Noname manuscript No.
(will be inserted by the editor)

How (and Why) Developers Use the Dynamic Features of Programming Languages: The Case of Smalltalk

Oscar Callaú · Romain Robbes · Éric Tanter · David Röthlisberger

the date of receipt and acceptance should be inserted later

Abstract The dynamic and reflective features of programming languages are powerful constructs that programmers often mention as extremely useful. However, the ability to modify a program at runtime can be both a boon—in terms of flexibility—, and a curse—in terms of tool support. For instance, usage of these features hampers the design of type systems, the accuracy of static analysis techniques, or the introduction of optimizations by compilers. In this paper, we perform an empirical study of a large Smalltalk codebase—often regarded as the poster-child in terms of availability of these features—, in order to assess how much these features are actually used in practice, whether some are used more than others, and in which kinds of projects. In addition, we performed a qualitative analysis of a representative sample of usages of dynamic features in order to uncover (1) the principal reasons that drive people to use dynamic features, and (2) whether and how these dynamic feature usages can be removed or converted to safer usages. These results are useful to make informed decisions about which features to consider when designing language extensions or tool support.

Keywords Dynamic Languages, Static Analysis, Smalltalk, Reflection

1 Introduction

Dynamic object-oriented languages such as Smalltalk [GR83] or Ruby allow developers to dynamically change the program at runtime, for instance by adding or altering methods; languages such as Java, C# or C++ provide reflective interfaces to provide at least part of the dynamism offered by dynamic languages. These features are extremely powerful: the Smalltalk language for instance ships with an integrated development environment (IDE) that uses these dynamic features to create, remove, and alter methods and classes while the system is running.

If powerful, these dynamic features may also cause harm: they make it impossible to fully check the types of a program statically; a type systems has to fall back to dynamic checking if a program exploits dynamic language features. Until

PLEIAD Laboratory, Computer Science Department (DCC), University of Chile
E-mail: {oalvarez,rrobbes,etanter,roethlis}@dcc.uchile.cl

recently, the problem of static analysis in the presence of reflection was largely sidestepped; current solutions to it fall back on dynamic analysis in order to know how the dynamic features are exercised at runtime [BSS+11]. Another example is the (static) optimization of program code, which is impossible to achieve for code using any dynamic language feature. Moreover, tools are affected by the use of these features. For instance, a refactoring tool may fail to rename all occurences of a method if it is used reflectively, leaving the program in an inconsistent state. In short, dynamic language features are a burden for language designers and tool implementors alike.

This problem is exacerbated since language designers and tool implementors do not know how programmers are using dynamic language features in practice. Dynamic features might only be used in specific applications domains, for instance in parsers/compilers, in testing code, in GUI code, or in systems providing an environment to alter code (eg. an IDE). Having precise knowledge about how programmers use dynamic features in practice, for instance how often, in which parts, and in which types of systems they are used, can help language designers and tool implementors find the right choices on how to implement a specific language extension, static analysis, compiler optimization, refactoring tool, etc. If it turns out that a given dynamic feature is used in a minority of cases, then it may be reasonable to provide a less-than optimal solution for it (such as resorting to dynamic type checking in a static type system). On the other hand, if the usage is pervasive, then a much more convincing solution needs to be devised. Hence, it is of a vital importance to check the assumptions language designers and tool implementors might have against reality.

In this paper, we perform an empirical study of the usage of dynamic language features by programmers in Smalltalk. We survey 1,000 Smalltalk projects, featuring more than 4 million lines of code. The projects are extracted from Squeaksource, a software super-repository hosting the majority of Smalltalk code produced by the Squeak and Pharo open-source communities [LRL10]. We statically analyze these systems to reveal which dynamic features they use, how often and in which parts. Next, we interpret these results to formulate guidelines on how language designers can best deal with particular language features, depending on how and how frequent such features are used in practice.

In addition to these quantitative results, we also performed a qualitative analysis of a representative sample of 377 usages of dynamic features across our corpus. By focusing on this restricted data set, we were able to perform a deeper analysis, with two goals. The first goal is to investigate the principal reasons why developers use dynamic features, in order to pinpoint areas of applications, and types of computations that are more prone to these usages than other. Understanding these practices can also indicate ways to extend a programming language to support certain patterns in a more robust manner. The second goal is to determine whether some of these dynamic usages can be removed (or converted to safer usages of the same feature), in order to reduce the extent of the problem and simplify static analyses. Thus, we provide guidelines on how to refactor some of the common idioms we encountered.

We focus on the Smalltalk programming language because it is a very salient data point. Smalltalk is, alongside with LISP, one of the languages with the most support for dynamic features, since it is implemented in itself. The kernel of the language (classes, methods, etc), and the development tools (compiler, code browser,

etc) make extensive use of the dynamic features in order to implement the vision of a "live" programming systems. Thus our hypothesis is that Smalltalk represents an *upper bound estimate* of the use of dynamic features in practice. For Smalltalk programmers, this dynamic behavior is the norm, hence they should use it more than their counterparts in other languages—especially since they are so easy to access.

Contributions. This paper explores the usage of dynamic features in Smalltalk in order to gain insight on the usage of these features in practice.

Our first contribution is a quantitative analysis of a large corpus of source code (1,000 Smalltalk projects) in order to validate, or invalidate, the following hypotheses:

1. Dynamic features are not used often. More precisely, we are interested in determining which features are used more than others.
2. Most dynamic features are used in very specific kinds of projects. We conjecture that they are more often used in core system libraries, development tools, and tests, rather than in regular applications.
3. The specific features that have been integrated in more static languages over time (eg. Java) are indeed the most used.
4. Some usages of dynamic features are statically tractable, unproblematic for static analyses and other tools.

While our study allows us to validate these hypotheses, we do so with some caveats; this makes it still necessary for language designers, tool implementors, and developers of static analyses to carefully consider dynamic features. We provide prelimary guidelines as to which are most important.

Since dynamic features can not be ignored outright, our second contribution is the qualitative analysis of a representative sample of 377 dynamic feature usages in order to better understand the reasons why developers resort to using these dynamic features, and whether some of them can be refactored away, or converted to safer usages of the same features. We find that some of the usages are unavoidable, others are due to limitations of the programming language used, some can be refactored, and others are mostly superfluous.

As a consequence of these two studies, we gain insight into how language designers and tool providers have to deal with these dynamic features, and into why the developers need to use them—yielding further insights in the limitations of programming languages that need to be addressed.

Structure of the Paper. We review related work (Section 2) before giving the necessary background on Smalltalk to the reader (Section 3). We then describe our experimental methodology, analysis infrastructure, and the dynamic features we look at (Section 4); our results follow (Section 5). We then discuss these results and their implications (Section 6). Later, we perform our qualitative analysis of dynamic feature usages and identify refactorable occurences (Section 7). We close the paper by first discussing the potential threats to the validity of this study (Section 8), before concluding (Section 9).

2 Related Work

There have been a number of empirical studies on the usage of programming language features by developers.

Knuth studied a wide variety of Fortran programs, informing quantitatively "what programmers really do" [Knu71]. Knuth performed static analysis on a sample of Fortran programs, and dynamic analysis on a smaller sample, recording the frequency of execution of each kind of instruction. Knuth found several possible optimizations to compilers and suggested compiler writers to consider not only the best and the worst cases, but also the average case of how programers use language features in order to introduce optimizations.

Melton and Tempero measured the size of cycles among classes in 78 Java applications, and found that most applications featured very large cycles (sometimes in the thousands of classes) [MT07].

Tempero *et al.* characterized the usage of inheritance in 90 Java programs, and found a higher usage of inheritance than they expected [TNM08]. Rysselberghe and Demeyer took evolution into account and proposed hypotheses on how the hierarchies change over time, based on observations about the evolution of two Java systems [RD07]. Later, Tempero analyzed a corpus of 100 Java programs in order to characterize how fields were used [Tem09]: a large number of classes had non-private fields, but less were actually accessed in practice.

Muschevici *et al.* performed an empirical study on how multiple dispatch is used in 9 applications written in 6 languages supporting it, and contrasted it with the Java corpus mentioned above [MPTN08].

Malayeri and Aldrich inspected 29 Java programs in order to determine if they would have benefited from structural (instead of nominal) subtyping; they found that the programs would benefit somewhat [MA09].

A large-scale study (2,080 Java applications found on Sourceforge) by Grechanik *et al.* asks 32 research questions on the usage of Java by programmers [GMD+10], related to the size of the applications, the number of arguments in methods, whether methods are overriden or not, etc.

Finally, Parnin *et al.* performed an empirical study of 20 Java systems [PBMH11], with the goal of assessing how programmers transitioned to Java Generics—or not. They found that adoption rates and delay to adoption varied from project to project (with some projects not adopting generics), and that usually one or two developers drove the adoption of Java generics.

Dynamic Features. In addition, several pieces of work have specifically investigated the usage of dynamic features in Java, Python and Javascript.

Bodden *et al.* investigated the usage of Java reflection in the case of the DaCapo benchmark suite, and found that the benchmark harness loads classes dynamically, and executes methods via reflection, causing the call graph extracted from static analysis to significantly differ from the call graph actually observed at runtime [BSS+10]. Furthermore, the class loaders that DaCapo uses are non-standard.

Holkner and Harland investigated the dynamic behavior of 24 Python programs, by monitoring their execution [HH09]. They found that the Python program in their corpus made a heavier usage of dynamic features during their startup

Smalltalk	Java
foo bar.	foo.bar();
foo bar: baz.	foo.bar(baz);
foo bar: baz with: quux.	foo.bar(baz, quux);
p := Point new.	Point p = new Point();
↑ foo	return foo;
self	this
super	super
'String'	"String"
#symbol	String.intern("symbol");

Table 1 Smalltalk and Java syntax compared

phase, but that many of them also used some of these features during their entire lifetime.

Most directly related to our work is the study of Javascript dynamic features by Richards *et al.* [RLBV10]. They analyzed a large amount of Javascript code from popular web sites, in order to verify whether the assumptions that are made in the literature about the usage of the dynamic features of Javascript match reality. Some of the assumptions they checked were: "the use of eval is infrequent and does not affect semantics" (found to be false), or "the prototype hierarchy is invariant" (also false); most of the assumptions were found to be violated. In further work, the same authors performed a more thorough analysis of the usages of the *eval* function in Javascript [RLBV11]. Again, assumptions that *eval* is rarely used were found to be wrong. While Richards *et al.* use dynamic analysis to monitor manual interaction on 103 websites, we use static analysis on 1,000 Smalltalk projects. An innovation of our study is to consider the kinds of projects that use the features; this is particularly relevant in a live environment like Smalltalk, where the whole system is developed in itself.

3 Smalltalk Basics

Smalltalk [GR83] is a pure object-oriented language (everything is an object) with no static typing. Compared to mainstream OO languages, Smalltalk has a distinct terminology: Smalltalk objects communicate by sending messages to each other. Each message contains a selector (method name) and arguments. When the object receives the message, it will look up the selector in its method dictionary, retrieve the associated method, and execute it with the arguments of the message (if any). Smalltalk's syntax is also distinct from C-like languages. We provide equivalent expressions for common cases in Table 1. Note that since Smalltalk has first-class classes, there are no constructors; instantiating a class is done by sending a message to a class object. Smalltalk features the concept of Symbols, which are unique strings; selectors are such symbols.

Many concepts that are implicit in other languages are made explicit through reification, and can be directly manipulated by programs; this is the case for classes, methods, and blocks of code. Smalltalk's programming environment is defined in itself and makes extensive use of these dynamic features: one can effortlessly use the compiler to add new behavior at runtime; in fact, this is the default way programs are built in Smalltalk. The prominent dynamic features of Smalltalk, which we investigate in this paper, are:

- **Classes as first-class objects.** Classes can be passed as arguments to functions, which makes it hard to know the type of new objects statically. Classes can also be created programmatically.
- **Behavioral reflection.** In Smalltalk, one can invoke a method based on its dynamically-determined name. It is also possible to access or modify the value of a dynamically-determined field in an object. In addition, Smalltalk also supports swapping all pointers between two objects.
- **Structural reflection.** Classes can be created and removed from the program at runtime; their subclasses can be dynamically changed; methods can be added, removed, or recompiled dynamically.
- **System dictionary.** In Smalltalk, the system dictionary is a global variable, called Smalltalk. This dictionary is the central access point to all global variables in the system, including all classes. The system dictionary provides several methods that can be used to access and alter classes at runtime.

All these features are readily available to programmers. We conjecture that dynamic features are used extensively in the core language libraries and development tools, yet it remains to be seen if and how they are actually used in applications.

4 Experimental Setup

To find out how developers use the dynamic features provided by Smalltalk in practice, we perform an analysis of a large repository of Smalltalk projects. This section describes the experimental setup, that is, the methodology applied to perform the analysis, the analysis infrastructure, and an explanation of the dynamic features we are analyzing. This section and the following focus only on the quantitative analysis; the qualitative analysis of a representative sample of dynamic feature usages is described entirely in Section 7.

4.1 Methodology

We started our analysis by looking at all 1850 software projects stored in Squeaksource in a snapshot taken in early 2010. We ordered all projects by size and selected the top 1000 projects, in order to exclude small or toy projects. Since Squeaksource is the *de facto* source code repository for open-source development in the Squeak and Pharo communities, we believe this set of projects is representative of medium to large sized Smalltalk projects originating from both open-source and academia. Table 2 summarizes the top ten projects sorted by lines of code (LOC), and also shows number of classes and methods. The last row shows the total for the 1000 projects analyzed in this study.

In order to analyze the 1000 projects, we developed a framework[1] in Pharo[2] to trace statically the use of dynamic features in a software ecosystem. This framework is an extension of *Ecco* [LRL10], a software ecosystem model to trace dependencies between software projects. Our analyzer follows three principal steps: *Trace*, *Collect* and *Classify*.

[1] Available at http://www.squeaksource.com/ff
[2] http://www.pharo-project.org

Project	LOC	Classes	Methods
Morphic	124,729	676	18,154
MinimalMorphic	101,190	483	13,887
System	91,706	502	10,970
Formation	89,172	695	9,833
MorphicExt	69,892	236	9,461
Balloon3D	68,020	397	7,784
Network	58,040	447	8,207
Collections	55,254	405	9,093
Graphics	52,837	139	5,267
SeaBreeze	47,324	228	3,466
Total (1000)	**4,445,415**	**47,720**	**652,990**

Table 2 The 10 largest projects in our study.

To *Trace*, first the analyzer reads Smalltalk package files from disk and builds a structure (an ecosystem) which represents all packages available on disk. Later, the analyzer flows across the ecosystem structure parsing all classes and methods from each package. In the method parsing process, the analyzer traces statically all calls of the methods that reflect the usage of dynamic features in Smalltalk. Section 4.3 describes these dynamic features in more details and lists the corresponding method names.

The *Collect* step gathers the sender, receiver and arguments of each traced message call AST nodes. The collected data is stored in a graph structure, which recursively catalogs the sender into packages and classes, and the receiver and arguments into several categories: literals (*e.g.* strings, nil, etc), local variables, special variables (*i.e.* self or super), literal class names, and arbitrary Smalltalk expressions.

The third step, *Classify*, is performed in the graph structure. Each call site is classified either as *safe* or *unsafe*: A safe call site is one for which the behavior can be statically determined (*e.g.* the receiver and arguments are literals, for instance), whereas an unsafe call may not be fully statically determined. The exact definition of what is safe and unsafe depends on each feature, as described in Section 4.3.

Characterizing usages as safe or unsafe is an indicator of *how* dynamic features are used, and how challenging it may be for a static analysis or development tool to support it. This study also answers the *where* question: which kind of projects make use of these features. For this, we introduce project categories, described below.

4.2 Project Categories

In order to characterize in which kinds of projects dynamic features are used, we classified each project according to five different categories:

- System core (**System, 25 projects**): Projects that implement the Smalltalk system itself.
- Language extension (**Lang-Ext., 55 projects**): Projects that extend the language, but are not part of the core (eg. extension for mixins, traits, etc.).
- Tools and IDE (**Tools, 63 projects**): Projects building the Smalltalk IDE and other IDE-related tools.

– Test suites (**Tests, 24 projects**): Projects or parts of projects representing unit and functionality tests[3].
– Applications (**Apps, 833 projects**): Conventional applications, which do not fit in any of the other categories; this is the overwhelming majority.

4.3 Analyzed Dynamic Features

We consider four groups of dynamic features of Smalltalk in this study: first-class classes, behavioral reflection, structural reflection, and system dictionary. We came to this classification by iterating over a sample of the usages of the features in order to delineate their intent; This process was supported by our own experience in using the features. Non-standard and seldom-used features were ommitted. In each of these groups, the use of the different features are identified by specific selectors, which we have identified based on the experience of the authors as Smalltalk developers. In addition to describing each feature and its corresponding selectors, this section explains how specific usages are characterized are safe or unsafe.

4.3.1 First-class Classes

This category includes features that are related to the usage of classes as first-class objects. As opposed to other object-oriented languages such as Java, Smalltalk classes can be receivers or arguments to methods. The use of first-class classes complicate matters for static analysis especially with respect to instance creation and class definition, as one can not know which class will be instantiated or created.

Instance Creation. In Smalltalk, the typical instance creation protocol consists of new, which is may be overriden in classes, while basicNew is the low-level method responsible for actually creating new instances. When tracing all occurrences of invocations of basicNew in the analyzed projects, we consider only two kind of occurrences to be unsafe:

```
x basicNew.
(z foo) basicNew.
```

In the first case the receiver is a local variable, in the second case the receiver is the result of a method invocation, or more generally, any arbitrary expression. Usages of basicNew with a literal class name or the pseudo-variables self or super as receiver are considered safe. Note that the type of self is statically tractable using self types, as in Strongtalk [BG93].

Class Creation. To create a new class, Smalltalk offers a range of subclass: methods that only differ in the arguments they accept. As for instance creation, we only consider a message send of subclass: to be unsafe if: (1) the receiver is a local variable or a complex Smalltalk expression, or (2) the argument (the class name to be created) is not a symbol. Examples of safe calls are:

```
Point subclass: #ColorPoint.
self subclass: #ColorPoint.
```

[3] All subclasses of TestCase are considered to represent tests, no matter how the rest of the project is categorized.

Examples of unsafe method calls are:

```
c subclass: #MySubClass.
Point subclass: x name.
```

The first example subclasses an undetermined class c, while the second example creates a subclass of Point with an undetermined name (the result of sending name to x).

4.3.2 Behavioral Reflection

Behavioral reflective features of Smalltalk allow programmers to change or update objects at runtime, or to dynamically compute the name of methods to be executed. We distinguish between the following features: object reference update, object field update, and message sending.

Object Reference Update. Selectors such as become: allow Smalltalk programmers to swap object references between the receiver and the argument. After a call to become:, all pointers to the receiver now point to the argument, and vice versa; this affects the entire memory. Determining at compile time, if this reference swap is safe or unsafe is challenging. We consider all calls to these selectors to be unsafe.

Object Field Read. In Smalltalk, object fields are private; they are not visible from the outside and must be accessed by getter and setter methods. The Smalltalk reflection API provides methods to access them by using their names or indexes, *e.g.* instVarNamed:. We categorize as safe usages of an object field read those with either a number, symbol or string literal as argument.

Object Field Update. Complementary to previous features, Smalltalk allows developers to reflectively change an object field value. For that propose, the Smalltalk reflection API offers methods such as instVarAt:put: and variants, to write into object fields without using the corresponding setter methods. We consider safe calls to be those where the object field index (the selector's first argument) is a number, symbol or string literal.

Message Sending. The perform: selector invokes a method by passing its name (a symbol) as the argument of the call, as well as the receiver object. This feature is also provided by the Java reflection API. Safe calls are those where the method name (the argument in the expression) can be determined statically—*i.e.* a symbol. In unsafe calls, the argument is a local variable or a composition of message calls (*e.g.* a string concatenation). Examples of unsafe calls are:

```
x perform: aSelector.
x perform: ('selectorPrefix', stringSuffix) asSymbol.
```

4.3.3 Structural Reflection

With the structural reflective features of Smalltalk, developers can modify the structure of a program at runtime by dynamically adding or removing new classes

or methods. We consider the following structural reflective features:

Class Removal. In Smalltalk, classes can be removed from the system at runtime. We include this feature to be analyzed through the removeFromSystem selector where the receiver is the class to remove. In our analysis, we consider unsafe occurrences to be calls in which the receiver is a local variable, or a Smalltalk expression. Examples are:

```
c removeFromSystem.
(x class) removeFromSystem.
```

Superclass Update. Smalltalk programmers can change at runtime the behavior of a class by updating the superclass binding. This powerful feature is handled by superclass: selectors. Safe calls to them are those where both the receiver (the subclass) and the argument (the new superclass) are either a literal class name (including nil[4]) or self. Any other case is potentially unsafe. Safe examples are:

```
Point3D superclass: MyPoint.
self superclass: nil.
```

Method Compilation. Adding behavior at runtime allows programmers to dynamically load runnable code. Smalltalk provides selectors such as compile: to compile and add methods to a particular class. Calls where the argument—the code to be compiled, or the selector name—is lexically a string are safe; others are not. We further categorize safe calls to the compile selector in the following categories: *trivial*, simple code such as returning a constant or a simple expresion ; *getter/setter*, which returns/sets an instance variable; and *arbitrary* code—everything else.

Method Removal. This feature complements the one above, adding the capability to remove behavior from a class at runtime, with selectors such as removeSelector:. When tracing all occurrences of invocations of this kind of selectors, we categorize those occurrences where the argument (the selector name) is a variable name or a composition of message calls (*e.g.* a string composition) as unsafe. Therefore, safe occurrences are when the argument is lexically a symbol. Example of unsafe occurrences are the following:

```
c removeSelector: aSelector.
c removeSelector: ('prefix' , varSuffix ) asSymbol.
```

4.3.4 System Dictionary

The Smalltalk system dictionary is a global variable (called Smalltalk), which registers all class definitions in the image. Smalltalk provides several methods to access, add, remove or alter class definitions. We focus only on usages of this dictionary that concern classes; other manipulations of the system dictionary concern global variables in general, and are hence out of the scope of this study. We distinguish between usages of this dictionary for reading, and writing. We also detect when aliases to this dictionary are created (*e.g.* by passing it as argument to a method).

[4] In Smalltalk the root superclass is nil.

Reading. The system dictionary provides several methods to both access class definitions and test for class existence. For example, at: returns the class object whose name is specified with the argument; hasClassNamed: verifies if the system defines a class with the given name. Safe usages are those where the class name (the argument) is a literal symbol or string. Unsafe expression are those where the argument cannot be determined statically, for instance:

```
Smalltalk at: myVar .
Smalltalk hasClassNamed: ('Preffix' , suffixVar) .
```

Writing. The system dictionary can also be used to alter the classes defined in the system. For instance, some usages are:

```
Smalltalk at: #MyClass put: classRef .
Smalltalk removeClassNamed: #MyClass .
```

Both expressions are alternative ways of doing class renaming or removal operations, which are already feasible using the reflective abilities of Smalltalk presented previously. In our study, we classify as safe the usages where the argument in the call is a literal symbol or string, as shown in previous example.

Aliasing. Because the system dictionary is a variable in Smalltalk, programmers can pass it around, and therefore aliases to this variable can be created. Aliasing makes it particularly difficult to track usages of the system dictionary. In this category, we collect direct aliasing (assignment expressions involving the system dictionary) as well as expressions that can yield aliases, *e.g.* passing the dictionary as an argument to a method or returning it in a block. All aliasing occurrences are categorized as unsafe usages.

5 Results

This section presents the results of the study. After presenting general results showing how many and how often projects use dynamic features, we analyze the usage of each dynamic feature in detail. In particular we distinguish between safe and unsafe usages as explained in Section 4.3 and application code, and system, tools, language extensions and tests (Section 4.2). When they exist, we list common patterns of usage of the features.

5.1 General Results

5.1.1 Are dynamic features used often?

In our analysis of the 1,000 projects, we found 20,387 dynamic feature occurrences, *i.e.* , calls to a method implementing a dynamic feature. Only 11,520 methods use at least one dynamic feature; this shows that a fair proportion of methods either use a feature more than once or use several features at once. The 11,520 methods using dynamic features represent 1.76% of the 652,990 methods we analyzed for this study. This shows that use of dynamic features is punctual: most methods do not make use of them.

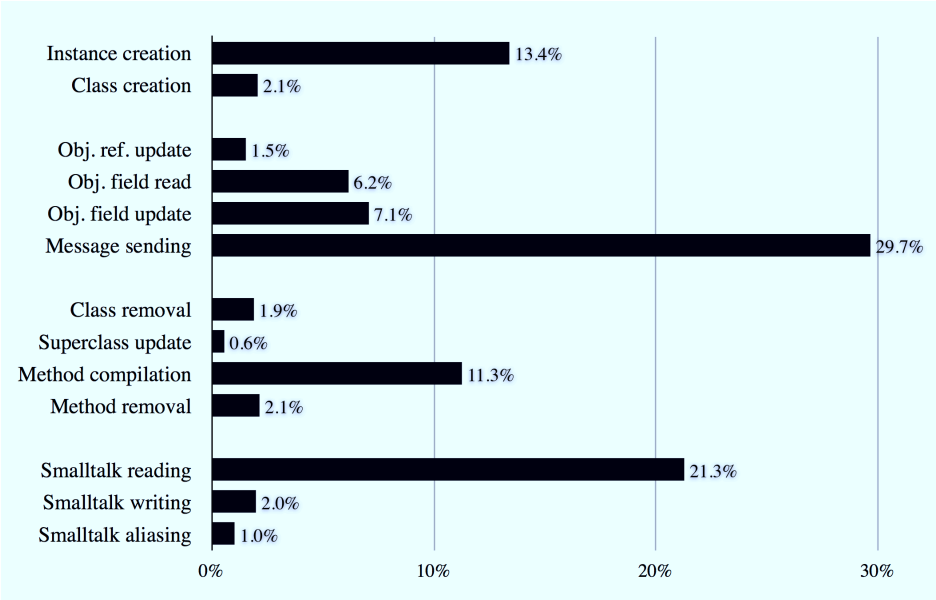

Fig. 1 Distribution of dynamic feature usages.

Of the total methods using dynamic features, 6,524 were in projects classified as "Applications" (1.00% of all analyzed methods) and 3,832 of these use dynamic features that we consider as unsafe (58.74% of the methods using dynamic features, or 0.59% of all methods); these results confirm the previous point.

Projects classified as applications represent 83% of the projects, yet contain only 56.63% of the methods using dynamic features, confirming the fact that other project categories use these features more extensively. Of all the dynamic feature usages, 12,094 were classified as unsafe (59.32%); 5,253 of those were in applications (43.43%).

5.1.2 Do regular applications use less dynamic features?

To confirm assumption 2, which states that most dynamic feature usages occur outside application projects, we ran a statistical test comparing all application projects to all other projects. As the data of dynamic feature usages is normally distributed across projects, we use a Student's t-test (two-tailed, independent, unequal sample sizes, unequal variance) which yields a p-value of 0.00958 (d.f = 610.9, t = 2.599). This result confirms assumption 2.

However, the data stemming from non-application projects has a high variance and its mean number of dynamic feature usages differs considerably from those in application projects. Due to this, the effect size of this experiment is rather low (as expressed with a Cohen's d of 0.169), which would require us to analyze even more projects in order to reliably confirm assumption 2. Still, the current results are a strong indication that dynamic feature usage is more widespread in special projects such as language extensions or system core projects.

5.1.3 What are the most prevalent dynamic features?

Figure 1 shows the distribution of the usage of dynamic features, with a maximum of 6,048 occurrences of message sending (29.67%) and a minimum of 114 occurrences for superclass updates (0.56%). Categories are distributed as follows: first-class classes with 15.46%, behavioral reflection with 44.41%, structural reflection with 15.85% and system dictionary with 24.28%. Four dynamic features—Message sending, Instance creation, Method recompilation, and Reading system dictionary—, account for more than 75% of the usages. Of these, Java provides three in its reflection API (message sending, instance creation, access to the class table), catering to 64% of the usages in the analyzed Smalltalk projects.

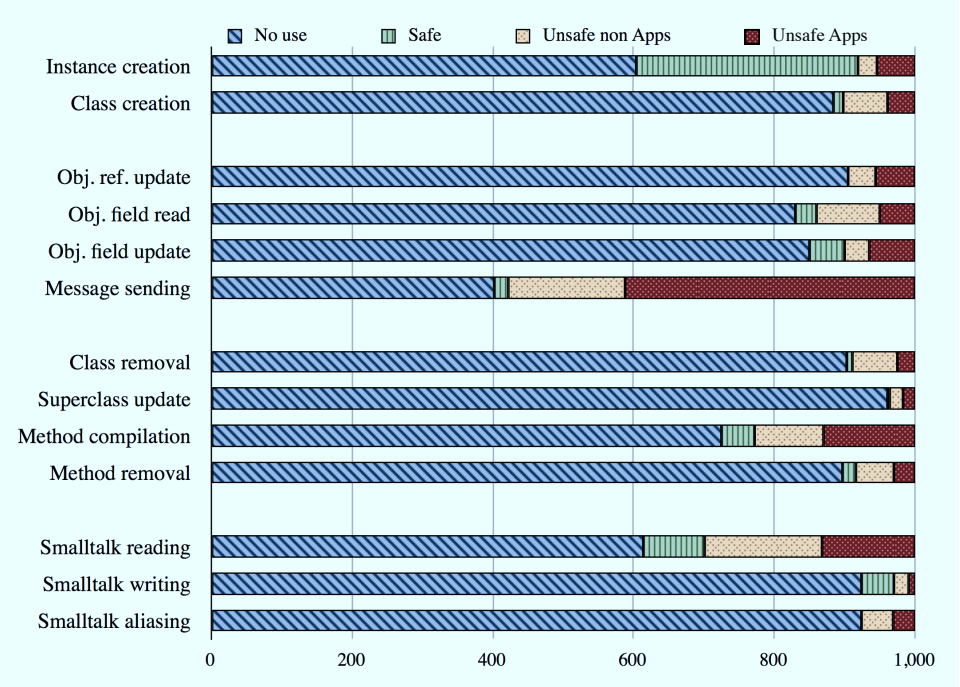

Fig. 2 Per-feature distribution of all projects arranged by category of use.

Figure 2 exhibits the per-feature distribution of all software projects arranged left to right in the following categories: *No Use*, projects with no occurrences of the analyzed feature (blue); *Safe*, projects that have one or more occurrences of the analyzed dynamic feature, but all occurrences are safe (green); *Unsafe in Systems, Tests, Language extensions or Tools* represents all projects in those project categories with at least one unsafe call of the feature (yellow); *Unsafe in Applications* includes application projects with at least one unsafe call (red). Most features (except instance creation, message sending and reading system dictionary) follow a common pattern:

Dynamic Feature	%Safe	%Apps	%Tools	%Ext	%Syst	%Tests
Instance Creation	92.53	4.32 (0.69)	0.59 (1.25)	0.70 (1.70)	1.76 (**9.42**)	0.11 (0.61)
Class Creation	30	18.33 (0.31)	9.76 (2.21)	1.90 (0.49)	8.57 (**4.90**)	31.43 (**18.71**)
Object ref. update	0	45.66 (0.55)	12.22 (1.94)	6.75 (1.23)	24.44 (**9.78**)	10.93 (**4.55**)
Object field read	25.52	35.09 (0.56)	11.24 (2.40)	3.99 (0.97)	23.13 (**12.42**)	1.04 (0.58)
Object field update	61.14	18.67 (0.58)	5.20 (2.12)	1.80 (0.84)	12.91 (**13.29**)	0.28 (0.30)
Message sending	7.03	47.52 (0.61)	26.57 (**4.54**)	3.17 (0.62)	9.79 (**4.21**)	5.92 (2.65)
Class removal	6.24	10.91 (0.14)	8.05 (1.36)	1.56 (0.30)	10.91 (**4.65**)	62.34 (**27.70**)
Superclass update	7.89	42.11 (0.55)	18.42 (3.17)	7.02 (1.39)	7.02 (3.05)	17.54 (**7.93**)
Method compilation	60.02	18.25 (0.55)	7.10 (2.82)	3.22 (1.46)	6.32 (**6.32**)	5.10 (**5.32**)
Method removal	39.13	13.27 (0.26)	15.10 (3.94)	3.43 (1.02)	18.76 (**12.33**)	10.30 (**7.05**)
System dict. reading	48.52	15.86 (0.37)	11 (3.39)	2.47 (0.87)	11.36 (**8.83**)	10.79 (**8.73**)
System dict. writing	80.69	4.95 (0.31)	2.72 (2.24)	0.5 (0.47)	4.95 (**10.25**)	6.19 (**13.36**)
System dict. aliasing	0	28.02 (0.34)	27.54 (**4.37**)	4.83 (0.88)	15.46 (**6.18**)	24.15 (**10.06**)

Table 3 Per-feature distribution of safe and unsafe calls, where unsafe calls are sorted by project category. In bold: category that is considerably over-represented (over-representation factor > 4)

– Many projects do not use the analyzed feature. This category ranges between 725 projects in method definition and 961 in the superclass update feature.
– Unsafe uses are almost equally distributed between applications and other categories, with an average of 45 and 53 projects respectively. Applications with 83% of the projects have comparatively less unsafe uses.
– Finally, projects having only safe usages of a dynamic feature are a minority (excepting instance creation features), with an average of 22 projects.

The cases of instance creation, message sending and reading system dictionary are distinct: 40% of the projects make use of dynamic instance creation, but the majority of them only have safe usages; less than 10% of the projects use it unsafely. Message sending is even more widespread—60% of all projects use it—, but follows an opposite distribution of safe/unsafe usages: most of the projects use it in an unsafe fashion. In the case of reading the system dictionary: 39% of projects use this feature with a majority of unsafe usages —30% of all projects, half of them in applications—. These three features are used pervasively by all kinds of projects.

5.1.4 Interpretation

– The methods using dynamic features are a very small minority. However, the proportion of projects using dynamic features is larger, even if still a minority. This confirms hypothesis 1—dynamic features are not used often—, but shows that they cannot be safely ignored. An analysis of each feature is needed.
– Dynamic features are more often used in core system libraries, development tools, and tests, rather than in regular applications (hypothesis 2). However, it is important to remark that it is not the case that conventional applications use few dynamic features: applications gather nearly half of the unsafe uses. Considering that applications account for 83% of all analyzed projects, applications are nonetheless clearly under-represented in terms of dynamic feature usage compared to most other project categories (cf. Table 3).
– The three most pervasive features—Instance creation, Message sending and Reading the system dictionary—correspond to features that static languages such as Java support, confirming hypothesis 3.

5.2 First-class Classes

For each feature, we provide basic statistics (number of uses, number of unsafe uses, and number of unsafe uses in applications), and a bar chart showing the classification of each feature in various, feature-specific, patterns of usage. We also provide percentage distributions among categories in Table 3. We highlight in bold categories that are particularly over-represented, measured by the over-representation factor (ORF). An ORF of 1 means that a category has a distribution of unsafe calls equal to its representation in the project corpus. The higher the ORF, the more over-represented are unsafe calls in a particular category. For instance, while only 2.5% of all projects belong to the System category, it is responsible for 23.56% of all unsafe instance creation occurrences (1.76% of all instance creations), hence the over-representation of unsafe instance creations in the System category is 9.42. Note that Application projects are under-represented for all dynamic features as denoted by an ORF smaller than 1; for Application project the factor varies between 0.14 for the class removal and 0.69 for the instance creation feature.

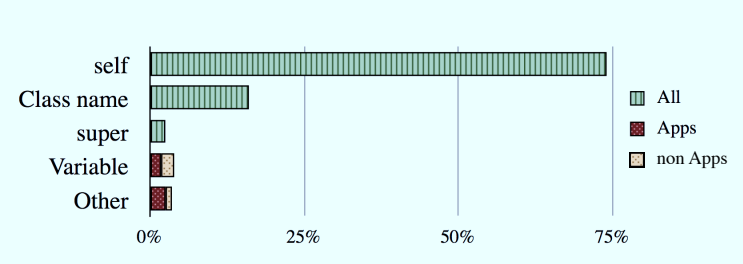

Fig. 3 Safe/unsafe usages of instance creation.

Instance Creation (2,732 calls, 204 unsafe, 118 in Apps) Figure 3 reveals that programmers use instance creation (basicNew) in a statically safe way (92.53%) while unsafe calls (see Table 3 for distribution) are restricted to a few occurrences (7.47%). Applications feature the most unsafe calls (118, *i.e.* 4.32%), but are actually under-represented as 83% of the projects are applications (ORF=0.69). On the contrary, System projects are the most over-represented (1.76%, ORF=9.42).

The most common (and safe) pattern is self basicNew (74%): Programmers define constructor methods (as class methods) and inside them call basicNew. A common unsafe pattern (almost a third of unsafe calls) is to defer the choice of the class to instantiate via polymorphism (self factoryClass basicNew).

Class Creation (420 calls, 294 unsafe, 77 in Apps) Figure 4 and Table 3 show that a strong minority of cases are safe uses (30%); 18% of unsafe usages are in application (ORF=0.31), while more than 50% are in other project categories. Tests are extremely over-represented, with nearly a third of unsafe usages, with an ORF of 18.71. Indeed, tests often create temporary classes for testing purposes, and the ORF confirms that this practice is indeed very common. Likewise, System and—to a lesser extent—Tools are both over-represented (ORFs of 4.90 and 2.21, respectively), each having close to 10% of uses of the features; both project categories

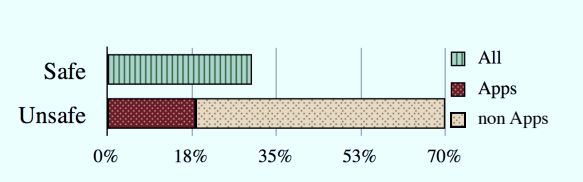

Fig. 4 Safe/unsafe usages of class creation.

are infrastructural in nature and may need to create classes as part of their responsibilities. Most unsafe usages in Apps are in class factory methods generating a custom class name, such as:

```
FactoryClass>>customClassWithSuffix: aStringSuffix
    ↑ Object subclass: ('MySpaceName' , aStringSuffix) asSymbol.
```

To provide perspective, the code base we analyze contains 47,720 statically defined classes, showing that dynamic class creation clearly covers a minority of cases, less than 1%.

Interpretation

- Instance creation is the third-most used dynamic feature, but its usage is mostly safe, with only 118 unsafe usages in applications.
- The majority of class creation uses are unsafe, but most of those are located in non-application code, primarily testing code. A lot of unsafe usages appear to be related to class name generation.
- Some support is still needed for a correct handling of these features in static analysis tools. In particular, support for self-types is primordial to make usages of self and super tractable and hence safe.

5.3 Behavioral Reflection

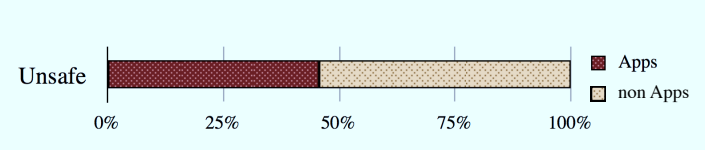

Fig. 5 Unsafe uses of object references updates.

Object Reference Update (311 calls, 311 unsafe, 142 in Apps) According to Table 3, System projects particularly are over-represented (2.5% of projects account for 25% of calls, an ORF of 9.78). For instance, some low-level system operations need to migrate objects when their classes are redefined, and use become: for such a task. Applications however do use this feature somewhat extensively, with more than 45% of calls, see Figure 5. They are still under-represented (ORF=0.55).

Object Field Read (1254 calls, 934 unsafe, 440 in Apps) Reading object field values is a commonly used dynamic feature, accounting for 6.15% of all dynamic feature usages. The distribution of safe (only 25.5%) and unsafe usages is displayed in Figure 6. Unsafe usages can be further categorized either in calls using as argument (i) a variable (64.59%) or (ii) a complex Smalltalk expression (9.89%, referred to as Other). Most unsafe reading of object fields occurs in App projects (35.09%), while in System projects this feature accounts for 23.13% of all unsafe usages. This makes System projects extremely over-represented, as the ORF is 12.42. The other project categories are less represented, particularly Tests projects, which as nearly the same ORF as application (0.58 and 0.56, respectively). Most unsafe usages in category variable follow the pattern:

```
obj allInstVarNames do: [:ivar |
    (obj instVarNamed: ivar) doSomething]
```

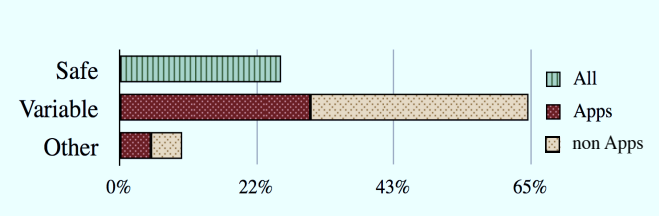

Fig. 6 Safe/unsafe usages of object field reads.

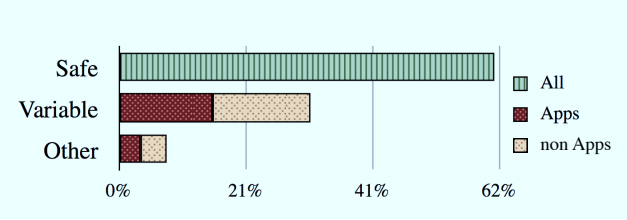

Fig. 7 Safe/unsafe usages of object field updates.

Object Field Update (1,441 calls, 560 unsafe, 269 in Apps) Writing and updating the values of object fields is the fifth-most used feature. Figure 7 gives the distribution of safe (61.14%) and unsafe calls. Unsafe calls are split in: Variable (31.16%), when the first argument is a variable; and Other (7.7%), when the first argument is a complex Smalltalk expression, such as a method call. Unsafe calls in applications make up 18.67% of the total (Table 3), while unsafe calls in the System category make up 12.91% of all field updates (ORF=13.29), for reasons similar to the uses of object reference updates. Here again, Tests are under-represented, with an ORF of 0.30, and Language extensions have an ORF of 0.84.

The following pattern is extremely common (664 or 46% of all calls, with 398 calls in Applications):

```
MyClass basicNew instVarAt: idx1 put: value1 ;
            instVarAt: idx2 put: value2;
           ...
            instVarAt: idxN put: valueN.
```

This code snippet creates a new object and initializes all its fields with predetermined values. Smalltalk provides the storeString method, which serializes the object in the form of a valid Smalltalk expression that, when executed, recreates the object in the same state; it is a relatively common practice to save objects as executable expressions that way. It is actually surprising that Tests do not use this feature more heavily.

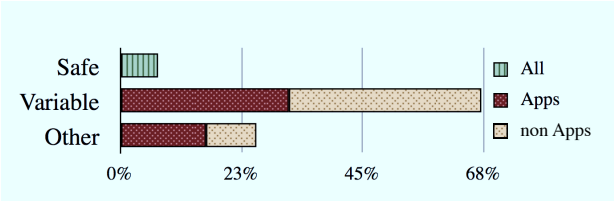

Fig. 8 Safe/unsafe usages of message sending.

Message Sending (6,048 calls, 5,623 unsafe, 2,874 in Apps) The most used dynamic feature accounts for 29.67% of all occurrences. Unfortunately, most of these usages (92,97%) are unsafe (Figure 8). This is not surprising: there is little value in calling a method reflexively if the message name is constant. Two thirds of all calls use as argument a local variable, and more complex Smalltalk expressions are used in one fourth of cases.

Table 3 indicates that almost half of the message sending feature occurrences (47.62%) are unsafe calls inside App projects. If Apps are —as usual— under-represented, the ORF of 0.61 is one of the highest. Tool projects follow with a quarter of all occurrences (26.57%, ORF=4.54); a possible explanation for that large over-representation is that tools often feature a UI, for which reflexive message sending is commonly used in Smalltalk—an example being the Morphic UI framework. System packages are also significantly over-represented (ORF=4.21), although the reason for that is less clear. The rest is split between the other project categories: Tests (5.92%) and Language-extensions (3.17%, under-represented).

Interpretation

- Supporting message sending is a priority: it constitutes almost 30% of dynamic feature usages; 60% of projects use it; nearly 93% of uses are unsafe. However, supporting message sending efficiently may be challenging. The state-of-the-art solution of Bodden *et al.* mixes enhanced static analysis with dynamic analysis to provide sufficient coverage [BSS+11].
- The other three behavioral features—object reference update and field accesses—are used infrequently. The exception is Systems projects, which do use them pervasively: in all three, Systems projects have an over-representation factor in excess of 9.

– Object field updates are 60% safe, due to their usage as a serialization mechanism. This contrasts with field reads, whose safe usages are only 25%; field reads are mostly used with a dynamically-determined instance variable name. Reference updates are much more challenging to support.

5.4 Structural Reflection

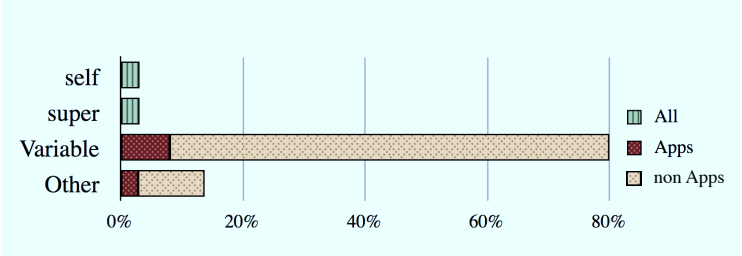

Fig. 9 Safe/unsafe usages of class deletion.

Class Removal (385 calls, 361 unsafe, 42 in Apps) Class removal is one of the lesser-used features. According to Figure 9, safe usages are in the minority (6.24%); calls with a local variable as receiver make 80% of the calls; more complex calls make the rest. It is also obvious that unsafe usages in applications are also a minority (10.91% of usages according to Table 3, and an extremely low ORF of 0.14), whereas System projects have the same number of usages (translating to a much higher ORF of 4.65). Tests provide the overwhelming majority of unsafe usages with 62.34%. This massive over-representation (ORF=27.70, the largest by far) ties up with the heavy usage by tests of dynamic class creation (which also had a very high ORF. 18.71). A common pattern in tests (208 instances, more than 80%), is to create a new class, run tests on it, and then delete it.

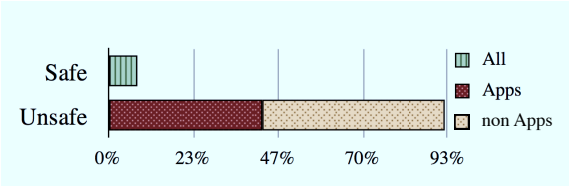

Fig. 10 Safe/unsafe usages of superclass updates.

Superclass Update (114 calls, 105 unsafe, 48 in Apps) This feature is the least used with just 114 occurrences, 0.56% of all dynamic feature occurrences. As shown in Figure 10, safe calls account for 7.89% while 42.11% are unsafe calls inside App projects; Tests are the heaviest users (ORF=7.93, 17.54%), followed by Tools (3.17, 18.45%) and Systems (3.05, 7.02%). Since tests often build classes to run test code on, it stands to reason that they would also need to specify their superclasses.

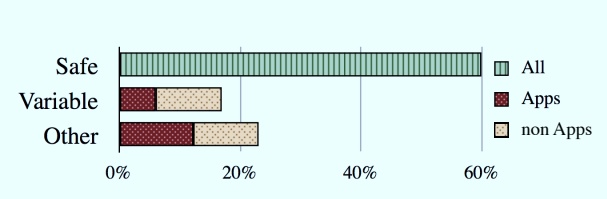

Fig. 11 Safe/unsafe uses of method compilation.

Method Compilation (2,296 calls, 918 unsafe, 419 in Apps) Method compilation is the fourth-most used feature, with nearly 2,300 of the 14,184 calls. A majority of the usages (60%) are statically known strings, and are thus safe (Figure 11). Of the rest, 17% hold the source code in a variable, while 23% are more complex expressions— *i.e.* a string concatenation, which represents 40% of complex expressions.

Applications feature a bit less than half of the usages (18.25%, ORF=0.55), and are hence under-represented (but not less than usual); on the other hand, Systems (ORF=6.32), and Tests (ORF=5.32) are over-represented. This behavior is similar to the one in class creation (although less skewed towards Tests), and has similar reasons.

In addition, we manually classified the safe method compilations that are known statically in trivial methods (returning a constant or a simple expression), getter/setter (returning/setting an instance variable), and arbitrary code. We found that the vast majority (75.91%) of the methods compiled were trivial in nature, while getter and setters constituted 7.55%, with the remaining 16.55% being arbitrary. Examples of methods classified as "trivial" follow:

```
ClassA>>one
  ↑ 1.

ClassB>>equals: other
  ↑ self = other.

ClassC>>newObject
  ↑ self class new.
```

Note that the code base we analyze contains 652,990 methods, so we can hypothesize that the number of statically defined methods vastly outnumbers the quantity of dynamically defined ones, but we cannot be sure of that fact without performing dynamic analysis.

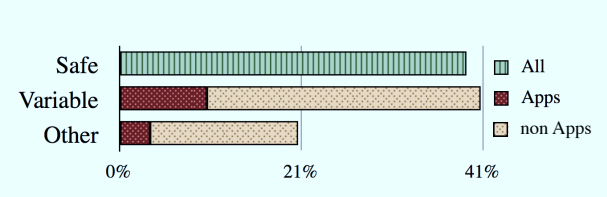

Fig. 12 Safe/unsafe uses of method removal.

Method Removal (437 calls, 266 unsafe, 58 in Apps) Method removals are used relatively sparsely, and unsafe uses are much more prevalent in Tools, Systems, Language extensions, and Tests than in Apps, as shown in Figure 12. Safe calls make up 39.13% of all the calls; unsafe calls with a variable 40.73%; complex unsafe calls 20.14%. We see in Table 3 that Apps are clearly under-represented (a low ORF of 0.26): System projects on the other hand are very over-represented (ORF=12.33), as are Tests, to a lesser extent (ORF=7.05); Tools follow (ORF=3.94). The low ORF in Apps is similar to Class Removal.

Interpretation.

- Besides method compilation, structural reflective features are rarely used. In addition, the vast majority of application projects does not use these features. It appears that support for superclass update, class removal, and method removal does not need to be as urgent/efficient than other features.
- Class removal seems to be quite correlated with class creation, which is expected. Table 3 shows that all project categories show similar numbers of usages (with Apps creating more classes than they remove); the total number of calls are also similar (385 vs 420). The over-representation factors are also quite similar.
- Changes to methods (method compilation and removal) have a large proportion of safe usages (40 to 60%). However, the significant proportion of unsafe uses means that support for method compilation cannot be neglected in the design of static analysis tools.
- Tests are the largest users of structural reflection, as they are heavily represented in all features. System projects follow (2 out of 4)

5.5 System Dictionary

Reading (4338 calls, 2233 unsafe, 688 in Apps). Reading the system dictionary is the second most used dynamic feature and accounts for 21.41% of all usages. Approximately half of usages are unsafe (51.5%, see Figure 13). Most unsafe usages occur by passing a local variable as argument to read objects stored in the dictionary (24.55%); more complex Smalltalk expressions are used in 26.92% of all unsafe usages.

As indicated in Table 3 the system dictionary is mostly read in App projects (15.9%), followed by System (11.36%), Tools projects (11.00%), and Tests (10.79%). Given the relative size of these projects categories, reading the system dictionary is overly represented in these categories compared to App projects (ORF=0.37). The trend we have been seing earlier, with Systems and Tests often being heavy users, is confirmed (ORF=8.83 and 8.73, respectively)

We found two common patterns: (1) using the system dictionary to check the existence of a class (60% in safe usages), and (2) accessing the class reference through the system dictionary (30% in safe usages). Below are some examples of the previous pattern:

```
Smalltalk at: #MyClass
    ifPresent: [...do something...]
    ifAbsent: [...do something else...] .

myClassRef := Smalltalk at: #MyClass.
```

These patterns are not the monopoly of safe usages, they are also present in unsafe usages in similar proportions.

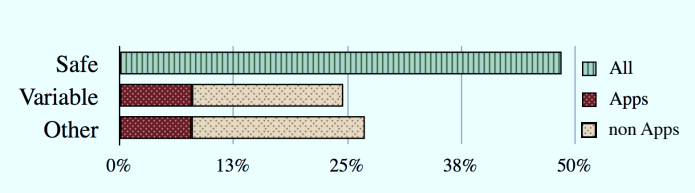

Fig. 13 Safe/unsafe uses of Smalltalk readings.

Writing (404 calls, 78 unsafe, 20 in Apps). Writing in the system dictionary is much less common than reading from it. By accounting for 1.99% of all dynamic feature usages it is one of the less-used features. Around a fifth of its usages are considered to be unsafe (19.3%). Similar to previous features, unsafe usages are split in two groups: passing a local variable to the dictionary to identify the object to be written (12.87%); and passing a complex Smalltalk expression to the dictionary (6.44%)(cf. Figure 14).

Unsafe writing to the system dictionary mostly occurs in Tests (6.19%), subsequently followed by Apps and System projects, both with a share of 4.95%. Both System projects and Tests are overly writing to the system dictionary compared to the relative sizes of these project categories (ORF=10.25 and 13.36, respectively).

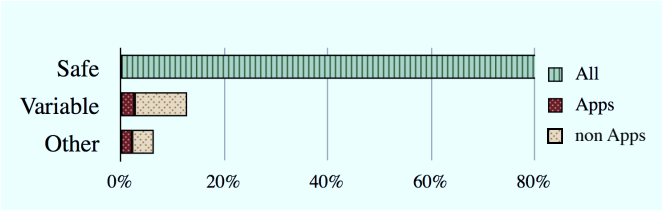

Fig. 14 Safe/unsafe uses of Smalltalk writings.

Aliasing (207 calls, 207 unsafe, 58 in Apps). Aliasing the system dictionary is even less common than writing to it, making it the second least used of all analyzed dynamic features behind superclass update, only accounting for 1.02% of all dynamic feature usages. However, as mentioned in Section 4.3.4, we consider all occurrences of aliasing to be unsafe. As Figure 15 illustrates, almost all usages of system dictionary aliasing happen by passing the dictionary as parameter to a method (89.86%); 7.7% are local aliases, which appear superfluous, and 2.4% are aliases to fields.

Most occurrences of system dictionary aliasing are in App projects (28.02%), followed by Tools (27.54%), Tests (24.15%) and System projects (15.46%). Hence, as with most other dynamic features, Tests (ORF=10.06) and System (ORF=6.18) projects are over-represented (and Tools, to a lesser extent).

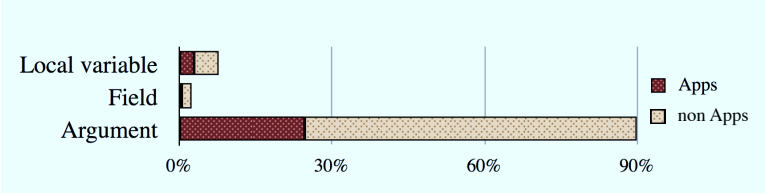

Fig. 15 Safe/unsafe uses of Smalltalk aliasing.

Interpretation.

- Smalltalk reading is the second most used dynamic feature with 21% of all usages. Testing for the existence of a class or obtaining a reification of it is also supported by Java, confirming assumption 3.
- Smalltalk writing is used infrequently. However more than 80% are safe usages. Tests and System applications are over represented (see Table 3), with 6.19% and 4.95% respectively.
- Smalltalk aliasing is the second least used dynamic feature. A few usages (less than 10%) are direct aliasing to local variables, which could be avoided through straightforward substitution. Other usages would require inter-procedural analysis to track down how the dictionary is used afterwards.
- Tests and Systems projects are largely over-represented for every feature.

6 Discussion

We discuss whether each of the assumptions and research questions we mentioned in the introduction is valid or not, and provide guidelines for each feature we studied.

1. **Dynamic features are rarely used.** Dynamic features were found to be used in a small minority of methods—1.76%. Assumption 1 is validated.
2. **Dynamic features are used in specific kinds of projects.** We conjectured that core system libraries, development tools, language extensions, and tests, were the main users of dynamic features. If these categories use on average much more dynamic features than regular applications (the 17% of projects make 56.57% of unsafe usages), the latter still makes up nearly half of all the unsafe usages. Going further, a statistical test showed us that there were more usages of dynamic features in projects not classified as applications but we found the effect size to be small. As such, assumption 2 is validated, albeit with a smaller effect than expected. If we look at categories individually, the over-representation factors of Table 3 show us that Systems and Tests are by far the largest users of unsafe dynamic features, Tools are also over-represented, whereas Language Extensions are close to normal, and Applications are systematically under-represented.
3. **The most used dynamic features are supported by more static languages.** The three most used features, reflective message sending, reading the system dictionary and instance creation, are supported by the Java reflection API, validating assumption 3.

4. **Some usages of dynamic features are statically tractable.** We found that 4 features (instance creation, object field updates, method compilation, and smalltalk writing) have a majority of safe uses. Three others (object field reads, class creation and method removal) have a strong minority (more than 30%) of safe uses. Assumption 4 is validated.

Even if dynamic features are used in a minority of methods (1.76%, validating assumption 1), they cannot be safely ignored: a large number of projects make use of some of the features in a potentially unsafe manner. We review each feature on a case-by-case basis, and in order of importance (as determined by overall usage of the features).

- **Message sending** is the most used feature overall, with 60% of projects using it and a majority of unsafe uses. Supporting it is both challenging and critical.
- **Smalltalk reading** represents the second most used feature, with almost 40% of projects using it. Half of system dictionary reads are unsafe. Supporting it is crucial and also challenging.
- **Instance creation** is used by 40% of the projects, but can be considered mostly safe if a notion of *self types* is introduced, as in Strongtalk [BG93].
- **Method compilation** is used in an unsafe manner by a little over 20% of the projects, and as such also needs improved support.
- **Object field reads and updates** are the last of the features that has a somewhat widespread usage. Although reads usages are mostly unsafe, updates are mainly safe.
- **Class creation and removal** are heavily used in tests, but class creation is used in applications as well.
- **Smalltalk writing** is rarely used, less than 8% of projects used it. Moreover, four of five system dictionary updates are safe.
- **Object reference updates** are somewhat problematic, as nearly 45% of the usages are in applications. Supporting such a dynamic feature is also a challenge.
- **Method removal** has a large number of safe uses, and is primarily used outside applications.
- **Smalltalk aliasing** are problematic, although occasionally used (7.9% projects). Supporting it is not vital, but programmers must take account of them.
- **Superclass updates** is a somewhat exotic features whose usages are few and far between.

As a rule of thumb, we conclude that message sending, system dictionary reads, method compilation, instance creation, object field reads and updates, and to a lesser degree also class creation, system dictionary updates, and object reference updates, are particularly important dynamic features that static analysis tools, type systems, and compiler optimizations should support well. Of less importance are class and method removals, smalltalk aliasing and superclass update since they are rarely used in an unsafe manner in application projects, nonetheless language designers cannot afford to completely ignore them.

7 Why do developers resort to using dynamic features? (and what to do about it)

Beyond a state of the pratice on the usage of dynamic features, we also wish to understand why developers end up using them. Even if Smalltalk is a language where these features are comparatively easier to access than most programming languages, developers should only use them when they have no viable alternatives, as they significantly obfuscate the control flow of the program, and add implicit dependencies between program entities that are hard to track (*e.g.* a dynamic invocation of a method does not show up in the list of users of the methods). As such, any usage of a dynamic feature that is superfluous, or that can be removed at a moderate cost, should be removed, or at least carefully considered for removal.

In this section, we answer the two following research questions:

RQ1: What are the principal reasons why developers use dynamic features?

RQ2: Are certain types of dynamic feature usages refactorable? Can they be removed, or can unsafe usages be refactored to safe ones?

However, these questions can not be addressed by a large-scale quantitative analysis as we performed above; each dynamic feature usage needs to be manually inspected to determine the reason of its existence, and whether it can be removed. Thus, we had to reduce the scope of the study.

7.1 Methodology

Sample size From a total of 1,000 Smalltalk projects, we collected 20,387 occurrences of dynamic features (Section 5). These occurrences represent our initial data set. Due to the fact that manual inspection is required, we extract a representative sample set that we inspected further. We establish the sample set size (n) with the formula [Tri06]:

$$n = \frac{N \cdot \hat{p}\hat{q} \cdot (z_{\alpha/2})^2}{(N-1) \cdot E^2 + \hat{p}\hat{q} \cdot (z_{\alpha/2})^2}$$

We keep the standard confidence level of 95% ($z_{\alpha/2}$) and the standard 5% margin of error (E). The proportions source code that is refactorable ($\hat{p}$), and of source code that is not refactorable ($\hat{q}$) are unknown a priori, so we consider the worst case scenario ($\hat{p} = \hat{q} = 0.5$, *i.e.* $\hat{p}\hat{q} = 0.25$). Statistically speaking, the population of our data set (20,387 dynamic features usages) is relatively small. Because of this, we include the finite population correction factor directly into the formula (N).

The sample size computed from the formula is 377. As we do not have previous information about the distribution of refactorable source code, we employ random sampling without replacement to select a reliable sample set from our initial data set. Table 4 shows the obtained distribution by feature. The first column is the name of each analyzed feature; the second represents the data set size; and finally, the third shows the sample size.

Dynamic Feature	Data size	Sample size
Instance Creation	2,732	58
Class Creation	420	10
Object ref. update	311	5
Object field read	1,254	23
Object field update	1,441	14
Message sending	6,048	125
Class removal	385	8
Superclass update	114	5
Method compilation	2,296	34
Method removal	311	3
Smalltalk reading	4,338	79
Smalltalk writing	404	8
Smalltalk aliasing	207	5
Total	20,387	377

Table 4 Per-feature distribution of the sample set size

Inspecting dynamic feature usages In order to understand the rationale for the usage of a given dynamic feature, and to determine whether and how a given feature is refactorable, we employed partial program comprehension techniques ([ES98]). Our infrastructure allows us to browse the source code of a dynamic feature usage, and, when necessary, browse the source code of the project using it, as well as inspecting the list of methods calling the one where the dynamic feature is used. As such, each code fragment was inspected for as long as it was deemed necessary in order to understand the reasons behind the usage of the dynamic feature, and whether it was refactorable. Three of the authors inspected the sample; each source code element was inspected by at least two authors.

To abstract away from the raw information in the sample, we elaborated a classification of the types of dynamic feature usage rationales in categories. We performed several iterations steps where the authors would examine individual examples in the sample, and either assign it to a category, or create a new one. After each step, we discussed the classification and altered the definition of the categories. This was performed until we arrived to an agreement on a stable classification. The classification was stored in an online repository, which was used as support for the discussion. We describe the different categories in Section 7.2 below.

In addition to classifying dynamic feature usage rationales, we also report, though less systematically, on the refactorability of dynamic feature usages. In particular, we pay attention to the locality of refactoring when possible. This is because refactorings that are local to a given method body are much easier to perform than global ones, such as modifying all call sites of a given method.

Finally, we also looked at prominent application domains where dynamic features are used, as reported in Section 7.3. We mention these when a particular kind of application is overwhelmingly represented in a given category.

7.2 Categorizing user intention when using dynamic features

In this section, we describe the eight categories we found in the inspection of the sample set, mining for user intent. We start with the most generic categories, that

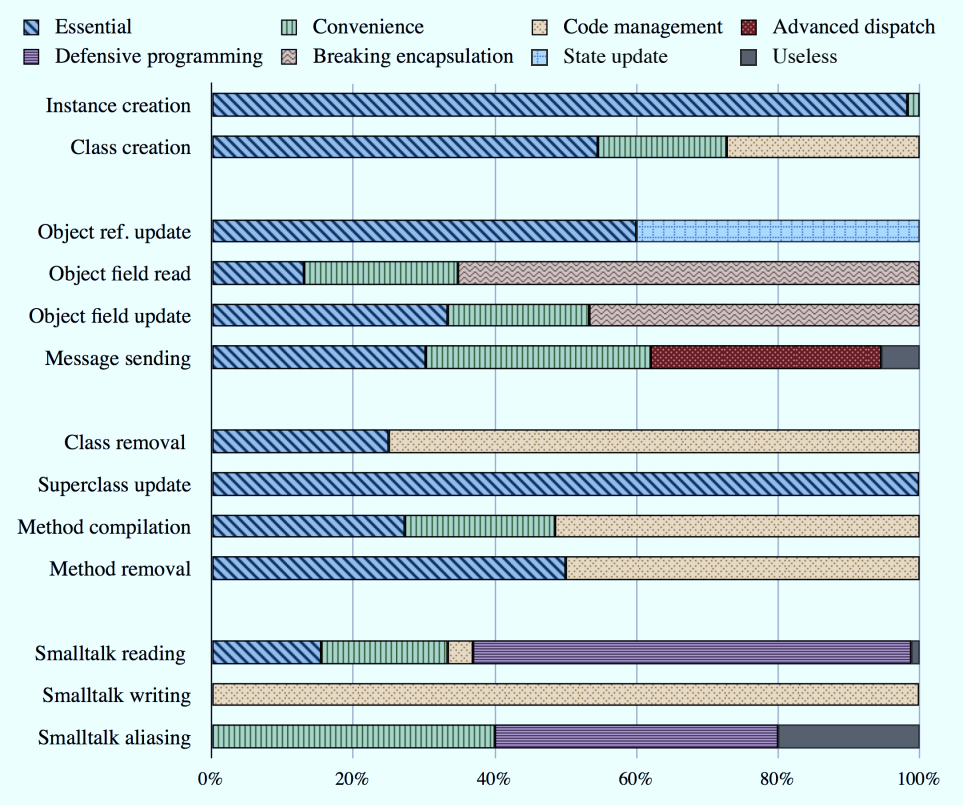

Fig. 16 Per-feature distribution of user intention.

are significantly present in several features. We then introduce categories that are related to some specific features. We explain each category, describe some interesting cases extracted from the samples, and discuss the potential for refactoring of the different scenarios. We end this section with a brief discussion of the false positives detected in our analysis.

Figure 16 summarizes our results, showing the per-feature distribution of user intention categories. For each feature, we count relative usages (percentage) discarding false positives.

7.2.1 Essential usages

Usages of dynamic features classified as *essentials* are intrisic usages of the dynamic feature. They constitute 37.5% of all dynamic feature usages—more than a third. All but two categories of dynamic features (Smalltalk writing and aliasing) have essential usages.

For instance, a remote communication framework needs to perform dynamic method invocation based on the instructions received from a socket. Hence, the original functionality of dynamic method invocation is embedded in a *wrapper*

that refines it to specify a communication protocol, but still uses the original functionality deep down. This category is present in almost all features.

In our analysis, we found that programmers wrap essential dynamic features usage within code of their own to alter their behavior, *e.g.* adding pre/post-conditions, or providing extended behavior such as the remove communication example mentioned above. Additional examples include essential dynamic feature usages aimed at facilitating debugging (logging), and wrappers around object field accesses in order to mirror changes made to an object in an object database. Language extensions (*e.g.* a dynamic component model on top of Smalltalk objects) also make use of dynamic features in such a fundamental manner.

In general, these usages cannot be refactored, because the possibilities to take into account are very large, even potentially infinite (*e.g.* they could depend on user input). Even when the usages may be refactored theoretically, in many cases it is at a prohibitive cost. For instance, removing a wrapper around an object field access in an object database could be achieved only at the cost of modifying every accesses to the state of every object susceptible to be stored in the database. Likewise, reflective method invocation in a remote communication framework could be replaced by a mechanism similar to Java RMI, where remote interfaces are declared statically and proxies are generated by the compiler. Proposals to alleviate the burden of RMI programming in Java are, unsurprisingly, based on reflective invocation.

Occasionally, some of these instances can be refactored. A particular example we found was a wrapper used in debugging that executed a given method, but only if it belonged in a predefined (and small) set of allowed method. Since the number of possibilities was small (only 4), the cases could be enumerated and the dynamic feature usage removed.

Beyond this wrapping behavior, we found two other categories of essential usages. The first is the overwhelming majority of dynamic class instantiation (98.3%—all but one case). These are essential because they are the very means by which objects are created in the language; however, as we mentioned previously, self types [BG93] can be used to make these usages safe.

The second one is test code that explicitly tests dynamic features. Obviously, removing it would negatively impact the test coverage of the projects, so these usages can not be parted with either, nor can they be refactored away.

7.2.2 Convenience

Reflective features are powerful, providing flexibility to programmers that can at times be abused. Since Smalltalk makes it extremely easy to call a method based on its selector (a symbol), iterate over sets of methods or instance variables, programmers often use these features to shorten repetitive source code, sometimes at the expense of readability. Convenience usages are pervasive: they are the second-most important category, just shy of 20% of the usages of dynamic features; 8 out of 13 dynamic features are used as such; of note, 32% of dynamic method invocations are classified as convenience.

An example is the following, which chooses a message to send based on a condition:

```
classesSelected := classesSelected
   perform: (aBoolean ifTrue: [ #copyWith: ] ifFalse: [ #copyWithout: ])
```

```
    with: (self classes at: anInteger ifAbsent: [ ↑ self ]).
```

This could be rewritten as the slightly longer, but far more legible:

```
arg := self classes at: anInteger ifAbsent: [ ↑ self ].
classesSelected := aBoolean
    ifTrue: [classesSelected copyWith: arg]
    ifFalse: [classesSelected copyWithout: arg].
```

There are surprisingly many of these types of usages in our sample, including a similar one where a class is similarly chosen based on a boolean condition, and is then instantiated.

Other examples include specifying a list of methods in an array, and then iterating over it, executing a method at each step of the iteration. We found several test cases that are organized in this fashion, where common behavior before and after the test is executed in the body of the loop. Some programs make use of the fact that Smalltalk methods are classified in categories, in order to execute all the methods belonging to a certain category. A particular example of this scenario executes all the methods in a given category, and returns the list of the returned values. As it turns out, each method actually returns a constant string, making the set of dynamically-executed methods equivalent to returning an array of strings (with, among others, the added benefit that an array of string is immune to errors in classifying methods in the wrong category). Another example implements an "undo" mechanism by setting up an array of method names of undo methods. To undo an action passed as parameter, it computes the index of the undo action and retrieves the associated method in the array, to execute it. Of course, such a scheme is sensible to the order of the methods in the array.

The same type of behavior is also common in order to iterate over instance variables names, or classes names. Unlike the wrapper case, the majority of these usages feature a small number of possibilities, and are possible to refactor locally, by simply enumerating the handful of possibilities. A minority are a bit more challenging to refactor, as they involve scattered modification (*e.g.* converting a set of dynamically-invoked methods to an array of strings).

7.2.3 Dynamic code management

This category deals with the runtime creation of new code entities (classes and methods), their modifications, and their disposal. Smalltalk provides free access to the compiler as a class of the system, in the same manner as many dynamic languages provide an "eval" function (*e.g.* Javascript [RLBV11]). As such creating and deleting classes and methods is common, as we have shown above. Dynamic code management is the third widespread category in terms of versatility, with 6 out of 13 dynamic features. Overall, 8.6% of all usages belong to dynamic code management, including all of Smalltak writing, and the majority of class removals, and of method compilations and removals.

The reasons for this are multiple:

- Some tests need to create classes and methods as part of their fixtures, and delete them afterwards.
- Some systems implements prototypes (as in Javascript), by defining one class for each would-be prototype.

– Others generate basic or more elaborate methods from parameters they are given. In our sample we found several implementations of auto-generated getters and setters for instance variables; needless to say, a standard implementation for this would be useful. Other cases are much more elaborate, involving iterations over a set variables to either generate several methods, or several statements in a method.
– A lot of the code generated in fixtures for test cases consist of constant methods. We found other uses for constant methods, namely patching the code of another application in order to fix a known issue. This is of course a brittle strategy, as it ignores the possible changes in the other application.
– Yet another use of code generation is as an ad-hoc serialization mechanism. Using structural reflection, each Smalltalk object can generate a piece of code that, when executed, returns a copy of the object itself. This produces sequences of calls to reflective field accesses, each of them setting up one field of the object. This expression can then be compiled in a method of a class in order to re-create the object at anytime, and is used in test fixtures.

Cleanup code—removing code entities—is also somewhat prevalent (especially, as we remarked above, in test code to remove previously generated classes or methods). Other cleanup operations remove all the subclasses of a given class, which is known to contain generated code. A variant of this is the dynamic recompilation of methods, replacing the statements that they contain with an empty method body. We found this in a handful of cases, when a program dynamically changes from development mode to deployment mode: methods logging behavior are altered to do nothing instead. Of course, a similar behavior could be achieved in a program that keeps its development/deployment mode more explicitly as a boolean status, which is checked by the methods above. Barring this particular case, and replacing ad-hoc serialization with a more robust serialization mechanism, refactoring these is difficult.

7.2.4 Advanced method dispatch

A large part of the usages of dynamic method invocation are alternative method dispatch mechanisms—32.6%, or 11% of all the dynamic feature usages. We found the following:

– An object stores another object and an associated method name, for later invocation. This is very common in UI frameworks, where the method is expected to be called when a UI action takes place. This is a lightweight variant of the Observer and Listener design patterns. The standard UI framework available for Pharo, Morphic, provides a very simple way to set up UI notifications as follows:

```
button on: #eventName send: #methodToExecute to: receiver
```

In this case, two symbols are passed as argument, one being the name of the event to react to (such as button click, double click, etc.), and the other the name of the method to call on the third argument. If the event handler determines that the message needs an argument, it will also provide the source of the event as an argument. As a side note, this example shows that passing

method names through symbols is problematic, as in this example, one symbol is a method name, while the second is not.
– An object performs a dispatch on itself based on a method name it receives as argument, or a method name that is obtained by processing the argument. This is again a lightweight variant of other design patterns, *e.g.* the Visitor or Strategy patterns. Examples that we found are an interpreter class, which interprets a stream of bytecodes represented as symbols, and dynamically invokes the method bearing the same name as the bytecode. Other examples are more complex, where the method name is constructed from one parameter (adding a prefix or suffix to match the name of the method), or two parameters (implementing an ad-hoc double-dispatch mechanism).

Of course these are quite fragile, as they are very sensitive to renaming: only a runtime error will give an indication that a method has changed, or that a new type of objects need to be handled. The more heavyweight versions of these patterns are not sensitive to these errors occasioned by renaming, but may trigger an explosion of small "glue classes", such as the anonymous Java listener classes that are omnipresent in Java UI code. Barring additional support, such a refactoring is not trivial.

The well-known lightweight solution for these programming idioms is to use first-class functions, as found in languages such as Lisp, ML, Haskell, or Scala. First-class functions can be type checked, while method names as symbols carry no type information. Surprisingly, this is also possible in Smalltalk, because Smalltalk code blocks are essentially anonymous functions! However, their syntax makes them a bit more verbose than symbols, which make programmer opt for the more concise alternative in many cases. A straightforward iteration on a collection is achieved by:

```
collection do: [:eachInteger | eachInteger squared]
```

But many Smalltalk dialects also support the more concise:

```
collection do: #squared
```

Since dynamic message sending achieved by passing names of methods is the most used feature in our corpus, it seems worthwhile to explore more robust mechanisms than plain symbols. Such a refactoring would be tedious, but not complicated.

7.2.5 Defensive programming

Due to the nature of Smalltalk as a dynamic language that, in addition, relies on whole system images, one is never sure if the environment contains the classes necessary for the correct execution of a program. Recall that accessing a class is done via the system dictionary; if the class is not defined, a null pointer will be returned. Defensive programming is distributed accross two features only (Smalltalk reading and aliasing) but is prevalent in these (61.9% of reading; 40% of aliasings; 14.17% overall).

Most code just assumes that all classes are present, but some code actually checks in the system dictionary if the required classes exist. Upon discovering that a needed classes does not exist, we found programs that react differently: some report

the error to the user; some return a default value computed without the missing class; some use an alternative class instead; and finally, some dynamically download and install a given version of the package that contains the class. Introspecting the system dictionary cannot be refactored in the context of a dynamic environment like Smalltalk; a better way to solve the problem would be to improve the way Smalltalk handles package dependencies, either at the language or infrastructural level.

7.2.6 Breaking encapsulation

Reflective field accesses are sometimes used to circumvent encapsulation, because the API of an object does not permit access to its fields. They constiture 5.77% of all the usages, but 65.3% of object field reads, and 46.7% of object field updates.

Breaking encapsulation is common when a test needs access to a field but the programmer does not wish to expose the field for the rest of the world. Other instances of this usage exists outside of tests, for which the only solution would be to extend the API to permit access to such field. Another solution would be to extend the language to better support privileged access to the internals of an object. Recall that Smalltalk is strongly encapsulated in the sense that fields are not accessible from outside an object. On the other hand, all methods are. Java or C++ support different visibility mechanisms used to control encapsulation at the level of both fields and methods. Even with such mechanisms, reflective access is needed at times, because they are not very flexible. Encapsulation policies are a flexible alternative to address this issue [SBD04].

We found one instance where a specific field was accessed reflectively, despite the presence of the accessor in the source code. In fact, the accessor was also performing additional computations, that the calling code did not wish to take place. A source code comment stated: "Ugly, but fast", indicating that the user was perfectly aware of the problem, but chose to ignore it on purpose.

Most examples we found break encapsulation in a systematic manner, iterating over all the fields of an object, sometimes recursively. This is the case of generic object copiers, and of some persistence frameworks that need to write objects on disk. It is also the case of the ad-hoc serialization mechanism mentioned above, which generates source code that, when evaluated, creates a copy of the object. These would be prohibitive to refactor, as every object would need to implement its own version of copy or serialization methods.

7.2.7 State update

Since object reference updates is a seldom-used feature, the corpus that we inspected does not feature a lot of these usages—only 5. We think that it is worth mentioning that out of these 5, 2 made the receiving object itself change its class before resuming operation.

It is interesting that this kind of usage of reference update directly corresponds to *state update* in typestate-oriented programming languages [ASSS09]. Such languages support stateful resources with state-specific behavior and representation. As such they are the language-level equivalent of the State design pattern. Tracking down such state changes statically is not trivial, but is feasible, even in a gradual manner [WGTA11].

7.2.8 Useless

A small, but still noticeable categories of usages (2.36%) were truly useless usages of dynamic features, where the usage did not appear to have any kind of benefit over its non-dynamic counterpart.

For instance:

```
(html effect id: id; perform: #appear)
```

Is strictly equivalent to:

```
(html effect id: id; appear)
```

We conjecture that these usages slipped in when a programmer reused and adapted a code fragment found somewhere else, and did not notice the dynamic usage was unnecessary. All of these can be safely replaced with their non-dynamic equivalent.

7.2.9 False positives

The final category we found were false positives, demonstrating the limit of our previous automated analysis. We fortunately only found barely more than a handful of them:

- We found seven usages of the Smalltalk system dictionary as a mean to define and alter global variables instead of storing classes. When programmers need to share state accross large portions of their programs, they usually instantiate the Singleton design pattern, but the Smalltalk dictionary offers a "quick and dirty" alternative.
- One usage of the method removeFromSystem: (class removal) was calling a similarly named method that had a wholly different purpose. It appears that the original method name is not specific enough and can be overloaded.
- One usage of the method superclass: was calling a method with a similar intent, but which did not end up using a dynamic feature; this was a call to an object that mirrors an actual Smalltalk class—as part of a type inference system—but that does not impact the actual class.
- Finally, one call to compile: was found in "junk code"—obviously incorrect code that was never meant to be executed, but part of a fixture for a test case. Often, these fixtures are compiled dynamically and removed when the test finishes, but this one was statically defined—in a method named "foo".

This totals 10 false positives, or 2.65% of the sample (recall that the percentages reported above exclude false positives). This low figure gives us strong confidence in the results of the preceding section.

7.3 Types of applications

In the sample we inspected, we noticed that some types of applications had a particular frequency of usage of dynamic features, as well as a particular distribution of usages.

7.3.1 Testing

As we already hinted at earlier, unit tests constitute a large proportion of all the usages of dynamic features. Unit tests have several reasons to be heavy users of dynamic features:

- Testing the dynamic features themselves. In order to achieve a good code coverage, all the functionality of a system should be tested, including the dynamic features.
- Generation of test objects. In some cases, tests need to generate classes and methods as part of their behavior. In addition, these objects need to disposed of once testing is over. The easy access to the compiler makes it trivial to do so, making this a very common occurence.
- Bypassing the public API. White-box testing is done with the knowledge of how the code under test works. A test may need to set up an object in a way that should not be possible in the public API, such as accessing a field that should not be accessed by regular clients. In these cases, several test cases access the instance variable by name, or by index. In other cases, objects that are fixtures of test cases are saved in source code, and restored by accessing their fields directly

All these factors make it often difficult to refactor test cases so that they do not use these features. It can be argued that this is not as problematic, as test cases are not part of the core of the application. However, the maintainability problems encountered while co-evolving application and test code are not considered in this study; further analysis is required to shed light on this subject.

7.3.2 Other types of applications

We noticed 3 other types of applications that were prominent, and with specific usage patterns.

- UI applications make heavy usage of dynamic method invocation as a lightweight form of an event notification system; the usage of this idiom is so pervasive that it has spread to other types of event handling systems, such as the Announcements framework for Pharo.
- Several frameworks that communicate with databases, or implement object databases, make heavy usage of serialization and de-serialization of objects. In order to do so in a generic way, they reflect on the structure of the objects they need to serialize, using field access reads and field access writes.
- Low-level system support code uses object field reads and writes to implement copy operations, saving the state of the system to disk, and convert numbers and strings from objects to compact bit representation.

7.4 Summary

We found a variety of reasons why developers use dynamic features, in a spectrum ranging from essential, fully-justifiable reasons, to ones that can, for all intent and purposes, be considered useless.

- Some of these dynamic feature usages are unavoidable, as they are a direct mapping to the dynamic feature itself (such as method dispatch in a remote communication framework, or tests of the dynamic feature themselves). Removing or refactoring these usages is most of the time impossible.
- Other usages point at limitations in the programming language (lack of first-class methods, privileged access to the private attributes of an object, objects changing state). Beyond extending the language, other solutions to most of these problems exist in the form of design patterns. Refactoring these would often be costly, however, and would trade one kind of complexity with another.
- Yet another class of usage deals with the generation and removal of new source code entities. There is no clear guidelines to address these cases, however.
- Defensive programming code idioms are in the majority spawned from the nature of Smalltalk as an image-based programming system, with historically little support for handling dependencies between packages.
- Finally, the mere availability of such powerful dynamic features make programmers abuse them in the name of conciseness. To save a few lines of code, programmers will go to considerable lengths, at times producing code barely shorter, but much harder to understand—dubious gains, to say the least. Fortunately, most of these usages can be easily removed as they are not critical to the program.

8 Threats to Validity

8.1 Threats to construct validity

We classified the projects in categories in order to investigate whether certain categories use dynamic features more often. We may have misclassified some of the projects. However, three of the authors individually classified all projects and discussed classification differences before coming to an agreement for each project.

Our list of methods triggering dynamic features is not exhaustive. Our criteria for inclusion of a given method was whether it was "standard", *i.e.* part of the Smalltalk-80 standard API. Non-standard methods triggering dynamic features were left out, however their usage is limited (for instance, there are 64 usages of the ClassBuilder class instead of the subclass: selector, and only 7 in regular applications).

A risk we had was to perform an analysis on a corpus that contained a high proportion of false positives. While performing our manual analysis of the sample, we kept track of the number of false positives we found, in order to gather an estimate of the false positive in our corpus. We found 10 false positives in our sample of 377 dynamic feature usages, or 2.65%, which we judge to be acceptably low for our general findings to hold.

We only use static analysis as it would be impractical to perform dynamic analysis on 1,000 projects. If costly, dynamic analysis would allow us to know the frequency with which code is executed: some parts of the source code could actually be dead code, while others may be hotspots taking the major part of the execution time. As it stands, we cannot be sure whether the code triggering dynamic features is actually executed; some dynamic feature usage could on the other hand be executed very often. In addition, Smalltalk features a system

dictionary—a dictionary bindings names to classes—that we did not include in the study, as it would require dynamic analysis to differentiate this specific dictionary from the other dictionaries used in the code.

Our manual analysis of the 377 feature usages in our sample involves partial program comprehension, and a degree of subjective judgement. As such, it is possible that mistakes were made in the classification of the source code fragments, or that the intent of some of them was misinterpreted. We tried to avoid that by having at least 2 of the authors reviewing each of the instances, and inspecting closely the ones where the two judges disagreed, before taking a final decision.

8.2 Threats to external validity

Our study includes only open-source projects for obvious accessibility reasons, hence we cannot generalize the results to industrial projects.

We only consider projects that are found in the Squeaksource repository. Squeaksource is the *de facto* standard source code repository for Squeak and Pharo developers, however, we cannot be sure of how much the results generalize to Smalltalk code outside of Squeaksource, such as Smalltalk code produced by VisualWorks users.

Our corpus of analyzed projects only contains Smalltalk source code. Our hypothesis is that Smalltalk code, with the ease of use of its reflective features, constitute an upper bound on the usage of dynamic features. This assumption needs to be checked empirically by replicating this study on large ecosystems in other programming languages.

We selected the top 1,000 projects based on their size to filter out projects that might be toy or experimental projects. We believe such filtering increases the representativeness of our results, however, this might also impose a threat.

8.3 Threats to internal validity

To distinguish pure application projects from other type of projects, we categorized projects in categories. Results show that application projects use dynamic features less often than most other project categories. However, code categorized in the crosscutting category *Tests* for instance might use more or less dynamic features depending on the project the test code belongs to rather than on the fact that it is test code. There might be other reasons why projects categorized as applications use dynamic features less often than explained by the categorization in application and non-application code.

8.4 Threats to statistical conclusion validity

To determine whether specific kind of projects such a system libraries or development tools use dynamic features more often than regular applications (hypothesis 2), we applied statistical tests to compare application projects to other kind of

projects (cf. Section 5.1.2). These tests are biased due to the fact that application projects are over-represented (83% of all analyzed projects belong to this category) than projects of the other categories. The applied t-test to a certain degree accounts for the unequal sample sizes of application and non-application projects, however, the over-representation of application projects clearly imposes a threat to conclusion validity.

To account for that in the later discussion, we defined an over-representation factor (ORF), that we use as support in the discussion on the unsafe dynamic feature usages. The ORF explicitly takes into account sample size, and allowed us to discover that for each individual dynamic feature, projects classified as Applications had less usages of unsafe dynamic features than expected. In contrast, Tests and Systems projects often had five times as many unsafe dynamic feature usages that one would expect. Due to the numerous tests that would have been involved (and potential type I errors associated), we did not test for statistical significance whether each individual feature was significantly more represented in each category of projects compared to applications.

9 Conclusions

We performed an empirical study of the usage of dynamic features in the 1,000 largest Smalltalk projects in the Squeaksource source code repository, accounting for more than 4 million lines of code.

We assessed the veracity of four high-level assumptions on the usage of dynamic features: Dynamic features are not used often (yet enough to be problematic); they are used more in certain kinds of applications than others; the more popular dynamic features are replicated in more static languages; and some of the dynamic feature usages are statically tractable.

We also analyzed in details the usage of each of feature, producing a list of features ordered by the importance of their support for applications. Some are critical (message sending, system dictionary reading, instance creation, method compilation); others less so.

Subsequently, we performed a qualitative analysis of a representative sample of 377 usages of dynamic features, in order to understand the rationale behind each feature usage, determining whether it was possible to remove these usages, and to pinpoint limitations of the language that, if addressed, could make it possible to avoid relying on dynamic features.

We found that, if a large portion of the usages of dynamic features are genuine usages that can not be refactored, others work around limitations of the programming language. In the absence of changes to the language, these could be replaced by more standard solutions to the same problems, albeit more complex. Finally, a significant minority of the usages are superfluous, and could be removed at a moderate cost to the programmer.

The usage of dynamic features is a double-edged sword. We hope that our results will incite the community to study the state of the practice in other languages as well. Further, we hope that our results—and the results of subsequent studies—will motivate practitioners to address the issues we encountered.

10 Acknowledgements

We thank the anonymous reviewers for their helpful comments. Romain Robbes is partially funded by FONDECYT Project 11110463, Chile. Éric Tanter is partially funded by FONDECYT Project 1110051, Chile. David Röthlisberger is funded by the Swiss National Science Foundation, SNF Project No. PBBEP2_135018.

References

[ASSS09] Jonathan Aldrich, Joshua Sunshine, Darpan Saini, and Zachary Sparks. Typestate-oriented programming. In *OOPSLA '09: Proceeding of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, pages 1015–1022, New York, NY, USA, 2009. ACM.

[BG93] Gilad Bracha and David Griswold. Strongtalk: Typechecking Smalltalk in a production environment. In *OOPSLA '93: Proceedings of the 8th International Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 215–230, 1993.

[BSS+10] Eric Bodden, Andreas Sewe, Jan Sinschek, Mira Mezini, and Hela Oueslati. Taming reflection (extended version): Static analysis in the presence of reflection and custom class loaders. Technical report, TU Darmstadt, 2010.

[BSS+11] Eric Bodden, Andreas Sewe, Jan Sinschek, Mira Mezini, and Hela Oueslati. Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders. In *ICSE '11: Proceedings of the 33rd ACM/IEEE International Conference on Software Engineering*, pages 241–250. ACM Press, 2011.

[ES98] Katalin Erdös and Harry M. Sneed. Partial comprehension of complex programs (enough to perform maintenance). In *IWPC '98: Proceedings of the 6th International Workshop on Program Comprehension*, page 98, Washington, DC, USA, 1998. IEEE Computer Society.

[GMD+10] Mark Grechanik, Collin McMillan, Luca DeFerrari, Marco Comi, Stefano Crespi, Denys Poshyvanyk, Chen Fu, Qing Xie, and Carlo Ghezzi. An empirical investigation into a large-scale java open source code repository. In *ESEM '10: Proceedings of the 4th International Symposium on Empirical Software Engineering and Measurement*, pages 11:1–11:10, 2010.

[GR83] Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, 1983.

[HH09] Alex Holkner and James Harland. Evaluating the dynamic behaviour of python applications. In *ACSC '09: Proceedings of the 32nd Australasian Computer Science Conference*, pages 17–25, 2009.

[Knu71] Donald E. Knuth. An empirical study of fortran programs. *Software: Practice and Experience*, 1(2):105–133, 1971.

[LRL10] Mircea Lungu, Romain Robbes, and Michele Lanza. Recovering inter-project dependencies in software ecosystems. In *ASE'10: Proceedings of the 25th IEEE/ACM international conference on Automated Software Engineering*, ASE '10, pages 309–312, 2010.

[MA09] Donna Malayeri and Jonathan Aldrich. Is structural subtyping useful? an empirical study. In *ESOP '09: Proceedings of the 18th European Symposium on Programming Languages and Systems*, pages 95–111, 2009.

[Mez11] Mira Mezini, editor. *Proceedings of the 25th European Conference on Object-Oriented Programming (ECOOP 2011)*, volume 6813 of *Lecture Notes in Computer Science*, Lancaster, UK, July 2011. Springer-Verlag.

[MPTN08] Radu Muschevici, Alex Potanin, Ewan D. Tempero, and James Noble. Multiple dispatch in practice. In *OOPSLA '08: Proceedings of the 23rd ACM International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 563–582, 2008.

[MT07] Hayden Melton and Ewan D. Tempero. An empirical study of cycles among classes in java. *Empirical Software Engineering*, 12(4):389–415, 2007.

[PBMH11] Chris Parnin, Christian Bird, and Emerson R. Murphy-Hill. Java generics adoption: how new features are introduced, championed, or ignored. In *MSR 2011: Proceedings of the 8th International Working Conference on Mining Software Repositories*, pages 3–12, 2011.

[RD07] Filip Van Rysselberghe and Serge Demeyer. Studying versioning information to understand inheritance hierarchy changes. In *MSR '07: Proceedings of the 4th International Workshop on Mining Software Repositories*, page 16, 2007.

[RLBV10] Gregor Richards, Sylvain Lebresne, Brian Burg, and Jan Vitek. An analysis of the dynamic behavior of javascript programs. In *PLDI '10: Proceedings of the 31st ACM conference on Programming Language Design and Implementation*, pages 1–12, 2010.

[RLBV11] Gregor Richards, Sylvain Lebresne, Brian Burg, and Jan Vitek. The eval that men do: A large-scale study of the use of eval in javascript applications. In Mezini [Mez11], pages 52–78.

[SBD04] Nathanael Schärli, Andrew Black, and Stéphane Ducasse. Object-oriented encapsulation for dynamically-typed languages. In *Proceedings of the 19th ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 2004)*, pages 130–149, Vancouver, British Columbia, Canada, October 2004. ACM Press.

[Tem09] Ewan D. Tempero. How fields are used in java: An empirical study. In *ASWEC '09: Proceedings of the 20th Australian Software Engineering Conference*, pages 91–100, 2009.

[TNM08] Ewan D. Tempero, James Noble, and Hayden Melton. How do java programs use inheritance? an empirical study of inheritance in java software. In *ECOOP '08: Proceedings of the 22nd European Conference on Object-Oriented Programming*, pages 667–691, 2008.

[Tri06] Mario Triola. *Elementary Statistics*. Addison-Wesley, 10th edition, 2006.

[WGTA11] Roger Wolff, Ronald Garcia, Éric Tanter, and Jonathan Aldrich. Gradual typestate. In Mezini [Mez11].