How Developers Use the Dynamic Features of Programming Languages: The Case of Smalltalk

Oscar Callaú Romain Robbes
Éric Tanter*
PLEIAD Laboratory
DCC, University of Chile
{oalvarez,rrobbes,etanter}@dcc.uchile.cl

David Röthlisberger
Software Composition Group
University of Bern
roethlis@iam.unibe.ch

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. These results are useful to make informed decisions about which features to consider when designing language extensions or tool support.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features

General Terms

Software Repositories

Keywords

Dynamic languages, static analysis, Smalltalk

1. INTRODUCTION

Dynamic object-oriented languages such as Smalltalk [4] 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 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 [2]. 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 [8]. We statically analyze these

*Partially funded by FONDECYT Project 1110051, Chile.

MSR'11, May 21–22, 2011, Waikiki, Honolulu, HI, USA

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.

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 validate, or not, 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. Dynamic features are mostly used in very specific kinds of projects. We conjecture that they are used essentially in core system libraries, development tools, and tests, rather than in regular applications.
3. Some usages of dynamic features are statically tractable, unproblematic for static analyses and other tools.
4. The specific features that have been integrated in more static languages over time (eg. Java) are indeed the most used.

As a consequence of this study, we also expect to gain insight into how language designers and tool providers have to deal with these dynamic features.

Structure of the Paper. We start with a review of 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). We close the paper by first discussing the potential threats to the validity of this study (Section 7), before concluding (Section 8).

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" [7]. 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) [10].

Tempero *et al.* characterized the usage of inheritance in 90 Java programs, and found a higher usage of inheritance than they expected [17]. 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 [14]. Later, Tempero analyzed a corpus of 100 Java programs in order to characterize how fields were used [16]: 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 [11].

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 [9].

Finally, 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 [5], related to the size of the applications, the number of arguments in methods, whether methods are overriden or not, etc.

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 [1]. 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 [6]. They found that the Python program in their corpus made a heavier usage of dynamic features during their startup 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.* [12]. They analyzed a large amount of Javascript code found on a number of 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 [13]. 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.

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

3. SMALLTALK BASICS

Smalltalk [4] 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 modify the value of a dynamically-determined field in an object. In addition, Smalltalk also supports swapping all pointers to two given 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.

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.

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.

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* [8], a software ecosystem model to trace dependencies between software projects. Our analyzer follows three principal steps: *Trace*, *Collect* and *Classify*.

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.

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

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 three groups of dynamic features of Smalltalk in this study: first-class classes, behavioral reflection, and structural reflection. 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 [3].

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

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.
```

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 Update. 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 such as instVarAt:put: and variants, which allows assigning 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.
```

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

In our analysis of the 1,000 projects, we found 14,184 dynamic feature occurrences. Only 8,349 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 8,349 methods using dynamic features represent 1.28% 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.

Of the 8,349 methods using dynamic features, 4,869 were in projects classified as "Applications" (0.75% of all analyzed methods) and 2,946 of these use dynamic features that we consider as unsafe (0.45% of methods); these results confirm the previous point.

[4]In Smalltalk the root superclass is nil.

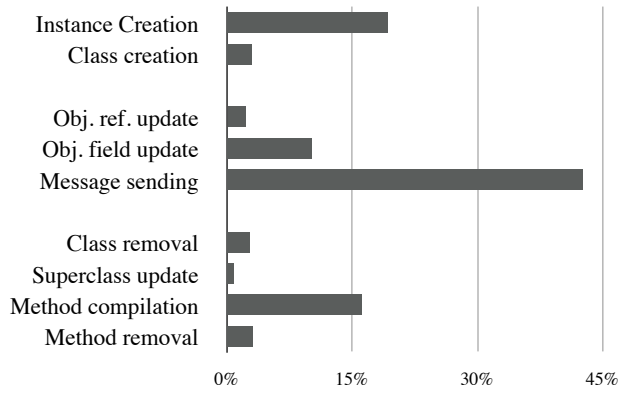

Figure 1: Distribution of dynamic feature usages.

Projects classified as applications represent 83% of the projects, yet contain barely 57% of the methods using dynamic features, confirming the fact that other project categories use these features more extensively. Of all the dynamic feature usages, 8,642 were classified as unsafe (60.92%); 4,047 of those were in applications (46.82%). This confirms the previous fact, but is not enough to validate our hypothesis that dynamic features are mostly used in specific kinds of projects. Their usage is clearly more widespread.

Figure 1 shows the distribution of the usage of dynamic features, with a maximum of 6,048 occurrences of message sending (42.64%) and a minimum of 114 occurrences for superclass updates (0.8%). Categories are distributed as follows: first-class classes with 22.22%, behavioral reflection with 55.99% and structural reflection with 22.79%. Three dynamic features—Message sending, Instance creation, and Method recompilation—, account for more than 75% of the usages. Of these, Java provides two in its reflection API (message sending and instance creation), catering to 60% of the usages in the analyzed Smalltalk projects.

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 and message sending) follow a common pattern:

- 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 53 and 52 projects respectively. Applications with 85% 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 20 projects.

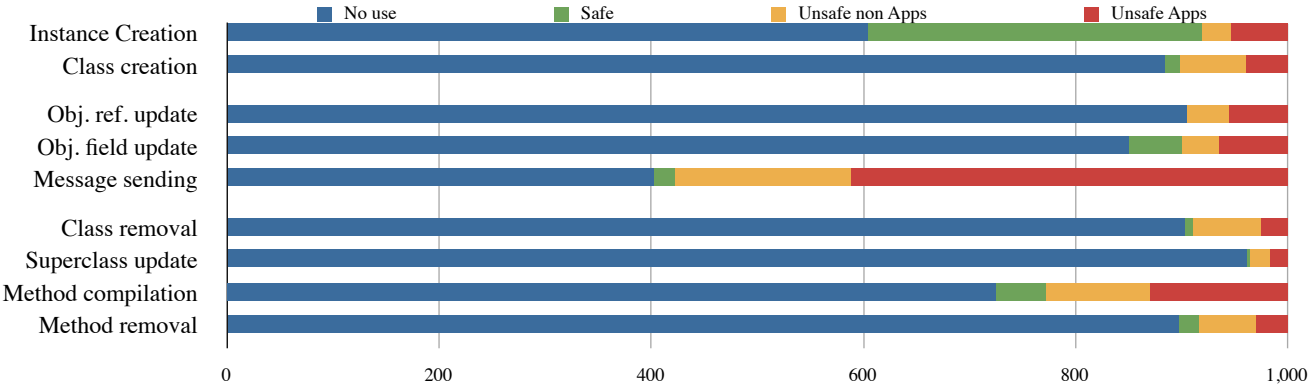

Figure 2: Per-feature distribution of all projects arranged by category of use.

The cases of instance creation and message sending 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. These two features are used pervasively by all kinds of projects.

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 the assumption that dynamic features are not used often, but shows that they cannot be safely ignored. An analysis of each feature is needed.
- Application projects use less dynamic features than other types of projects. The assumption that conventional applications use few dynamic features is invalid: applications constitute nearly half of the unsafe uses.
- The two most pervasive features, Instance creation and Message sending, are the ones that static languages such as Java implement, confirming assumption 4.

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 italic categories that are particularly over-represented with respect to the Applications category: Since applications constitute more than 83% of the projects, we consider that any other category that has at least one-third of the number of unsafe calls that applications have is over-represented.

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

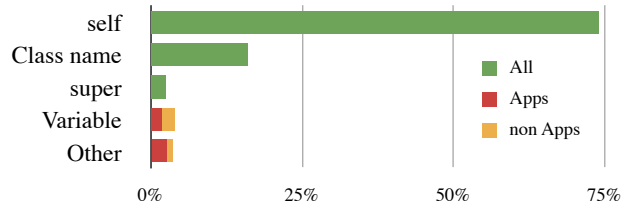

Figure 3: Safe/unsafe usages of instance creation.

as 83% of the projects are applications. On the contrary, System projects are the most over-represented (1.76%).

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).

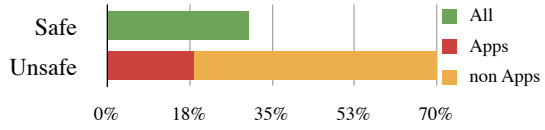

Figure 4: Safe/unsafe usages of class creation.

Figure 4 and Table 3 show that a strong minority of cases are safe uses (30%); 18% of unsafe usages are in application, while more than 50% are in other project categories. Tests are especially over-represented, with nearly a third of unsafe usages. Indeed, tests often create temporary classes for testing purposes. Likewise, System and Tools are both heavily over-represented, each having close to 10% of uses of the features; both project categories 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%.

Dynamic Feature	Nb. of Calls	% Safe Calls	% Apps	% Tools	% Lang-Ext.	% System	% Tests
Instance Creation	2,732	**92.53**	4.32	0.59	0.70	*1.76*	0.11
Class Creation	420	30	18.33	*9.76*	1.90	*8.57*	**31.43**
Object ref. update	311	0	**45.66**	12.22	6.75	*24.44*	10.93
Object field update	1,441	**61.14**	18.67	5.20	1.80	*12.91*	0.28
Message sending	6,048	7.03	**47.52**	*26.57*	3.17	9.79	5.92
Class removal	385	6.24	10.91	*8.05*	1.56	*10.91*	**62.34**
Superclass update	114	7.89	**42.11**	*18.42*	7.02	7.02	*17.54*
Method compilation	2,296	**60.02**	18.25	7.10	3.22	6.32	5.10
Method removal	311	**39.13**	13.27	*15.10*	3.43	*18.76*	*10.30*

Table 3: Per-feature distribution of safe and unsafe calls, where unsafe calls are sorted by project category. In bold: largest category; In italics: category that is considerably over-represented (< 1/3 of Apps usages)

Interpretation.

- Instance creation is the second-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

Object Reference Update (311 calls, 311 unsafe, 142 in Apps).

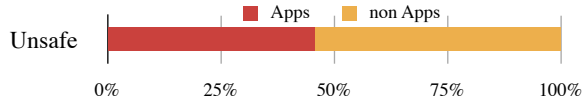

Figure 5: Unsafe uses of object references updates.

According to Table 3, System projects are over-represented (2.5% of projects account for 25% of calls). 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.

Object Field Update (1,441 calls, 560 unsafe, 269 in Apps).

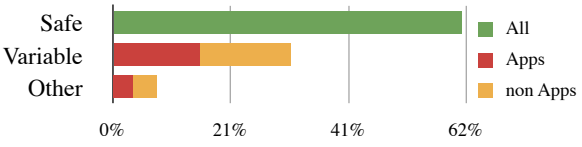

Figure 6: Safe/unsafe usages of object field updates.

Accessing and changing the values of object fields is the fourth-most used feature. Figure 6 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, for reasons similar to the uses of object reference updates. 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.

Message Sending (6,048 calls, 5,623 unsafe, 2,874 in Apps).

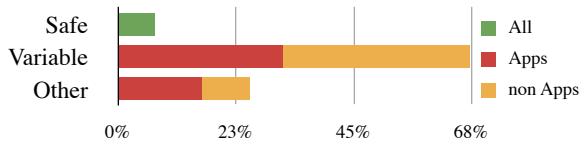

Figure 7: Safe/unsafe usages of message sending.

The most used dynamic feature accounts for 42,63% of all occurrences. Unfortunately, most of these usages (92,97%) are unsafe (Figure 7). 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. Tool projects follow with a quarter of all occurrences (26.57%); 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. The rest

is split between the other project categories in the following, descending order: System (nearly 9.79%, also over-represented), Tests (5.92%) and Language-extensions (3.17%).

Interpretation.

- Supporting message sending is a priority: it constitutes more than 40% 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 [2].
- The other two behavioral features—Object reference and field updates—are used infrequently. System projects on the other hand do use them pervasively.
- Object field updates are 60% safe, due to their usage as a serialization mechanism. In addition, unsafe calls could be tractable by a gradual type system [15]. Reference updates are much more challenging to support.

5.4 Structural Reflection

Class Removal (385 calls, 361 unsafe, 42 in Apps).

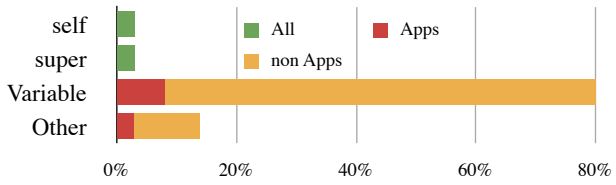

Figure 8: Safe/unsafe usages of class deletion.

Class removal is one of the lesser-used features. According to Figure 8, 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), whereas System projects have the same number of usages. Tests provide the overwhelming majority of unsafe usages with 62.34%. This behavior ties up with the heavy usage by tests of dynamic class creation. A common pattern in tests (208 instances, more than 80%), is to create a new class, run tests on it, and then delete it.

Superclass Update (114 calls, 105 unsafe, 48 in Apps).

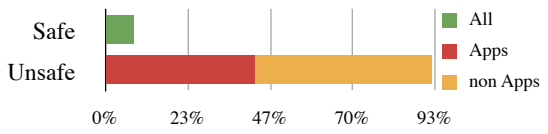

Figure 9: Safe/unsafe usages of superclass updates.

This feature is the least used with just 114 occurrences, 0.80% of all dynamic feature occurrences. As shown in Figure 9, safe calls account for 7.89% while 42.11% are unsafe calls inside App projects; Tools and Tests are also heavy users, with 18.22 and 17.54% respectively. Since tests often build classes to run test code on, it stands to reason that they would also need to specify their superclasses.

Method Compilation (2,296 calls, 918 unsafe, 419 in Apps).

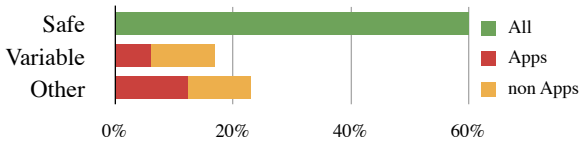

Figure 10: Safe/unsafe uses of method compilation.

Method compilation is the third-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 10). 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.

There is no clear distribution of unsafe method compilation among categories. Applications feature a bit less than half of the usages (18.25%), and are hence under-represented; other categories fluctuate between 3.22 and 7.10% (Table 3).

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.

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 11. 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: both Tools and System projects have more usages than Apps, while Test projects have nearly as much as Apps (10.3 vs. 13.27%).

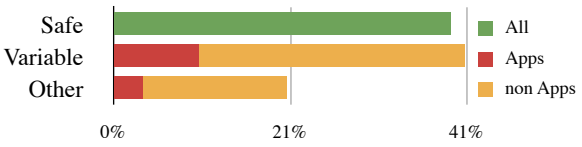

Figure 11: Safe/unsafe uses of method removal.

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).
- 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.

6. DISCUSSION

We discuss whether each of the assumptions 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—1.28%. Assumption 1 is validated.
2. **Dynamic features are used in specific kinds of projects.** We conjectured that core system librairies, 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 53.17% of unsafe usages), the latter still makes up nearly half of all the unsafe usages. Assumption 2 is invalidated.
3. **Some usages of dynamic features are statically tractable.** We found that 3 features (instance creation, object field updates, and method compilation) have a majority of safe uses. Two others (class creation and method removal) have a strong minority (more than 30%) of safe uses. Assumption 3 is validated.
4. **The most used dynamic features are supported by more static languages.** The two most used features, reflective message sending and instance creation, are supported by the Java reflection API, validating assumption 4.

Even if dynamic features are used in a minority of methods (1.28%, 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.

- **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.
- **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 [3].
- **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 updates** is the last of the feature that has a somewhat widespread usage. A majority of usages are safe however.
- **Object reference updates** are somewhat problematic, as nearly 45% of the usages are in applications. Supporting such a dynamic feature is also a challenge.
- **Class creation and removal** are heavily used in tests, but class creation is used in applications as well.
- **Method removal** has a large number of safe uses, and is primarily used outside applications.
- **Superclass updates** is a somewhat exotic features whose usages are few and far between.

As a rule of thumb, we conclude that message sending, method compilation, instance creation, object reference and field updates, and to a lesser degree also class creation, 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 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. THREATS TO VALIDITY

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).

We only use static analysis as it would be impractical to perform dynamic analysis on 1,000 projects. However, we cannot be sure whether the code triggering dynamic features is actually executed; some dynamic feature usage could on the other hand be and 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.

External Validity. Our study features 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.

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 cross-cutting 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. 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, invalidating one. 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, instance creation, method compilation); others less so.

To increase confidence in our results, we plan to (1) perform dynamic analysis on a subset of the projects, in order to study whether and how often the dynamic features are used in running programs, and (2) to replicate the study in other programming languages such as Java.

9. REFERENCES

[1] Bodden, E., Sewe, A., Sinschek, J., Mezini, M., Oueslati, H.: Taming reflection (extended version): Static analysis in the presence of reflection and custom class loaders. Tech. rep., TU Darmstadt (2010)

[2] Bodden, E., Sewe, A., Sinschek, J., Mezini, M., Oueslati, H.: 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 (2011), to appear.

[3] Bracha, G., Griswold, D.: Strongtalk: Typechecking Smalltalk in a production environment. In: OOPSLA '93: Proceedings of the 8th International Conference on Object-Oriented Programming Systems, Languages and Applications. pp. 215–230 (1993)

[4] Goldberg, A., Robson, D.: Smalltalk-80: The Language and its Implementation. Addison-Wesley (1983)

[5] Grechanik, M., McMillan, C., DeFerrari, L., Comi, M., Crespi, S., Poshyvanyk, D., Fu, C., Xie, Q., Ghezzi, C.: 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. pp. 11:1–11:10 (2010)

[6] Holkner, A., Harland, J.: Evaluating the dynamic behaviour of python applications. In: ACSC '09: Proceedings of the 32nd Australasian Computer Science Conference. pp. 17–25 (2009)

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

[8] Lungu, M., Robbes, R., Lanza, M.: Recovering inter-project dependencies in software ecosystems. In: ASE'10: Proceedings of the 25th IEEE/ACM international conference on Automated Software Engineering. pp. 309–312. ASE '10 (2010)

[9] Malayeri, D., Aldrich, J.: Is structural subtyping useful? an empirical study. In: ESOP '09: Proceedings of the 18th European Symposium on Programming Languages and Systems. pp. 95–111 (2009)

[10] Melton, H., Tempero, E.D.: An empirical study of cycles among classes in java. Empirical Software Engineering 12(4), 389–415 (2007)

[11] Muschevici, R., Potanin, A., Tempero, E.D., Noble, J.: Multiple dispatch in practice. In: OOPSLA '08: Proceedings of the 23rd ACM International Conference on Object-Oriented Programming, Systems, Languages, and Applications. pp. 563–582 (2008)

[12] Richards, G., Lebresne, S., Burg, B., Vitek, J.: An analysis of the dynamic behavior of javascript programs. In: PLDI '10: Proceedings of the 31st ACM conference on Programming Language Design and Implementation. pp. 1–12 (2010)

[13] Richards, G., Lebresne, S., Burg, B., Vitek, J.: The eval that men do: A large-scale study of the use of eval in javascript applications. Tech. rep., Purdue University (2010)

[14] Rysselberghe, F.V., Demeyer, S.: Studying versioning information to understand inheritance hierarchy changes. In: MSR '07: Proceedings of the 4th International Workshop on Mining Software Repositories. p. 16 (2007)

[15] Siek, J., Taha, W.: Gradual typing for objects. In: ECOOP '07: Proceedings of the 21st European Conference on Object Oriented Programming. pp. 2–27 (2007)

[16] Tempero, E.D.: How fields are used in java: An empirical study. In: ASWEC '09: Proceedings of the 20th Australian Software Engineering Conference. pp. 91–100 (2009)

[17] Tempero, E.D., Noble, J., Melton, H.: 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. pp. 667–691 (2008)