How Do Developers Use Dynamic Features? The Case of Ruby

Elder Rodrigues Jr and Ricardo Terra
Department of Computer Science, UFLA, Brazil
elderjr@computacao.ufla.br,terra@dcc.ufla.br

Abstract

Programming languages provide features that allow developers—at run time—to evaluate strings as expressions, to define and invoke methods, etc. Nevertheless, the overuse of dynamic features may negatively impact performance, decrease the accuracy of static code analysis techniques, and hinder compiler optimizations. This paper, therefore, investigates how developers use dynamic features based on 28 open-source Ruby projects. The main findings are fourfold: (i) dynamic features correspond on average to 2.58% of the language features in a Ruby project, and almost half of them are `send` when we disregard features that are ascribed to common programming practices; (ii) we identified that 1,417 out of 2,849 dynamic statements (49.7%) could be easily converted to static code; (iii) we identified, classified, and also illustrated the ten most common scenarios where developers opt for dynamic statements instead of static ones, e.g., we found that developers rely mostly on dynamic features to access private members (20.35%), which reveals flaws in the architectural design; and (iv) we identified five motivations why developers use dynamic features: unusual coding style, overpassing privacy restrictions, defining new structures, contextualizing block executions, and generalizing code tasks.

Keywords: programming languages; dynamic features; empirical study.

1 Introduction

Programming languages provide features that allow developers—at run time—to evaluate strings as expressions, to define and invoke methods, etc. In a nutshell, these dynamic features promote flexibility at many levels [7]. As an example, developers can write a few dynamic statements that are responsible for adapting the code in several scenarios. This results in fewer lines of code and higher changeability.

Nevertheless, the overuse of dynamic features might be undesired. Essentially, their usage may jeopardize the accuracy of static analysis techniques [3, 2, 8, 9, 10] and may cause the loss of legibility, maintainability, etc. [12]. For instance, (i) it may jeopardize early error detection and optimizations by compilers [5]; (ii) it may reduce the accuracy of type inference algorithms (especially in dynamically typed languages), which can hide type errors [13] or architectural violations [11]; (iii) it may

inhibit IDE features, such as auto-complete and refactoring tasks [12]; and (iv) it may make the software more complex to understand, which impacts negatively on program comprehension.

In view of such circumstances, this paper investigates how developers use dynamic features based on 28 open-source Ruby projects (which include more than a half million lines of code). From the results from our empirical study, we can point out four main findings:

1. On average, dynamic features correspond to 2.58% of the language features in a Ruby project. Almost 50% of them are `send` when we disregard features that are ascribed to common programming practices. Stated differently, lines of code that contain dynamic statements correspond to 4.68% and methods that contain dynamic statements correspond to 7.46%.

2. We identified that 1,417 out of 2,849 dynamic statements (49.7%) could be easily converted to static code. This percentage increases to 60.75% and 60.55% for the domains *Applications* and *Frameworks*, respectively.

3. We identified, classified, and also illustrated the ten most common scenarios where developers opt for dynamic statements instead of static ones, e.g., we found that developers rely mostly on dynamic features to access private members (20.36%), which reveals flaws in the architectural design.

4. We identified five motivations why developers use dynamic features: unusual coding style, overpassing privacy restrictions, defining new structures, contextualizing block executions, and generalizing code tasks.

The remainder of this paper is organized as follows. Section 2 presents our study setup. Sections 3, 4, 5, and 6 present and discuss the results of each research question. Section 7 enumerates threats to validity. Finally, Section 8 presents related work and Section 9 concludes the paper.

2 Study Setup

2.1 Research Questions

We designed a study to address the following overarching research questions:

RQ #1 - *How often do developers use dynamic features?*

RQ #2 - *How complex is removal of dynamic features?*

RQ #3 - *When do developers use dynamic features?*

RQ #4 - *Why do developers use dynamic features?*

2.2 Dynamic Features in Ruby

Ruby is a dynamically and strongly typed language. Regarding dynamic features, Ruby provides functions to dynamically invoke methods, define blocks, and execute strings as code. Each of the dynamic features handled in this paper is described in Table 1.

Table 1: Description of dynamic features

#	Name	Description
*1	`require`	Loads a Ruby file named by the given string.
*2	`include`	Includes a module's methods in the class/module body.
*3	`respond_to`	Answers if an object responds to the given method name.
*4	`method_missing`	It is the invoked procedure when a missing method is called.
5	`class_eval`	Evaluates the given string or block in the context of the class/module.
6	`module_eval`	It is an alias to `class_eval`.
7	`instance_variable_get`	Gets the value of an instance variable named by the given string.
8	`instance_variable_set`	Sets the value of an instance variable named by the given string.
9	`class_variable_get`	Gets the value of a class variable named by the given string.
10	`class_variable_set`	Sets the value of a class variable named by the given string.
*11	`attr_reader`	Creates an instance variable and corresponding getter method.
*12	`attr_writer`	Creates an instance variable and corresponding setter method.
*13	`attr_accessor`	Creates an instance variable and corresponding getter and setter methods.
14	`const_get`	Gets the value of a constant named by the given string.
15	`const_set`	Sets or creates the value of a constant named by the given string.
16	`define_method`	Defines methods dynamically named by the given string and a given block is the body.
17	`eval`	Evaluates the Ruby expressions inside a string.
18	`instance_eval`	Executes blocks in the context of an object.
19	`instance_exec`	Similar to `instance_eval` where blocks can receive parameters.
20	`send`	Invokes a method named by the given string.

* *These dynamic features are only considered in RQ #1, as further explained.*

2.3 Subject Systems

Table 2 presents the 28 systems considered in the study, including their names, versions, lines of code (LOC), number of rb files, and number of libraries (Gems). Our selection criterion was the 30 most starred Ruby projects on GitHub.[1] We discarded only two projects: Bootstrap for Sass, a tiny project that solely provides support to the Sass-based bootstrap CSS framework; and Software Engineering Blogs, which is not an application but a plain Ruby script that generates an OPML[2] file with a list of technology web sites. In total, we analyzed 524,730 LOC inside 9,825 rb files.

[1] https://github.com/search?l=&o=desc&q=language:Ruby+stars:¿1&ref= advsearch&s=stars&type=Repositories, as retrieved in August 2015.

[2] Outline Processor Markup Language (OPLM) is an XML format for outlines, which is easily imported by RSS readers.

Table 2: Subject Systems

#	Project and version	LOC	# of rb files	# of Gems
1	Active Admin (v1.0.0.pre1)	5,473	154	42
2	CanCan (v1.6.10)	753	16	13
3	Capistrano(v.3.4.0)	1,908	44	7
4	Capybara (v2.5.0)	6,984	107	20
5	Carrierwave (v0.10.0)	1,856	37	15
6	CocoaPods (v0.39.0.beta.4)	6,415	94	41
7	DevDocs (66cefbd)	10,090	293	27
8	devise (v3.4.1)	3,678	93	19
9	diaspora* (v0.5.2.0)	13,818	376	128
10	Discourse (v1.4.0.beta8)	35,039	655	101
11	FPM (v1.4.0)	3,382	25	11
12	GitLab (v7.14.1)	26,658	572	137
13	Grape (v0.13.0)	3,022	88	24
14	Homebrew (8278b89)	123,557	3429	4
15	Homebrew-Cask (v0.56.0)	5,364	136	8
16	Huginn (f4b8e73)	8,016	124	90
17	Jekyll (v3.0.0.pre.beta8)	3,288	61	39
18	Octopress (v2.0)	1,147	23	13
19	Paperclip (v4.3.0)	2,663	59	34
20	Rails (v4.2.4)	48,081	849	82
21	Rails Admin (v0.7.0)	3,966	111	48
22	Resque (v1.25.0.pre)	1,630	25	12
23	Ruby (v2_2_3)	140,797	1,073	0
24	Sass (v3.4.18)	12,198	130	8
25	Simple Form (v3.1.0.rc2)	1,898	55	9
26	Spree (v3.0.4)	45,256	1,057	6
27	Vagrant (v1.7.4)	7,195	126	21
28	Whenever (v0.9.4)	598	13	3
	Total	**524,730**	**9,825**	**962**

2.4 Tool Support

Although most analyses in this study are manual, we developed a tool, named `rdf` (Ruby Dynamic Features Analysis Tool), which aids us in conducting our investigation process in a more organized and effective way.[3] `Rdf` has the following features:

- ./`rdf setup`: Inserts a default mark after every dynamic instruction (refer to Section 2.2, e.g., `const_get`, `eval`, `send`, etc.) in the project to indicate that those have still not been classified. The default marking is:
  #`rdf` ⟨`ID`−⟨`instruction − type`⟩⟩ ⟨`not−yet−classified`⟩
- ./`rdf show_locations` ⟨`instruction−type`⟩: Lists the files where that instruction type has already been marked.
- ./`rdf show_locations_without_classification` ⟨`instruction−type`⟩: Lists the files where that instruction type has already been marked, but has still not been classified.
- ./`rdf show_classifications`: Summarizes the number of statements by each classification.

[3]The tool, its source code, and our dataset are publicly available at: `https://github.com/rterrabh/rdf`

3 How often do developers use dynamic features? (RQ #1)

In this research question, our goal is to measure the frequency of use of dynamic features in Ruby.

3.1 Methodology

We estimate the use of dynamic statements in three ways. First, we contrast the number of dynamic features to the total number of language features in the subject systems. In this context, language features are expressions, statements, and declarations, such as assigns, method calls, branches, loops, etc. Dynamic features, on the other hand, are dynamic invocations or builds (dynamic definitions of classes, methods, and variables). In this study, we considered the 19 Ruby features (plus one alias) in Table 2 to be dynamic. Second, we compare the total of lines of code (LOC) to those that contain dynamic statements. Third, we compare the total number of methods to those that contain dynamic statements. An example is illustrated in Listing 1.

```
1 class Test
2   def bar
3     x = Z.new
4     if x.send(:foo)
5       y = 3
6     end
7   end
8 end
```

Listing 1: Number of dynamic statements

In this example, there are seven language features: (i) class definition at line 1, (ii) method definition at line 2, (iii) instantiation of type `Z` at line 3, (iv) assign to variable `x` also at line 3, (v) conditional branch at line 4, (vi) dynamic invocation of method `foo` also at line 4, and (vii) assign to variable `y` at line 5. First, the ratio between the number of dynamic features and the total number of language features in this example is $1/7 = 14.3\%$. Second, since the total of LOC is five[4] and a single line contains a dynamic statement (line 4), the dynamic LOC is $1/5 = 20\%$. Third, since there is only one method and it contains a dynamic statement, the rate of methods that use dynamic statements is $1/1 = 100\%$.

3.2 Result

Table 3 reports the total of dynamic statements in the subject systems according to our metrics defined in Section 3.1. Column "Dynamic statements" reports the total of statements that are dynamic. For instance, Active Admin has 371 out of 8,978 statements (4.13%) that are dynamic. Column "Dynamic LOC" reports the total of LOC that contain dynamic statements. For instance, Active Admin has 340 from 5,473 LOC (6.21%) that contain dynamic statements. Column "Dynamic methods" reports the total of methods that contain dynamic statements. For instance, Active Admin has 84 out of 786 methods (10.69%) that contain dynamic statements.

[4]Our script disregards the `end` keyword as a line of code.

Table 3: Total of dynamic statements in the subject systems

Project	Dynamic statements		Dynamic LOC		Dynamic methods	
Active Admin	4.13%	(371/8,978)	6.21%	(340/5,473)	10.69%	(84/786)
CanCan	4.00%	(61/1,526)	7.84%	(59/753)	15.83%	(22/139)
Capistrano	1.61%	(78/4,834)	4.04%	(77/1,908)	4.97%	(9/181)
Capybara	1.31%	(218/16,658)	3.12%	(218/6,984)	3.01%	(15/499)
Carrierwave	4.86%	(160/3,292)	8.46%	(157/1,856)	14.29%	(37/259)
CocoaPods	2.62%	(330/12,631)	5.10%	(327/6,415)	4.15%	(34/820)
DevDocs	0.75%	(142/18,832)	1.41%	(142/10,090)	2.0%	(13/651)
devise	3.77%	(227/6,018)	6.01%	(221/3,678)	12.13%	(57/470)
diaspora*	1.47%	(513/34,892)	3.65%	(504/13,818)	4.65%	(68/1462)
Discourse	1.09%	(707/64,567)	1.96%	(688/35,039)	5.42%	(226/4171)
FPM	2.79%	(182/6,516)	5.20%	(176/3,382)	3.80%	(6/158)
GitLab	1.21%	(627/51,779)	2.35%	(626/26,658)	3.38%	(101/2,985)
Grape	3.36%	(178/5,301)	5.76%	(174/3,022)	10.29%	(39/379)
Homebrew	0.71%	(1,196/169,204)	0.96%	(1,182/123,557)	3.39%	(217/6401)
Homebrew-Cask	3.41%	(314/9,197)	5.72%	(313/5,364)	7.08%	(48/678)
Huginn	1.19%	(185/15,533)	2.30%	(184/8,016)	2.94%	(25/850)
Jekyll	1.94%	(114/5,886)	3.47%	(114/3,288)	4.62%	(23/498)
Octopress	2.05%	(44/2,151)	3.84%	(44/1,147)	3.23%	(4/124)
Paperclip	5.09%	(240/4,714)	8.60%	(229/2,663)	18.69%	(83/444)
Rails	3.65%	(3,076/84,381)	6.22%	(2,990/48,081)	8.45%	(611/7,228)
Rails Admin	4.84%	(387/7,994)	9.53%	(378/3,966)	14.96%	(57/381)
Resque	3.48%	(100/2,872)	5.52%	(90/1,630)	8.78%	(23/262)
Ruby	2.16%	(5,181/239,962)	3.62%	(5101/140,797)	5.03%	(771/15,341)
Sass	1.97%	(489/24,845)	3.99%	(487/12,198)	4.87%	(71/1,459)
Simple Form	2.61%	(84/3,216)	4.32%	(82/1,898)	12.17%	(28/230)
Spree	1.2%	(1,244/103,582)	2.68%	(1,213/45,256)	6.63%	(161/2,428)
Vagrant	2.78%	(342/12,298)	4.70%	(338/7,195)	4.71%	(28/595)
Whenever	2.19%	(26/1,186)	4.35%	(26/598)	8.82%	(6/68)
Average	**2.58%**		**4.68%**		**7.46%**	
Total Dynamic St. (x)		16,816		16,480		2,867
Total Statements (y)		922,845		524,730		49,947
Overall average (x/y)		1.82%		3.14%		5.74%

The use of dynamic feature is quite low. Using the Shapiro-Wilk test, the data from dynamic statements and LOC come from a population that has a normal distribution (p-value = 0.1529 and 0.5444, respectively), whereas the data from dynamic methods deviate from normality (p-value = 0.0056). On average at the 95% confidence interval, [2.08,3.08] of statements are dynamic (t-test), [3.83,5.52] of lines of code has dynamic statements (t-test), and [5.00,9.37] of the methods contain dynamic statements (wilcoxon test).

We argue that the low usage of dynamic statements in comparison to static ones is due to the fact that a few dynamic features are sufficient for providing flexibility to a project. For example, Paperclip has to be very generic because it provides easy file attachment management that works on any database schema, even though only 5.09% of its statements are dynamic. Moreover, the most popular Ruby framework Rails—whose purpose is to provide generic functionality to other applications—has only 6.22% of its lines of code with dynamic statements or, stated differently, only 8.45% of its methods contain dynamic statements.

Table 4 reports the occurrences of each dynamic feature per project. Specifically in the `method_missing` column, the total of classes that implement such method are compared to the total of classes of the system, e.g., Active Admin has only two classes out of a total of 453

that implement `method_missing`. It is worth noting that the large number of `require` (7,171), `attr` variants (2,827), `include` (2,651), and `respond_to` (1,219) is due to common programming practices, such as changing visibility of variables and including methods in classes. Furthermore, the use of the `send` feature is recurrent in Ruby. Developers relied on this feature 1,271 times to dynamically invoke a method named by the given string. For example, you can write `s = o.send(x)` instead of `s = o.name()`, assuming `x = "name"`. We argue that it is easier for developers to use `send` when the name of the called method is determined at run time.

Table 4: Occurrences of dynamic features per project (zeros are omitted)

Project	attr_accessor*	attr_reader*	attr_writer*	class/module_eval	class_variable_get	class_variable_set	const_get	const_set	define_method	eval	include*	instance_eval	instance_exec	instance_variable_get	instance_variable_set	method_missing*	require*	respond_to*	send
Active Admin	14	27	6				2	1	11	2	81	1	33	2	1	2/453	91	35	62
Cancan		2	2	2			1				4			4	3	0/40	12	9	22
Capistrano		3	1								10					1/67	53	7	3
Capybara	9	19	1	2					3	1	87	3				0/105	69	14	10
Carrierwave	6	8	1	9				2		2	34				1	1/130	57	25	14
CocoaPods	84	97	6				1		3	1	13				1	1/320	96	7	20
Devdocs	7	5		3			3			1	19				1	0/722	80	2	21
Devise	6	7		10			4				49	1		3	5	0/194	59	53	30
Diaspora	38	6	3	3			1		2	1	144	29	1	13	17	2/433	165	34	54
Discourse	70	41	10	3			2		38	25	89	2	2	1	4	2/842	231	39	148
FPM	20								1		5			1	1	0/34	136	7	11
GitLab	48	77		2		1	3	1	4	1	204			1		5/847	184	46	50
Grape	12	14	2	3			2	1	7		25	8	6			1/298	66	23	8
Homebrew	18	98	2	4			3	2	16		368	15			1	0/3,639	605	22	42
Homebrew-Cask	15	27					5		7	1	14	5			1	6/181	185	33	15
Huginn	7	7		3			5	5	2	4	54	1	1		1	1/226	66	7	21
Jekyll	15	26	1	1			1		2		15					0/158	39	8	6
Octopress	3	1									12					0/62	24	4	
Paperclip	5	4	3	3			5		5	1	2	2		1	1	0/158	91	63	54
Rails	117	203	20	94		1	24	11	71	8	415	16	22	19	20	33/2,704	1,364	359	279
Rails Admin	18	18					1		4		21	21		9	9	4/407	213	19	50
Resque	12	2	3				2				3			1		0/52	36	31	10
Ruby	464	651	51	139			39	63	51	94	681	166	29	33	84	27/3,647	2,224	266	119
Sass	111	81	8	6			4	1			7	1		2	16	3/324	199	9	41
Simple Form		5			1	1	1				21		2			0/149	15	25	13
Spree	49	35	2	27			1		17	1	255	6	8	6	8	5/1,468	609	60	155
Vagrant	23	66		2			1		7		19			6	3	5/606	189	10	11
Whenever	1	3		1					1			2			1	0/19	13	2	2
Total	1,172	1,533	122	317	1	3	111	87	252	143	2,651	279	104	102	179	99/18,285	7,171	1,219	1,271
Average	41.86	54.75	4.36	11.32	0.04	0.11	3.96	3.10	9	5.11	94.68	9.96	3.71	3.64	6.39	3.54/653.03	256.11	43.54	45.39

In order to statistically check whether any dynamic feature is more used than another, we ran a one-way ANOVA (ANalysis Of VAriance) with post-hoc Tukey HSD (Honestly Significant Difference). First, we obtained the averages per dynamic feature and ran ANOVA to assess whether there are differences in the number of statements among the features. A p-value of $1.57e^{-10}$ for `require` indicates that this feature has significant effect on the response. The other p-values ranged from 0.108 which indicate that other features have no significant effect on the response. Next, we ran the Tukey test to see where the differences lie. It indicates that the differences between `require` and any other feature is significant. It is worth noting that the higher p-value was $1.867e^{-4}$ with the `include` feature; all others were practically zero. Therefore, we can statistically assert that `require` is more used than any other dynamic feature.

3.3 Final remarks

The use of dynamic features is quite low (2.58% on average). When we disregard dynamic statements that refer to common programming practices (`require`, `attr` variants, `include`, and `respond_to`), the average decreases to 0.52%. Even with this lower percentage, we claim that their usages must be addressed because they may represent code that overpasses privacy restrictions or that hampers the comprehensibility of the software design. Therefore, based on the quantitative results, the effort has to be first addressed to `send`, which is the most used dynamic feature that is not ascribed to a common programming practice.

4 How complex is removal of dynamic features? (RQ #2)

The complexity to remove dynamic features is a crucial factor to our study since it is directly linked with the need for dynamic statements. We claim the more trivial it is to remove dynamic features, the more superfluous their usages are. Since there is no direct benefit in removing the `include`, `require`, `respond_to`, and `attr` variants features, we disregard those from this point on. For example, when developers remove an `include` or `require` statement, they would have to locally copy the used methods from another module. As another example, developers usually see `attr` variant feature as a common practice of syntactic sugar, otherwise they have to manually create accessors methods.

4.1 Methodology

Aided by our supporting tool, we manually assigned a complexity level to remove every single dynamic statement analyzed in this study. Basically, for each of the 2,849 dynamic statements, we replaced them (or tried to) with static statements that provide the same behavior. Thus, we established the following complexity levels to remove dynamic features:

- *trivial*: Only local changes; no new statements.
- *easy*: Local and target class changes; no new statements.
- *moderate*: Local and target class changes; requires new statements.
- *complex*: Complex changing procedures; requires modifying many classes or external resources.

4.2 Result

Table 5 reports our results (we omit lines only with zeroes). In a general context, the number of dynamic statements that are not complex to remove (*trivial*, *easy*, and *moderate*) is 49.7%. This means that almost half of the parameters of the analyzed dynamic statements in this research question could be easily tracked by static analysis. When we only regard the `send` feature, which is the most used in this RQ, this number increases to 53.5%.

Table 5: Complexity analysis per system (zeros are omitted)

		Active Admin	CanCan	Capistrano	Capybara	Carrierwave	CocoaPods	DevDocs	devise	diaspora*	Discourse	FPM	GitLab	Grape	Homebrew	Homebrew-Cask	Huginn	Jekyll	Octopress	Paperclip	Rails	Rails Admin	Resque	Ruby	Sass	Simple Form	Spree	Vagrant	Whenever	Total
class/module_eval	Trivial	2																			8			5						**15**
	Moderate				1	2		3	9	1	2		1	2	3		3	1		3	31			42	6		21	2	1	**134**
	Complex				1	7			1	2	1		1	1	1						55			92		1	6			**168**
class_var_get	Moderate																													**1**
class_variable_set	Moderate										1																			**2**
	Complex																				1						1			**1**
const_get	Trivial								1			2	1		3						2			3						**12**
	Easy	1																						1						**1**
	Moderate								1	1	1				1	1	1				2			7	2		1			**19**
	Complex	2							1	3	2		1	3		1	1			5	20	1	2	28	2	1		1		**79**
const_set	Trivial													1	1		1				3			44						**50**
	Moderate	1											1		1		4				2			8	1					**11**
	Complex					2							1		4						6			11						**26**
define_method	Moderate	4			3					2	33		3	1	14	7	2	2		1	38	3		39		6	7	1		**166**
	Complex	7				3				5	1	1	6	2						4	33	1		12		11				**86**
eval	Trivial												1																	**1**
	Moderate								1	1	1													6						**9**
	Complex	2			1	2	1				24					1	4			1	8			88		1				**133**
instance_eval	Easy										24													8		1				**33**
	Moderate														4		1				3	1		95						**104**
	Complex	1			3				1	5	2		1	8	11	5				2	13	20		63		5		2		**142**
	Moderate									1																				**1**
instance_exec	Complex	33									2		6		1						22			29	2	8				**103**
	Trivial																							6						**6**
	Easy									2																	1			**3**
instance_variable_get	Moderate	1									10		1											9	4	10	2	4	2	**43**
	Complex	1	4							1	3	1		1						1	10	5	1	17		2	3			**50**
	Trivial									1														5						**6**
	Easy																											1		**2**
instance_variable_set	Moderate	1	1								16		1		1						12	5		57	16	5	2	1		**118**
	Complex		2			1	1		5	1	3					1	1			1	8	4		22		3				**53**
send	Trivial	9		1	1		4	4	3	3	5		5	2	1		1			17	22	3		9	7	1	15	1		**114**
	Easy	4	4		5	1		16		25	9		6			14				7	29	1		22	7		44			**194**
	Moderate	10	4		3	1	11		3	20	67	8	23	2	28	5	3	3		6	55	6	1	43	14		49	5	2	**372**
	Complex	39	14	2	1	12	5	1	24	6	67	3	16	4	13	10	4	2		24	173	40	9	45	13	12	47	5		**591**

On one hand, there are dynamic features that have a low percentage of *complex* classification, e.g., `define_method` (34.13%) and `instance_variable_set` (29.61%). These statements provide more specific features and hence it is more practical to track their parameters. On the other hand, there are dynamic features that have a high percentage of *complex* classification, e.g., `eval` (93%) and `instance_exec` (99.04%). Almost all instances of these features depend on variable values. For example, we have identified several cases of `eval` depending on variable values to know what will be executed and also in which context it will be executed. In these cases, the analysis of the values of the variables may not be trivial.

As reported in Table 6, we compare the number of dynamic features that are trivial, easy, or moderate to be converted to static code with those that are complex in our dataset and in parts of it, e.g., only `send` statements. The results indicate—globally and for almost every feature—that the number of trivial, easy, or moderate statements to remove is quite similar to the number of complex ones. Particularly for `const_get`, `eval`, and `instance_exec`, we can note that their statements are likely to be complex to remove. We believe that those are important effects to pursue, which requires a few more experiments (future work).

Table 6: Complexity to remove dynamic features

Target set	**Trivial, easy, or moderate**	**Complex**
All dynamic features	1,417 (49.74%)	1,432 (50.26%)
`class/module_eval`	149 (47.00%)	168 (53.00%)
`class/variable_get`	1 (100%)	0 (0%)
`class/variable_set`	2 (66.67%)	1 (33.33%)
`const_get`	32 (28.83%)	79 (71.17%)
`const_set`	61 (70.11%)	26 (29.89%)
`define_method`	166 (65.87%)	86 (34.13%)
`eval`	10 (6.99%)	133 (93.01%)
`instance_eval`	137 (49.10%)	142 (50.90%)
`instance_exec`	1 (0.96%)	103 (99.04%)
`instance_variable_get`	52 (50.98%)	50 (49.02%)
`instance_variable_set`	126 (70.39%)	53 (29.61%)
`send`	680 (53.50%)	591 (46.50%)

Domains: In a combined analysis, we grouped the subject projects into the following four domains:

- *Applications*: Capybara, DevDocs, diaspora*, Discourse, Huginn, Jekyll, Octopress, Rails Admin, Spree, Vagrant.
- *Frameworks*: Capistrano, Grape, Sass.
- *Libraries/Plug-ins*: ActiveAdmin, Cancan, devise, Paperclip, Resque, Simple Form, Whenever.
- *Managers*: Carrierwave, CocoaPods, FPM, GitLab, Homebrew, Homebrew-Cask.

We did not group the *Ruby* and the *Rails* projects—i.e., we analyzed them individually—since they have a large number of dynamic statements, which may skew the results since only Ruby and Rails are responsible for 48.5% of the analyzed statements in this paper.

Table 7 reports out results (we omit lines only with zeroes). Actually, the percentage of dynamic statements that are not complex to remove are 60.75%, 61.11%, 33.23%, 54.62%, 38.23%, and 50.18% in *Applications*, *Frameworks*, *Libraries/Plug-ins*, *Managers*, *Rails*, and *Ruby*, respectively.

Table 7: Complexity analysis per domain (zeros are omitted)

		Applications	**Frameworks**	**Libraries/Plugins**	**Managers**	**Rails**	**Ruby**
`class/module_eval`	Trivial			2		8	5
	Moderate	34	8	13	6	31	42
	Complex	10	1	1	9	55	92
`class_variable_get`	Moderate			1			
`class_variable_set`	Moderate			1	1		
	Complex					1	
`const_get`	Trivial	3	2	1	1	2	3
	Easy						1
	Moderate	4	2	2	2	2	7
	Complex	8	2	12	9	20	28
`const_set`	Trivial	1	1		1	3	44
	Moderate		1			2	8
	Complex	4		1	4	6	11
`define_method`	Moderate	58	1	6	24	38	39
	Complex	17	6	11	7	33	12
`eval`	Trivial				1		
	Moderate	3					6
	Complex	30		3	4	8	88
`instance_eval`	Easy	25					8
	Moderate	2			4	3	95
	Complex	35	8	6	17	13	63
`instance_exec`	Moderate	1					
	Complex	11	6	35		22	29
`instance_variable_get`	Trivial						6
	Easy	1		2			
	Moderate	20	2	1	1	9	10
	Complex	14		8	1	10	17
`instance_variable_set`	Trivial	1					5
	Easy	2					
	Moderate	28	16	3	2	12	57
	Complex	12		8	3	8	22
`send`	Trivial	33	10	30	10	22	9
	Easy	114	7	15	7	29	22
	Moderate	156	16	26	76	55	43
	Complex	173	19	122	59	173	45

On one hand, the *Applications* domain has a low percentage of the complex classification (39.25%), which means that regular developers could avoid the overuse of dynamic features. Ruby (which was not grouped in the *Applications* domain) is quite different by having almost half of its features classified as complex (49.82%). *Managers*—which mostly groups applications that manage package dependencies—also has almost half of its dynamic features classified as complex (45.38%). As an unexpected finding, except for the Rails framework that has 61.77% of its features classified as complex, the *Frameworks* domain has only 38.89% of its features classified as complex, which reveals that frameworks rely on dynamic features with a manageable level of flexibility (at least in the three systems of such domain). Finally, *Libraries/Plug-ins* is the domain with more dynamic features classified as complex (66.77%), which evidences the flexibility that libraries and plug-ins must have.

We use the same statistical procedure as above to compare the number of dynamic features that are trivial, easy, or moderate to be converted to static code with those that are complex

considering our domains. Table 4.2 reports the results. Since the data of any domain follow a normal distribution, we rely on Wilcoxon Signed-Rank Tests to check the null hypothesis of the means remain essentially unchanged. We rejected the null hypothesis only for the *Libraries/Plug-ins* domain and accepted the alternative hypothesis H_n, which means that features in this domain are most complex to remove.

Table 8: Statistical analysis on the complexity to remove dynamic features per domain

Target set	**Normal Distribution?**	**Null hypothesis, H_0**	**Alternative hypothesis**
Applications	no (p-value¡$2.2e^{-16}$)	accept (p-value=0.06107)	—
Frameworks	no (p-value=$1.236e^{-8}$)	accept (p-value=0.4079)	—
Libraries/Plug-ins	no (p-value=$4.348e^{-15}$)	**reject** (p-value=0.01086)	**accept** H_n (more complex)
Managers	no (p-value=$1.565e^{-11}$)	accept (p-value=0.6587)	—

4.3 Final Remarks

Applications that require a high level of flexibility are likely to have more dynamic statements that are complex to remove. Although frameworks also need to be very generic, our result presented a low percentage of complex dynamic statements to be removed, although it represents the category with the small number of systems (three).

Our complexity levels may be used as the basis for creating tools to convert dynamic statements into static ones. Dynamic statements classified as trivial or low were mostly invoked from static values and hence the transformation is straightforward and safe. Those classified as moderate, on the other hand, were usually invoked from dynamic values. Therefore, we need an algorithm to infer the values a variable assumes to automatically make these transformations, which are likely to not be safe. Last, those classified as complex would tie in to the difficulty of actually writing tools to make these transformations.

5 When do developers use dynamic features? (RQ #3)

This research question aims to identify, classify, and illustrate the scenarios where developers opt for dynamic statements instead of static ones.

5.1 Methodology

We manually inspected the 2,849 dynamic statements from the subject systems. Aided by our supporting tool, for each dynamic feature, the first author of this paper deeply analyzed their statements aiming to group them in scenarios.

5.2 Result

Table 9 reports the 39 identified scenarios by dynamic feature. It also provides a description of each scenario and also reports the complexity in removing its respective dynamic statements. In

this section, we properly describe the ten most common scenarios, besides providing a concrete example of the most common complexity level in each scenario.

Table 9: Scenarios (zeros are omitted)

Feature	Scenario	Description	Trivial	Easy	Moderate	Complex	Total
class_eval module_eval	CE #1 (block execution)	Executes statements in a class context	12		26	46	84
	CE #2 (method definition)	Dynamically builds methods	3		93	114	**210**
	CE #3 (private access)	Overpasses visibility of methods			15	8	23
class_ variable_get	CVG #1 (change-prone variable)	Gets a class variable from a variable value			1		1
class_ variable_set	CVS #1 (change-prone variable)	Sets a class variable from a variable value			1	1	2
	CVS #2 (private access)	Overpasses visibility of class variables			1		1
const_set	CS #1 (static values)	Defines constants from static values	48				48
	CS #2 (change-prone variable)	Creates constants from a variable value			3	26	29
	CS #3 (array)	Defines many constants from an array	2		8		10
const_get	CG #1 (static values)	Gets public constants from static values	12				12
	CG #2 (change-prone variable)	Gets constants from a variable value			8	52	60
	CG #3 (array)	Gets many constants from an array			11	27	38
	CG #4 (private access)	Overpasses visibility of constants		1			1
define_method	**DM #1 (array)**	Defines many methods from an array			97	7	**104**
	DM #2 (events)	Dynamically builds methods from invoked functions			69	79	**148**
eval	EV #1 (method definition)	Dynamically builds methods	1		5	12	18
	EV #2 (change-prone variable)	Executes code from a variable value			1	78	79
	EV #3 (private access)	Overpasses visibility of methods				11	11
	EV #4 (scope)	Executes code in a class context				22	22
	EV #5 (class definition)	Dynamically builds classes			2	3	5
	EV #6 (variable definition)	Dynamically builds variables			1	7	8
instance_eval	**IEV #1 (private access)**	Overpasses visibility of methods/variables		33	64	39	**136**
	IEV #2 (method definition)	Dynamically builds methods			38	10	48
	IEV #3 (block execution)	Executes statements in an object context			2	93	**95**
instance_exec	IEX #1 (block execution without parameters)	Executes statements without external values in an object context			1	42	43
	IEX #2 (block execution with parameters)	Executes statements with external values in an object context				61	61
instance_ variable_get	IVG #1 (static values)	Gets public inst. variables from static values	6				6
	IVG #2 (change-prone variable)	Gets inst. variables from a variable value			12	23	35
	IVG #3 (array)	Gets many instance variables from an array			5	17	22
	IVG #4 (private access)	Overpasses visibility of variables		3	26	10	39
instance_ variable_set	IVS #1 (static values)	Sets static values in public inst. variables	6				6
	IVS #2 (change-prone variable)	Sets instance variables from a variable value			15	14	29
	IVS #3 (array)	Sets many instance variables from an array			7	23	30
	IVS #4 (private access)	Overpasses visibility of instance variables		2	72	11	**85**
	IVS #5 (variable definition)	Creates an instance variable			24	5	29
send	**SD #1 (static values)**	Invokes public methods from static values	113				**113**
	SD #2 (arrays)	Invokes many methods using an array	1	2	137	98	**238**
	SD #3 (change-prone variable)	Invokes methods from a variable value		9	188	439	**636**
	SD #4 (private access)	Overpasses visibility of methods		183	47	54	**284**

SD #3 (change-prone variable): This is the most common scenario, when developers invokes methods from variable values in order to provide more flexibility for methods. From 636 statements, nine were classified as *easy*, 188 as *moderate*, and 439 as *complex*. The large number of *complex* statements is due to the complexity to assess every possible value of a particular variable. For instance, Listing 2 illustrates a piece of code from Spree that dynamically invokes methods named

by the variable `name` (line 4). However, in our analysis we could not precisely determine the values of such variables and this particular instance was hence classified as complex.

```
1 # (from core/app/models/spree/preferences/preferable.rb, Spree)
2 def get_preference(name)
3   ...
4   send self.class.preference_getter_method(name)
5 end
```

Listing 2: `send` from change-prone variable (SD #3) classified as complex

SD #4 (private access): This scenario occurs when developers want to overpass visibility of methods. From 284 statements, 183 were classified as *easy*, 47 as *moderate*, and 54 as *complex*. The large number of *easy* statements is because most statements in this scenario require only changing the method visibility to public. Listing 3 illustrates a piece of code from Active Admin that uses `send` to call a private method named `children=` (line 4). Therefore, the conversion to static statements is *easy* by requiring only the change of the visibility of method `children=` to public (we do not illustrate this change due its simplicity).

```
1 # (from lib/active_admin/menu.rb, Active Admin)
2 def _add(options)
3   item = ActiveAdmin::MenuItem.new(options)
4   item.send :children=,
5     self[item.id].children if self[item.id]
6   self[item.id] = item
7 end
```

Listing 3: `send` from private access (SD #4) classified as easy

SD #2 (array): This scenario occurs when developers must call many methods and prefer to do it with less code. From 238 statements, one was classified as *trivial*, two as *easy*, 137 as *moderate*, and 98 as *complex*. Listing 4 illustrates a piece of code from GitLab that dynamically calls methods `name`, `username`, `skype`, `linkedin`, and `twitter` using `send` over an array (lines 3-6). The conversion to static code requires developers to locally (in this case) inspect the values of the array and then manually call each method (lines 11-20). This instance was classified as *moderate* because new statements have been included.

```
1  #before (from app/models/user.rb, GitLab)
2  def sanitize_attrs
3    %w(name username skype linkedin twitter).each do |attr|
4      value = self.send(attr)
5      self.send("#{attr}=",
6        Sanitize.clean(value)) if value.present?
7  end
8
9  #after
10 def sanitize_attrs
11   value = self.name
12   self.name = Sanitize.clean(value) if value.present?
```

```
13    value = self.username
14    self.name = Sanitize.clean(value) if value.present?
15    value = self.skype
16    self.name = Sanitize.clean(value) if value.present?
17    value = self.linkedin
18    self.name = Sanitize.clean(value) if value.present?
19    value = self.twitter
20    self.name = Sanitize.clean(value) if value.present?
21  end
```

Listing 4: `send` from arrays (SD #2) classified as moderate

CE #2 (method definition): This scenario occurs when developers dynamically build methods. From 210 statements, three were classified as *trivial*, 93 as *moderate* and 114 as *complex*. Listing 5 illustrates a piece of code from Spree where a method named by the value of variable `name` is dynamically created. It also depends on variables `blk` and `method`, which might be the method body. Therefore, the conversion to static statements require developers to analyze all possible values of such variables. In our analysis, we could not precisely determine these values and this particular instance was hence classified as *complex*.

```
1  # (from core/lib/spree/core/delegate_belongs_to.rb, Spree)
2  def class_def(name, method=nil, &blk)
3    class_eval {
4      method.nil? ? define_method(name, &blk) :
5                    define_method(name, method)
6    }
7  end
```

Listing 5: `class_eval` from method definition (CE #2) classified as complex

DM #2 (events): This scenario occurs when developers dynamically build methods when particular functions are invoked. From 148 statements, 69 were classified as *moderate* and 79 as *complex*. The large number of *complex* statements is due to the need to assess possible values of a particular variable. For instance, consider the piece of code from the Discourse project illustrated in Listing 6. The conversion to static statements requires developers to analyze all possible values of variables `klass`, `attr`, and `block`, which is not trivial. In our analysis, we could not precisely determine these values and this particular instance was hence classified as *complex*.

```
1  # (from lib/plugin/instance.rb, Discourse)
2  def add_to_class(klass, attr, &block)
3    klass = klass.to_s.classify.constantize
4    hidden_method_name = :"#{attr}_without_enable_check"
5    klass.send(:define_method, hidden_method_name, &block)
6    plugin = self
7    klass.send(:define_method, attr) do |*args|
8      send(hidden_method_name, *args) if plugin.enabled?
9    end
10 end
```

Listing 6: `define_method` from events (DM #2) classified as complex

IEV #1 (private access): Again, this scenario occurs when developers want to overpass visibility of methods. From 136 statements, 33 were classified as *easy*, 64 as *moderate*, and 39 as *complex*.

The analysis of `instance_eval` requires developers to analyze all possible classes of the variable, since we need to know the context where the code will be executed. More important, this task is never trivial and hence this scenario obtained results ranging from *easy* to *complex*. Listing 7 illustrates a piece of code from Rails that accesses the private method `remove_const` of module `Mime` (line 4). Particularly in this case, the conversion to static code requires the Ruby language to include a public version of such a method, since it is private by its nature (lines 10-12). This instance was, therefore, classified as *moderate* because new statements have been included (three additional lines).

```
1  #before
2  #(actionpack/lib/action_dispatch/http/mime_type.rb, Rails)
3  def unregister(symbol)
4    Mime.instance_eval { remove_const(symbol) }
5  end
6
7  #after
8  module Mime
9    ...
10   def public_remove_const(symbol)
11     remove_const(symbol)
12   end
13   ...
14 end
15
16 def unregister(symbol)
17   Mime.public_remove_const(symbol)
18 end
```

Listing 7: `instance_eval` from private accesses (IEV #1) classified as moderate

SD #1 (static values): This is the simplest scenario when developers invokes public methods from static values. Due to this ordinariness and no introduction of additional lines, we classified its 113 statements as *trivial*. Listing 8 illustrates a piece of code from the Discourse project that calls method `include` using `send` (lines 2-3). In the same listing, we illustrate static code that provides the same behavior (lines 6-7). Note that the conversion is straightforward by calling the method in direct form.

```
1 #before (from config/initializers/i18n.rb, Discourse)
2 I18n::Backend::Simple.send
3   (:include, I18n:Backend:Pluralization)
4
5 #after
6 I18n::Backend::Simple.include
7   (I18n:Backend:Pluralization)
```

Listing 8: `send` from public methods (SD #1) classified as trivial

DM #1 (array): This scenario occurs when developers must define many methods with similar syntaxes and prefer to do it with less code. From 104 statements, 97 were classified as *moderate* and seven as complex. Listing 9 illustrates a piece of code from the Discourse project that creates several methods with a similar syntax using an array of static values (lines 3-6). The conversion to static statements requires developers to inspect values of the array and then manually declare a

method for each one (lines 9-23). Particularly, this instance was classified as moderate because new statements have been included.

```
1  #before
2  # (from app/models/topic_status_update.rb, Discourse)
3  %w(pinned_globally pinned autocoled closed).each
4  do |status|
5    define_method("#{status}?") {name == status}
6  end
7
8  #after
9  def pinned_globally?
10     "pinned_globally" == name
11 end
12 def pinned?
13     "pinned" == name
14 end
15 def autoclosed?
16     "autoclosed" == name
17 end
18 def closed?
19     "closed" == name
20 end
21 def pinned_globally?
22     "pinned_globally" == name
23 end
```

Listing 9: `define_method` from array (DM #1) classified as moderate

IEV #3 (block execution): This scenario occurs when developers want to execute a block of statements inside an object context. From 95 statements, two were classified as *moderate* and 93 as *complex*. Once again, the large number of *complex* statements is due to the need to assess every possible value of a particular variable, which may not be a trivial task (the number of values to be considered can be very large or even infinite if the input is given by the user). Listing 10 illustrates a piece of code from the Homebrew project that executes a block in the context of the `bottle_specification` variable (line 3). As mentioned before, the conversion to static statements requires developers to analyze all possible values of the block and all possible classes that `bottle_specification` can be (which might even require a type inference algorithm). In our analysis we could not precisely determine these values and this particular instance was classified as *complex*.

```
1  # (from lib/Homebrew/software_spec.rb, Homebrew)
2  def bottle(&block)
3    bottle_specification.instance_eval(&block)
4  end
```

Listing 10: `instance_eval` from block execution (IEV #3) classified as complex

IVS #4 (private access): This scenario occurs when developers want to overpass visibility of instance variables. From 85 statements, two were classified as *easy*, 72 as *moderate* and 11 as *complex*. The large number of *moderate* occurrences is due to the need of new statements in the target class to change the visibility of the variable to public. Listing 11 illustrates a piece of code

from the Sass project, where variable `elements` is accessed by the dynamic feature (lines 6-7). Therefore, the conversion to static code requires the creation of a public setter method.

```
1 # from (lib/sass/script/tree/list_literal.rb, Sass)
2 class ListLiteral < Node
3   attr_reader :elements
4   def deep_copy
5     ...
6     node.instance_variable_set('@elements',
7                                elements.map {|e| e.deep_copy})
8   end
9 end
```

Listing 11: `instance_variable_set` from private access (IVS #4) classified as moderate

5.3 Final Remarks

As the most representative findings, we could enumerate the following:

- Dynamic features whose parameters are static values represent only 6.49% of the total of dynamic features. Each of these statements has been classified as trivial.
- A considerable percentage of the occurrences (20.36%) refers to features dynamically overpassing the visibility of methods, variables, and constants. We claim that these cases are likely to evidence flaws in the architectural design since more than 77% of them could be easily converted to static code by only changing their visibility.
- Scenarios that depend on external values are mostly classified as complex. For example, scenarios *change-prone variables* and *block execution* have, respectively, 72.68% and 85.51% of their instances classified as complex. In fact, dynamic code that enables user-provided extensions (e.g., Homebrew, where users can create new packages to install, or frameworks with extension points for user-provided code) will naturally be complex since values for these dynamic statements will come from this user-provided code.
- Although 53.5% of the `send` feature can be easily transformed to static code, such refactoring usually generates more lines of code, which may cause loss of readability. For instance, there were cases in our dataset in which removing a `send` feature would generate dozens of new lines of code. Disregarding any other aspect, this indeed reduces lines of code and saves programming time.

6 Why do developers use dynamic features? (RQ #4)

This research question investigates the advantages and disadvantages of dynamic features, which allows us to conjecture why developers use them.

6.1 Methodology

Our main goal is to determine what motivates developers to use dynamic features. We hence investigated a small subset of dynamic usages to come up with the following five motivations:

1. unusual coding style,
2. overpassing privacy restrictions,
3. defining new structures,
4. contextualizing block executions, and
5. generalizing code tasks.

Next, we analyzed the dynamic usages of the 39 scenarios of RQ #3 to come up with a direct mapping, i.e., we identified the main underlying motivation for each scenario.

6.2 Results

Table 10 presents the mapping between the scenarios and motivations. The *unusual coding style* motivation may be related with old patterns. It represents the occurrences of dynamic features that receive static values. Since there are only *trivial* cases in the scenarios mapped to this motivation, we claim that when developers are guided by this motivation they should rely on static features instead.

Table 10: Motivations

Motivation	Related Scenarios	Description
Unusual coding style	CS—CG—IVG—IVS—SD #1 (static values)	Dynamic statements that receives a static values in the parameters
Overpassing privacy restrictions	IEV #1, CVS #2 , CE—EV—ME #3, IVS—SD #4 (private access)	The use of dynamic statements to overpass privacy restrictions
Defining new structures	EV #1, CE—ME—IEV #2 (method definition) DM #2 (events) EV #5 (class definition) IVS #5, EV #6 (variable definition)	Dynamic builds of methods, classes and variables
Contextualizing block executions	CE—ME #1, IEV #3 (block execution) EV #4 (scope) IEX #1 (block without parameters) IEX #2 (block with parameters)	The use of dynamic statements to execute a block of statements in a object or class context
Generalizing code tasks	CVG—CVS #1, CS—CG—EV—IVG—IVS #2, SD #3 (change-prone variable) DM #1, SD #2, CS—CG—IVG—IVS #3 (array)	the use of dynamic statements to provide a agile way to invoke, build, get, and set methods, variables and constants

The *overpassing privacy restrictions* motivation represents the use of dynamic statements to overpass privacy restrictions, which reveals flaws in the architectural design. When it refers to resources of the same project, we claim that developers should revise the visibility modifiers instead. However, there were cases where the violation of privacy restrictions refers to external libraries, which might be seen as defensible due to the difficulty in contacting third-party developers.

The *defining new structures* motivation represents dynamic builds of methods, classes and variables. Such motivation may provide good support for the development of frameworks, for instance, dynamic builds of methods to automate the process of storing values of variables mapped directly to a database column.

The *contextualizing block executions* motivation represents the use of dynamic features to execute a block of statements in an object or class context, which in some cases may reveal an absence of methods to handle a particular case. Developers are mostly guided by this motivation when developing libraries to dynamically call blocks provided as formal parameters, e.g., a method that needs to run a given block inside it.

Finally, the *generalizing code tasks* motivation represents the use of dynamic statements to provide an agile way to invoke, build, get, and set methods, variables and constants. This motivation represents the majority uses of dynamic statements and has at least two advantages: (i) reducing code duplication, e.g., methods with the same structure may be generalized into just one method that dynamically creates them; and (ii) flexibility, i.e., to write a few statements that will be responsible for adapting the code in several scenarios (for example, dynamic features may be used to allow the software to operate on any database model).

6.3 Final Remarks

The advantages of using dynamic statements—such as flexibility, good support for the development of frameworks, and reducing code duplication—are provided by the *defining new structures*, *contextualizing block executions*, and *generalizing code tasks* motivations. Nevertheless, the *overpassing privacy restrictions* is an alternative to access a library's functionality rather than an advantage. The *unusual coding style* motivation is even more concerning since it is guided by old patterns where, now, a single static statement could do exactly the same.

7 Threats to Validity

We must state at least three threats to validity of the reported evaluation. First, our study investigated the use of dynamic features in 28 Ruby projects. As usual in empirical studies in software engineering, we cannot claim that our approach will provide equivalent results in other systems or other dynamically typed languages (*external validity*). However, we argue there is heterogeneity among the projects since they have been developed by different teams and they belong to different domains, as presented in Section 4.

Second, we relied on a single experienced Ruby developer (and also first author of this paper) for the classification (*construct validity*). To mitigate this threat, our entire dataset is publicly available, which means that researchers can access and double check our classifications.

Third, dynamic features could interact in unusual ways with other dynamic features and consequently undermine our classification (*internal validity*). Even having a low probability, a dynamic

feature could add code that would move, e.g., one of the *moderate* occurrences elsewhere to *complex*. Especially for `eval` and variants, these statements can easily change the structure of a class (by adding new variables, changing visibility of methods, etc.). This behavior, therefore, can invalidate some of our manual analysis; for instance, changing the visibility of a method to private in a dynamic way could undermine a *trivial* classification, moving it to a *low* classification.

8 Related Work

We divided the related work into three groups: (i) *Empirical studies of Smalltalk*; (ii) *dynamic features in dynamically typed languages*; and (iii) *dynamic features in Ruby*.

8.1 Empirical studies of Smalltalk

Callaú et al. [3, 2] conducted a large and contributory study about dynamic statements. From 1,000 Smalltalk projects, each dynamic statement was gathered and classified as safe or unsafe. Basically, safe was when a dynamic statement receives a static parameter and unsafe was when it receives a dynamic parameter. Although there is no direct mapping between our complexity classification and safe/unsafe statements, we argue that most statements we classified as trivial or easy are likely to be safe, whereas most moderate or complex statements are likely to be unsafe. Callaú et al. demonstrated that some dynamic statements have more safe uses while others have the opposite result. However, in our study, except for `const_set`, all dynamic statements present a majority of uses as unsafe (moderate and complex, in our complexity analysis). They also reported that dynamic features are used in 1.76% of methods, which contrasts somewhat with our results since we reported that dynamic features are used in 5.74% of methods in a Ruby project, on overall average. Another discovery was that dynamic features classified as message sending (equivalent to `send` in our study) are the most used (30%), which corroborates our results since the use of `send` is responsible for 44.61% of the dynamic features analyzed in RQs #2 and #3. Furthermore, the authors also concluded that dynamic features are used in specific kinds of projects, such as core system libraries, development tools, language extensions, and tests. However, through our studies, we could not conclude whether dynamic Ruby statements are used in specific projects.

Callaú et al. [3, 2] also investigated the reasons developers use dynamic features and which types of dynamic feature usages can be refactorable. In order to answer these research questions, a subset of their data set (377 of the 20,387 dynamic statements) was manually checked. Some of their usage classifications are very similar to those stated in this paper, such as: (i) *convenience* (20%), which considers the use of reflection through dynamic statements (in our study it refers to scenarios *SD-2 (arrays)* and *SD-3 (change-prone variables)*, totaling 30.68%); (ii) *dynamic code management* (8.6%), which considers the use of dynamic statements to create code at run time (in our study it refers to all scenarios named as method, class, or variable definition, and all scenarios of define_method, totaling 20%); and (iii) *breaking encapsulation* (5.8%), which in our study refers to

all scenarios named as private access, totaling 20.35%). Furthermore, the authors also classified some cases of dynamic statements as essential (37.5%), which in general represents the cases where refactoring is complex (the number of values to be considered can be very large or even infinite if the input is given by the user). Compared to this study, 50.3% of the dynamic statements could *not* be converted to static statements, although, in the *Applications* and *Frameworks* domains, the results were similar to those of Callaú et al. (39.25% and 38.89%, respectively).

8.2 Dynamic features in dynamically typed languages

Based on 19 open-source PHP projects, Hills et al. [9] investigated which dynamic features are used, how often they are used, and how they are distributed across the files. Similarly to our study, the authors investigated if the dynamic statements are indeed dynamic. As the result, they could statically infer 61% of calls through variables while in our study we could statically infer 49.7% of the parameters of dynamic statements.

In another work, Hills [8] investigated the evolution of dynamic features in PHP projects. The authors classified dynamic statements in three categories: (i) *variable features*, (ii) *magic methods*, and (iii) `eval`. Connecting with our study, variable features and magic methods are similar to the use of `send` in Ruby, and the `eval` of PHP presents the same behavior of the `eval` in Ruby. He concluded that `eval` is being least used over time, but *variable features* and *magic methods* are being more used. Such results corroborate with ours since Ruby developers use `send` much more than `eval` (1,271 *vs.* 143 in our study).

Holkner and Harland [10] evaluated the dynamic behavior of Python based on 24 open-source projects. They concluded that the majority of projects rely on the `attr_add` function, which is a dynamic feature that adds an attribute inside of an object (in Ruby, the `attr` variants and `instance_variable_set` are the correspondent features). More important, they found that 90% of dynamic features occur during program startup, which indicates that the dynamic statements are being used to create methods, create variables, and call functions in a more rapid way, i.e., fewer lines of code than using static coding. In another work, Åkerblom et al. [1] evaluated the dynamic features in 19 open-source Python projects. Similarly to the previous paper [10], they report that most dynamic features (55% to be more specific) occur during program startup. Although we have not inspected where dynamic features occur, we indicate this as future work. They also report that dynamic features are more used in library code than in program specific code (69% *vs.* 31%). In our study, this observation also holds. If we gather the number of dynamic statements (Table 3) per our domains, we can infer that the *Libraries/Plug-ins* domain uses more dynamic features than the *Application* one (3.61% *vs.* 1.86%, on average).

Richards et al. [14] conducted a study about the dynamic behavior of JavaScript. Based on execution traces recorded from a large corpus of real-world systems, the authors analyzed how the dynamic features of JavaScript are being used. As the main finding, they provide evidence that `eval` is commonly used (nearly every site of their experiment have at least one `eval` usage). They

also argue that the use of `eval` to call methods and create structures dynamically is very powerful and clearly significant for the logic of programs. In our study, 14 out of 28 projects use at least one `eval` statement. Since the majority of `eval` calls in our study were classified as complex (93%), we claim that `eval` in Ruby projects is also clearly significant for the logic of programs.

8.3 Dynamic features in Ruby

Furr et al. [4] proposed the Ruby Intermediate Language, which lets Ruby code be easily extended and analyzed by tools and developers. For example, in our study, we found that many `eval` calls are complex to remove. RIL would catch the parameters of `eval` through dynamic analysis and replace the statement with all possible `eval` executions. This makes explicit what `eval` is doing and hence may be helpful for developers in maintenance tasks, for example.

Furr et al. [6] also designed PRuby, a tool that relies on a profiler to remove dynamic features. They reported that developers use more `eval` than `send` (27 *vs.* 15), in contrast to our study in which `send` was much more common (143 *vs.* 1,271). Using dynamic analysis in their study, the authors found that `eval` evaluated 426 distinct strings and `send` invoked 96 distinct methods. Although we did not determine such values in our study, we could replace 49.7% of dynamic statements with static code using only non-invasive static analysis.

9 Conclusion

This study investigated how developers use dynamic features based on 28 open-source Ruby projects. First, dynamic feature usage in Ruby ranges from 2.08% to 3.08% at a 95% confidence interval. We claim the more flexibility the system requires, the higher the usage of dynamic features (which reached up to 5.09% in Paperclip). When we disregard features that are ascribed to common programming practices—such as the standard use of require—almost 50% of dynamic usages refer to `send`. Thus, we claim that effort to remove dynamic features should be first addressed to `send`.

Second, on average 49.7% of the evaluated dynamic statements could be feasibly converted to static code. This promotes studies to remove unnecessary dynamic usages (and hence all their cons), specially when dealing with projects from the domains *Applications* and *Frameworks* where this number raises to more than 60%. These studies, however, should *not* primarily focus on `const_get`, `eval`, and `instance_exec` since their statements seem to be mostly complex to remove.

Third, we identified, classified, and also illustrated the ten most common scenarios where developers prefer to use dynamic features rather than static ones. Developers could benefit from such a qualitative and organized catalog to avoid the overuse of dynamic features, e.g., when accessing private members and then introducing flaws in the architectural design. This study also highlighted some cases in which refactoring to static code generates more lines of code, which may cause loss of readability.

Fourth, we have identified five motivations why developers use dynamic statements: *unusual coding style*, *overpassing privacy restrictions*, *defining new structures*, *contextualizing block executions*, and *generalizing code tasks*. While the last three motivations provide advantages (such as flexibility, good support on the development of frameworks, and reducing code duplication), we claim that the first two motivations (*unusual coding style* and *overpassing privacy restrictions*) must be avoided and hence developers should prefer static statements instead.

Acknowledgment

Our research is supported by CNPq (National Council for Scientific and Technological Development).

References

[1] Åkerblom, B., Stendahl, J., Tumlin, M., and Wrigstad, T. (2014). Tracing dynamic features in Python programs. In *11th Working Conference on Mining Software Repositories (MSR)*, pages 292–295. ACM.

[2] Callaú, O., Robbes, R., Tanter, É., and Röthlisberger, D. (2011). How developers use the dynamic features of programming languages: The case of Smalltalk. In *8th Working Conference on Mining Software Repositories (MSR)*, pages 23–32. ACM.

[3] Callaú, O., Robbes, R., Tanter, E., and Röthlisberger, D. (2013). How (and why) developers use the dynamic features of programming languages: The case of Smalltalk. *Empirical Software Engineering*, 18(6):1156–1194. Springer.

[4] Furr, M., An, J., Foster, J. S., and Hicks, M. (2009a). The Ruby intermediate language. In *5th Dynamic Languages Symposium (DLS)*, pages 89–98. ACM.

[5] Furr, M., An, J., Foster, J. S., and Hicks, M. (2009b). Static type inference for Ruby. In *24th Symposium on Applied Computing (SAC)*, pages 1859–1866. ACM.

[6] Furr, M., hoon (David) An, J., and Foster, J. S. (2009c). Profile-guided static typing for dynamic scripting languages. In *24th International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 283–300. ACM.

[7] Hanenberg, S. (2010). An experiment about static and dynamic type systems: Doubts about the positive impact of static type systems on development time. In *25th International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 22–35. ACM.

[8] Hills, M. (2015). Evolution of dynamic feature usage in PHP. In *22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 525–529. IEEE.

[9] Hills, M., Klint, P., and Vinju, J. (2013). An empirical study of PHP feature usage: a static analysis perspective. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 325–335. ACM.

[10] Holkner, A. and Harland, J. (2009). Evaluating the dynamic behaviour of Python applications. In *32nd Australasian Conference on Computer Science (ACSC)*, pages 19–28. ACM.

[11] Miranda, S., Rodrigues, E., Valente, M. T., and Terra, R. (2016). Architecture conformance checking in dynamically typed languages. *Journal of Object Technology*, 15(3):1–34. ETH Zurich.

[12] Palsberg, J. and Schwartzbach, M. I. (1991). Object-oriented type inference. In *6th International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 146–161. ACM.

[13] Ren, B. M., Toman, J., Strickland, T. S., and Foster, J. S. (2013). The Ruby type checker. In *28th Symposium on Applied Computing (SAC)*, pages 1565–1572. ACM.

[14] Richards, G., Lebresne, S., Burg, B., and Vitek, J. (2010). An analysis of the dynamic behavior of JavaScript programs. In *31st Conference on Programming Language Design and Implementation (PLDI)*, pages 1–12. ACM.