism of Smalltalk in Section 9, we are only able to consider third-party operation extensions individually. We cannot relate the addition of several methods of the same name to different third-party classes, because we cannot determine whether they belong to the same class hierarchy. In particular, the repository we analyze does not allow us to determine the specific version of an external library at the time it is extended.

10.2 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 (Schwarz et al. 2012). This aligns with the code duplication rate found in the literature (Kapser and Godfrey 2006; Mayrand et al. 1996). 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.

In case of the extension analysis, it is not always clear if a class extending another class from an external project should really be considered as a third-party data extension because in Smalltalk also core packages are organized as packages external to a user-defined package (that is, subclassing classes such as `TestCase` or `Exception` leads to a third-party data extension). Hence we might over-estimate the number of third-party data extensions as we only exclude `Object` as an external class of which user-defined subclasses are not considered to be third-party data extensions.

We took dispositions against the ecological fallacy (Posnett et al. 2011)—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 and/or commits also applied at the level of projects, when it was pertinent to do so.

10.3 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 quite representative of object-oriented software in Smalltalk. We however cannot ascertain whether practices specific to the community of Smalltalk users would bias the results towards that specific programming language. A replication in another object-oriented language would clarify whether there is a bias.

Another 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.

11 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 131,544 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 the number of methods in an extension 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.
5. Although methods involved in data or operation extensions are slightly less change-prone than normal methods, maintaining them can be difficult. In particular changes to operation extensions often encompass a high entropy, that is, all implementors of an operation extension have a similar likelihood to change. To make matters worse, the entropy generally increases with the number of implementors of an operation extension and also with their change proneness, which renders the maintenance of operation extensions difficult in many cases.
6. Finally, third-party data and operation extensions are nearly equally common in a language that supports both. However, third-party data extensions are considerably larger than third-party operation extensions. Both mechanisms are useful; however the familiar object-oriented extension mechanism of subclassing an external class is still the work horse for most systems, leaving method addition to external classes for more focused extensions.

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, its presence in a majority of selected projects shows that practical solutions will be used when available.

Acknowledgments We thank the ECOOP and EMSE reviewers for their thorough and helpful comments. R. Robbes and É. Tanter are partially funded by FONDECYT Projects 11110463 and 1110051, respectively.

References

Arcuri A, Briand LC (2011) A practical guide for using statistical tests to assess randomized algorithms in software engineering. In: Proceedings of the 33rd international conference on software engineering, (ICSE 2011). pp 1–10

Aversano L, Canfora G, Cerulo L, Del Grosso C, Di Penta M (2007) An empirical study on the evolution of design patterns. In: Proceedings of the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT international symposium on foundations of software engineering (ESEC/SIGSOFT FSE 2007). pp 385–394

Baxter G, Frean MR, Noble J, Rickerby M, Smith H, Visser M, Melton H, Tempero ED (2006) Understanding the shape of Java software. In: Proceedings of the 21st annual ACM SIGPLAN conference on object-oriented programming, systems, languages, and applications (OOPSLA 2006). pp 397–412

Booch G (1994) Object-oriented analysis and design with applications, 2nd edn. Addison-Wesley, Reading

Callaú O, Robbes R, Tanter É, Roethlisberger D (2012) How (and why) developers use the dynamic features of programming languages: the case of Smalltalk. Empirical Software Engineering. Available Online: doi:10.1007/s10664-012-9203-2

Clifton C, Leavens GT, Chambers C, Millstein T (2000) MultiJava: modular open classes and symmetric multiple dispatch in java. In: Proceedings of the 15th international conference on object-oriented programming systems, languages and applications (OOPSLA 2000). ACM SIGPLAN notices, 35(11). ACM Press, Minneapolis, pp 130–145

Cook WR (1990) Object-oriented programming versus abstract data types. In: Proceedings of the REX workshop/school on the foundations of object-oriented languages, volume 73 of Lecture Notes in Computer Science. Springer

Cook WR (2009) On understanding data abstraction, revisited. ACM SIGPLAN Not 44(10):557–572

Erlikh L (2000) Leveraging legacy system dollars for e-business. IT Prof 2(3):17–23

Gamma E, Helm R, Johnson R, Vlissides J (1994) Design patterns: elements of reusable object-oriented software. Professional computing series. Addison-Wesley, Reading

Gîrba T, Lanza M, Ducasse S (2005) Characterizing the evolution of class hierarchies. In: Proceedings of the 9th European conference on software maintenance and reengineering (CSMR 2005). pp 2–11

Gorschek T, Tempero ED, Angelis L (2010) A large-scale empirical study of practitioners' use of object-oriented concepts. In: Proceedings of the 32nd ACM/IEEE international conference on software engineering (ICSE 2010). pp 115–124

Grechanik M, McMillan C, DeFerrari L, Comi M, Crespi S, Poshyvanyk D, Fu C, Xie Q, Ghezzi C (2010) An empirical investigation into a large-scale Java open source code repository. In: Proceedings of the 4th international symposium on empirical software engineering and measurement (ESEM 2010). pp 11:1–11:10

Hassan AE (2009) Predicting faults using the complexity of code changes. In: Proceedings of the 31st international conference on software engineering. IEEE Computer Society, Washington DC, pp 78–88

Kapser CJ, Godfrey MW (2006) Supporting the analysis of clones in software systems: a case study. J Softw Maint Evol Res Pract 18(2):61–82

Kiczales G, Hilsdale E, Hugunin J, Kersten M, Palm J, Griswold W (2001) An overview of AspectJ. In: Knudsen JL (ed) Proceedings of the 15th European conference on object-oriented programming (ECOOP 2001), number 2072 of Lecture Notes in Computer Science. Springer, Budapest, pp 327–353

Krishnamurthi S, Felleisen M, Friedman DP (1998) Synthesizing object-oriented and function design to promote reuse. In: Jul E (ed) Proceedings of the 12th European conference on object-oriented programming (ECOOP 98), volume 1445 of Lecture Notes in Computer Science. Springer, Brussels, pp 91–113

Lehman M, Belady L (1985) Program evolution: processes of software change. London Academic Press, London

Louridas P, Spinellis D, Vlachos V (2008) Power laws in software. ACM Trans Softw Eng Methodol 18(1). Article No. 2

Mayrand J, Leblanc C, Merlo EM (1996) Experiment on the automatic detection of function clones in a software system using metrics. In: Proceedings on the 1996 international conference on software maintenance. pp 244 –253

Meyer B (2009) Software architecture: functional vs. object-oriented design. In: Spinellis D, Gousios G (eds) Beautiful Architecture. OReilly, pp 315–348

Oliveira BCDS (2009) Modular visitor components: a practical solution to the expression families problem. In: Drossopoulou S (ed) Proceedings of the 23rd European conference on object-oriented programming (ECOOP 2009), number 5653 in Lecture Notes in Computer Science. Springer, Genova, pp 269–293

Parnin C, Bird C, Murphy-Hill E (2012) Adoption and use of java generics. Empir Softw Eng 18(6):1047–1089. http://link.springer.com/article/10.1007%2Fs10664-012-9236-6

Posnett D, Filkov V, Devanbu P (2011) Ecological inference in empirical software engineering. In: Proceedings of the 26th ACM/IEEE international conference on automated software engineering (ASE 2011). pp 362–371

Reynolds JC (1975) User-defined types and procedural data structures as complementary approaches to data abstraction. In: Proceedings of the conference on new directions in algorithmic languages. Munich, pp 157–168

Robbes R, Lanza M (2005) Versioning systems for evolution research. In: IWPSE 2005: proceedings of the 8th international workshop on principles of software evolution. pp 155–164

Robbes R, Lungu M (2011) A study of ripple effects in software ecosystems. In: Proceedings of the 33rd ACM/IEEE international conference on software engineering (ICSE 2011), new ideas and emerging results track. ACM Press, Honolulu, pp 904–907

Robbes R, Lungu M, Röthlisberger D (2012a) How do developers react to API deprecation? The case of a Smalltalk ecosystem. In: FSE-20: proceedings of the symposium on the foundations of software engineering. p 56

Robbes R, Röthlisberger D, Tanter É (2012b) Extensions during software evolution: do objects meet their promise? In: Noble J (ed) Proceedings of the 26th European conference on object-oriented programming (ECOOP 2012), volume 7313 of Lecture Notes in Computer Science. Springer, Beijing, pp 28–52

Schwarz N, Lungu M, Robbes R (2012) On how often code is cloned across repositories. In: Proceedings of the 34th ACM/IEEE international conference on software engineering (ICSE 2012, NIER Track)

Shalloway A, Trott JR (2004) Design patterns explained: a new perspective on object-oriented design, 2nd edn. Addison-Wesley, Reading

Shannon CE (1948) A mathematical theory of communication. Bell Syst Tech J 27

Tempero ED, Noble J, Melton H (2008) How do Java programs use inheritance? An empirical study of inheritance in Java software. In: Proceedings of the 22nd European conference on object-oriented programming (ECOOP 2008). pp 667–691

Torgersen M (2004) The expression problem revisited (four new solutions using generics). In: Odersky M (ed) Proceedings of the 18th European conference on object-oriented programming (ECOOP 2004), number 3086 in Lecture Notes in Computer Science. Springer, Oslo, pp 123–146

Van Rysselberghe F, Demeyer S (2007) Studying versioning information to understand inheritance hierarchy changes. In: Proceedings of the 4th international workshop on mining software repositories (MSR 2007). p 16

Vargha A, Delaney HD (2000) A critique and improvement of the CL common language effect size statistics of McGraw and Wong. J Educ Behav Stat 25(2):101–132

Wadler P (1998) The expression problem. Mail to the java-genericity mailing list

Zenger M, Odersky M (2005) Independently extensible solutions to the expression problem. In: Workshop on foundations of object-oriented languages (FOOL). Long Beach

Zimmermann T, Weißgerber P, Diehl S, Zeller A (2005) Mining version histories to guide software changes. IEEE Trans Softw Eng 31(6):429–445

Romain Robbes is assistant professor at the University of Chile (Computer Science Department), in the PLEIAD research lab, since January 2010. He earned his PhD in 2008 from the University of Lugano, Switzerland and received his Masters degree from the University of Caen, France. His research interests lie in Empirical Software Engineering and Mining Software Repositories. He authored more than 50 papers on these topics at top software engineering venues (ICSE, FSE, ASE, EMSE, ECOOP, OOPSLA), and received best paper awards at WCRE 2009 and MSR 2011. He was program co-chair of IWPSE-EVOL 2011, IWPSE 2013, and WCRE 2013, is involved in the organisation of ICSE 2014, and the recipient of a Microsoft SEIF award 2011.

David Röthlisberger is an assistant professor at Universidad Diego Portales, Santiago, Chile. He received his PhD degree in computer science from the University of Bern, Switzerland, in 2010. His research interests include software maintenance and visualization, integrated development environments, empirical software engineering, and mining software repositories. He is a member of ACM and IEEE.

Éric Tanter is an Associate Professor in the Computer Science Department of the University of Chile, where he founded and co-leads the PLEIAD laboratory. He received the PhD degree in Computer Science from both the University of Nantes and the University of Chile. His research interests cover programming languages and software engineering, ranging from the theoretical underpinnings of programming languages to the empirical study of the practice of programming. He is a member of the ACM, the IEEE, and the SCCC.