Domain Specific Warnings: Are They Any Better?

Andre Hora, Nicolas Anquetil, Stéphane Ducasse, Simon Allier

Domain Specific Warnings: Are They Any Better?

André Hora, Nicolas Anquetil, Stéphane Ducasse, Simon Allier
RMoD Team
INRIA, Lille, France
{*firstName.lastName*}*@inria.fr*

***Abstract*—Tools to detect coding standard violations in source code are commonly used to improve code quality. One of their original goals is to prevent bugs, yet, a high number of false positives is generated by the rules of these tools, *i.e.*, most warnings do not indicate real bugs. There are empirical evidences supporting the intuition that the rules enforced by such tools do not prevent the introduction of bugs in software. This may occur because the rules are too generic and do not focus on domain specific problems of the software under analysis. We underwent an investigation of rules created for a specific domain based on expert opinion to understand if such rules are worthwhile enforcing in the context of defect prevention. In this paper, we performed a systematic study to investigate the relation between generic and domain specific warnings and observed defects. From our experiment on a real case, long term evolution, software, we have found that domain specific rules provide better defect prevention than generic ones.**

I. Introduction

Tools to detect coding standard violations in source code such as FindBugs [1] are commonly used to ensure the quality of the source code. Over the years several tools have been made available to provide such analyses [2], [3], [4], [5]. The rules provided by these tools are usually created for generic purposes such as ensuring that classes and methods are commented, checking identifiers length (long enough to be significant), and of the methods (not too long), or searching an assignment that has no effect. These rules can be targeted towards multiple goals, such as reliability, portability or maintainability [6], but very few are focused on the domain of the software under analysis.

A significant percentage of violations (warnings) reported by these rules are false positives, that is to say, most violations do not indicate real bugs. There are empirical evidences supporting the intuition that bug finding tools, with these rules, do not prevent the introduction of bugs in software [7], [8], [9], [6], [10], [11]. In general, warnings and observed defects are independent, there is no correlation between them.

We hypothesized that this happens because the rules used are generic and are not focusing on domain-specific problems of the software under analysis. For example, studies indicate that the most prevalent type of bug is semantic or program specific [12], [13], [9]. These kinds of bugs cannot be easily detected by generic rules [7]. In addition, using domain specific rules is not easy since they must be defined by an expert of the domain under analysis, for each domain. The efficiency of such domain specific rules for defect prevention or reduction needs therefore to be demonstrated.

In this paper, we performed a systematic study to investigate the relation between, on one side, generic or domain specific warnings and, on the other side, observed defects. For that, we first consider the warnings reported by generic and domain specific rules. Then, we consider the warnings reported by generic and domain specific rules that better correlate with the presence of bug, that we call *top rules*. The study is performed on Seaside, a web application framework, that has been used and maintained for years and for which domain specific rules have been created [14]. The results show that domain specific rules provide a better defect prevention than generic ones.

The contributions of this work are: (1) replication of previous experiments as to the lack of correlation between generic warnings and bugs; (2) new experiments on the correlation between domain specific warnings and bugs; and, (3) comparison between the precision of generic and domain specific rules as to bug prediction. The replication of previous experiments (first contribution) was required because we work with an OO language and a bug-finding tool not yet covered by previous studies.

This paper is structured as follow. Section II gives an overview of the approaches used to assess and match warnings and defects in source code. These approaches are used to support our study. Section III details our research questions. Section IV presents our experiment setting. Section V details the results of the empirical study. In section VI we present the evaluation of the results and we discuss threats to validity of the experiment. Finally, Section VII discusses related work and we conclude the paper in Section VIII.

II. Assessing and Matching Warnings and Defects

This section describes the approach used to assess and match warnings and defects in source code. More specifically, we describe how to assign warnings and defects to software components such as methods or lines of code. Then, we present how to match warnings and defects to detect which warnings are pointing to lines with defects and

978-1-4673-2312-3/12/$31.00

which are not pointing, *i.e.*, the true and false positives and negatives.

A. Determining Warning and Defect

Previous research try to predict bugs at the levels of classes, methods, or lines of code [8], [15], [6], [9]. This work is about evaluating the relation between warnings and defects, which requires to match them to source code. Working at the level of line is considered more difficult but giving better results [7] because it gives a more detailed level of granularity. Tools to detect coding standard violations usually give warnings at the level of lines of code, therefore, it is best to identify bugs at the same level so that both information match. In contrast, a problem that may occur when adopting classes or methods level is the possibility of wrong matching. Consider for example matching bugs and warnings at the class level, a warning may occur in one method of the class and a bug in another method. It is not clear in this case whether a match between a warning and a bug at the class level is really meaningful. The same mismatch happens, to a lesser extent, at the method level. For this reason, in our research, we work at the level of lines of code.

To match warnings and defects, we must identify lines with defects or warnings:

- *Identifying lines with defects*. This is done by mining commit messages in the software history to find bug fix changes. Two approaches for this step are normally used: searching for keywords such as "Fixed" or "Bug" [16] and searching for references to bug reports [17]. Identifying a bug fix commit allows one to identify the code changes that fixed the bug and therefore, where the bug was located in the source code. A line of code is related to a defect if it is modified by a bug fix change, since to resolve a problem the line was changed or removed [7]. Such line is marked as defect related because a single bug (error) may be caused by several defects (lines of code).
- *Identifying lines with warnings*. Things are simpler for warnings as many code checking rules and tools work at the level of line, therefore, a warning points directly to the rule breaking line.

That way, each line of code may be marked as warning and/or defect related. If one considers that warnings should prevent errors, one is interested in the lines that have both markers, called a True Positive (TP), the warning truly identified a defect; a line marked only as defective is a False Negative (FN), the code checking tool and rules failed to point to a defect; a line marked only with a warning is a False Positive (FP), the code checking rule pointed to a line that has no defect; and, a line with neither warning nor defect is a True Negative (TN), the code checking tool and rules correctly ignored a line with no defect. Of course rules may be actually created with other purposes than detecting defects, but in this research field we focus on their ability to prevent bugs. We will come back on this issue later in our experiment by considering only rules that better correlate with the presence of bug (top rules). This approach is also used by [9], [6], [7] to assess the true positives.

B. Unique Lines of Code

Errors (bugs) happen at various moments in software life and are corrected at different time. One needs to consider many versions of a system to collect data on various errors and be able to meaningfully correlate warnings and defects. During the life of the system, lines are added, removed or changed for many reasons not all related to bug fixes. Line based rules will raise (or not) a warning for a line as long as the line remains unchanged. A defect will be marked on a line from the time it is corrected (actually, just before) back to the time it was created in this form. So one works with unique lines of code (ULoC) [6], [9].

A unique line of code is a line that remains unchanged, possibly across several versions of the system. It is created at one point in time (a version of the system) and ends when it is changed or deleted. Different ULoCs in the system, even if they are physically contiguous in the code, may have different extension in time depending when they are created or ended. To match warnings to defects at the line level, one needs to identify all the ULoCs in the system during the whole time period of the experiment. This is done by creating a graph that represents a method's history (a method history contains one or more method versions) in which each node represents a physical line of code and each edge represents a non-changed line between two method versions (see Figure 1). A path in the graph is a ULoC. This approach is similar to the result of the SVN *annotate* command, where it is possible to know in one method version when the line was last modified. This graph is also close to an *annotation graph* [18], [19], but the former just keeps track of non-changed lines while an annotation graph also keeps track of modified lines.

Before creating the graph, the source code of the methods is normalized such that blank lines and formating changes are ignored. By doing this, we avoid, for example, cases where a warning points to one line in one version and the same warning points to two lines in another version, due to changes of formating between versions.

ULoCs support the computation of true and false positives and negatives. Figure 1 shows an example of the generated graph representing a method history with four versions (four commits). We can see that the first physical line of code has never been modified in these four versions, therefore it makes a single ULoC (ULoC-1). The second physical line of code in version 1 has been modified in version 2, therefore it also makes a single ULoC (ULoC-2), and another one, ULoC-3, for the same physical line of code starting at version 2. Numbers in the top left corner of the

boxes are unique identifiers for the ULoCs. There are eleven of them in total.

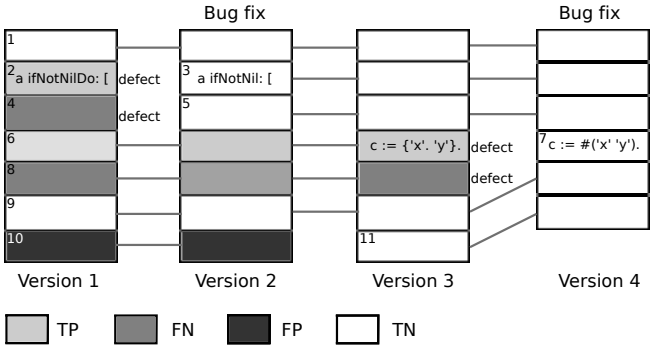

Figure 1. Example of graph representing a method history with four versions and with ULoCs marked with warnings and defects

In Figure 1 there are also two bug fixes, in version 2 and 4. The lines changed or removed to correct the bugs are marked with the word "defect". For illustration, the actual line of code is shown in the node just before and after the bug fix. Warning-only ULoCs (False Positive, see Section II-A) are presented in strong gray, defect-only ULoCs (False Negative) in medium gray, ULoCs with warning and defect (True Positive) in light gray, and ULoCs with neither markers (True Negative) are left in white. Thus, in Figure 1 there are 2 TPs, 1 FP, 2 FNs, and 6 TNs. One can measure the efficiency of a rule from the portion of warnings predicted correctly over all ULoCs marked with warnings (*e.g.* $TP/(TP + FP)$).

III. Research Questions

We want to assess the correlation between generic warnings and defects as well as between domain specific warnings and defects. Also, we want to study if domain specific rules are better bug predictors than generic rules. As part of this research we need to replicate previous experiments ([7], [8], [9], [6], [10]) on our case study, showing that generic coding standard rules generate many false positives with regard to bug prediction. We rephrase here our three contributions in the form of three questions. These questions will then be formalized into more specific research questions that will allow us to define formal hypotheses.

1) Can generic warnings be used for defect prevention?
2) Can domain specific warnings be used for defect prevention?
3) Are domain specific warnings more likely to point to defects than generic warnings?

We derive two versions of such questions. The first simply considers *all* generic rules and *all* domain specific rules. The second considers a more restricted set of generic and domain specific rules, the *top rules*.

A. Evaluating All Rules

The interrogations in this subsection take into account all rules (generic and domain specific). A first question is about the relation between generic warnings and defects. We want to know if these two variables are related or independent.

RQ1 *Is there a relation between generic warnings and defects?*

This question has already been answered, mostly negatively, in other papers. We replicate it here because we work with a language that was not considered previously (Smalltalk) and with a bug-finding tool similarly not studied (Smalllint).

We also want to know if there is a relation between domain specific warnings and defects and this motivates our second research question:

RQ2 *Is there a relation between domain specific warnings and defects?*

The previous questions analyze the relation between warnings and defects, but we also want to assess if domain specific warnings are better than generic warnings with respect to defect prevention.

RQ3 *Are domain specific warnings more likely to point to defects than generic warnings?*

B. Evaluating Top Rules

Considering all rules raises an issue with generic rules. These rules are somehow easier to define than the domain specific ones, because one can do it once for all, every addition is a "definitive" contribution. On the other hand, domain specific rules must be created for each new domain. With more generic rules, one should expect more warnings and consequently, as is typical in information retrieval, one should also expect better bug coverage (more bugs will be covered by at least one warning) but also lower precision, that is to say, each warning will have a lower probability of indicating a bug. So, we must expect that generic rules will naturally fare lower (*e.g.* RQ3) than domain specific ones, simply because they are more numerous and will give more hits. To have a fairer comparison, we will perform the same experiments for the top rules of the two sets, *i.e.*, the rules that better correlate with the presence of bug.

Thus, given selected groups of top generic and top domain specific rules we can ask the same questions as RQ1, RQ2, and RQ3.

RQ4 *Is there a relation between top generic warnings and defects?*

RQ5 *Is there a relation between top domain specific warnings and defects?*

RQ6 *Are top domain specific warnings more likely to point to defects top generic warnings?*

By answering such questions we can define which groups of rules are worthwhile for defect prevention in the case study under analysis.

IV. Experiment Setting

In this section we plan our experiment as suggested in [20]. We have two different experiments to setup, to answer research questions RQ1, RQ2, RQ4, and RQ5 on one hand, and to answer RQ3 and RQ6 on the other hand.

A. Context

The *context* of the experiment is real systems for which source code, commits logs linked to an issue database, and generic and domain specific rules to detect coding standard violations are available. We need real systems to ensure that our experiment is meaningful. Also, we need systems with commits linked to issue database and generic and domain specific rules to assess the relation between defects and warnings. Domain specific rules are defined by domain experts.

One difficulty of this research is to find systems that fulfill these requirements, and particularly for which a set of domain specific rules is defined. We selected Seaside to perform our empirical studies mainly because of its set of domain specific rules [14], but also because it has the advantage of being a real-world and non-trivial application, with a consolidated number of users and a relevant history of bugs. **Seaside**[1] is an open-source web application framework written in Smalltalk [21]. This system defines various internal domain-specific languages to configure application settings, nest components, define the flow of pages, and generate XHTML. We analyze the impact of domain specific rules for defect prevention when compared with generic rules on a long term evolution of Seaside. We analyze 943 snapshots of Seaside core, which were produced in almost four years of development (from November 2007 to September 2011). Table I presents an overview of the size of our case study.

Table I
Overview of Seaside core size

Number of snapshots	943
Average classes per snapshot	216
Average methods per snapshot	1,592
Average LOC per snapshot	6,428

Seaside includes *Slime* [14], a Seaside-specific program checker consisting of a set of rules working at the level of the abstract syntax tree (AST), that we call the *domain specific rules*. Smalltalk includes *Smalllint* [5], a generic program checker consisting of rules also working at the level of the AST, that we call the *generic rules*. Smalllint can be compared to other bug-finding tools such as FindBugs [1] and JLint[2] and it comes with rules targeting common bugs and code smells in Smalltalk.

[1]http://www.seaside.st

[2]http://jlint.sourceforge.net

B. Experiment for RQ1, RQ2, RQ4, RQ5

This first experiment is about the relation between warnings and defects. We want to know if these two variables are related or independent.

1) Hypotheses Formulation: The null and alternative hypotheses may be formalized as:

$H_0^{1,2,4,5}$ *Warnings and defects are independent.*

$H_a^{1,2,4,5}$ *Warnings and defects are related.*

2) Variable and Subject Selection: The *independent variable* is the warnings raised by tools to detect coding standard violations on lines of code. It is categorical and can take two values: *with warning* and *without warning*. The *dependent variable* in this study is the defects, which are raised by bug fix changes on lines of code. It is also categorical and can take two values: *with defect* and *without defect*.

The subjects for these experiments will be ULoCs from the case study chosen (Seaside) and we will measure the number of ULoCs in each of four categories: with warning/with defect (true positive), with warning/without defect (false positive), without warning/with defect (false negative), and without warning/without defect (true negative).

3) Experiment Design: To test the hypotheses $H^{1,2,4,5}$ we use the Chi^2 test, which can be used when there are two categorical variables, each with two or more possible values. The null hypothesis is that the frequencies for the dependent variable (defects) are the same for different values of the independent variable (warnings). If we cannot reject the null hypothesis, we must conclude that the variables are in fact independent. When we can reject the null hypothesis (*i.e.*, the variable are dependent), it is also important to understand how the variables are related. This is done by observing the Pearson residuals, which measure the difference between the observed and expected frequencies. When the absolute value of the residual is greater than two (> 2), one considers that the observed frequency is significantly higher than the expected and that more of the independent variable should induce more of the dependent one.

As is customary, the tests will be performed at the 5% significance level which means there will be a probability of 5% or less of erroneously rejecting the null hypothesis.

C. Experiment for RQ3, RQ6

1) Hypotheses Formulation: The null and alternative hypotheses may be formalized as:

$H_0^{3,6}$ *Domain specific and generic warnings are equally precise in identifying defects.*

$H_a^{3,6}$ *Domain specific warnings are more precise in identifying defects than generic warnings.*

Note that we make a directional (one-tailed) hypothesis. This should be made when there is evidence to support such a direction. This evidence will stem from the results of the first experiments where we will answer negatively to RQ1 (there is no relation between generic warnings and defects)

and positively to RQ2 (there is a relation between domain specific warnings and defects).

2) Variable and Subject Selection: The *independent variable* in this study is the group of rules (generic or domain specific). The *dependent variable* could have been the precision of the coding standard rules where precision is the percentage of ULoC with warning and defect among all ULoC with warning. However, we will show in the result section that we do not have enough rules with non-null precision to perform a test: many rules didn't give any warning, so precision is undefined for them, and other got warnings but not on ULoCs with defects, so precision would be null for them.

To bypass this issue, we will group ULoCs according to another criteria. We will consider all ULoCs in the history of a method as one subject. Another way to see it is to say that the combined versions of one method will be a subject. We considered working with "normal" methods as subject, that is to say one method in one version, but this would have the drawback that the same ULoC can appear in several versions of a method (if it does not change between these versions, *e.g.* see ULoC-1 in Figure 1), and therefore would have been counted more than once. By taking the whole history of each method, we avoid giving more weight to some ULoCs.

The metric used will be the Positive Predictive Value (PPV) which has the same formula as precision: proportion of ULoC with defects among those with warning. A high PPV indicates that the method, in its history, tends to have defects where it breaks some generic (or domain specific) coding standard rule.

3) Experiment Design: For this experiment, we use an unpaired setting, which means the methods composing one sample may not be the same than those composing the other sample. This is due to the fact that not all methods break generic and domain specific coding standard rules, many of them break either one or the other category of rule, and even fewer methods would have a non null PPV for both categories.

We test the hypotheses H^3 and H^6 with a Mann-Whitney test which is used for assessing whether one of two samples of independent observations tends to have larger values than the other. It can be used when the distribution of the data is not normal and there is different participants (not matched) in each condition. The null hypothesis is that the median PPV is the same for both samples.

Again the tests will be performed at the 5% significance level.

D. Instrumentation

1) Defects: For our research, we use the prediction at the level of lines of code because it is a more precise level of granularity and it also avoids the issues cited in Section II-A. To identify bug fix changes, we use the technique of searching for keywords, since Seaside has a normal practice of writing bug fix commits with the keyword "Issue". Seaside history contains $14,416$ ULoCs, from which 664 (4.6%) contained defects.

2) All Rules: With respect to the used rules, we considered two sets of rules, the first one with 91 generic rules and the second one with 29 domain specific rules. We considered only rules that work at the level of lines of code, excluding, for example, rules like "method too long". Below, we briefly describe the groups of generic rules. The number of individual rules by group is also showed.

- *Unnecessary code (20).* It targets code that is not needed or can be avoided (replaced) since other pieces of code can be more efficient or legible, *e.g.* an assignment that has no effect.
- *Spelling (5).* It looks at identifiers to find words wrongly spelled.
- *Possible bugs (20).* It targets general code that is considered likely to cause bugs, *e.g.* an unconditional recursion, or modification of a collection while iterating over it.
- *Pharo bugs (7).* It searches for code patterns specific to Pharo[3] (Smalltalk dialect used in this work) that could cause bugs, *e.g.* debugging code left in a method.
- *Bugs (7).* Another kind of code that can cause bugs, *e.g.* a missing super implementation, a method that overrides a "system" message.
- *Miscellaneous (13).* It searches for different patterns that, for example, a programmer coming from other languages might produce, *e.g.* in arithmetic expressions[4].
- *Intention revealing (19).* It searches for code related to the intention revealing pattern, *e.g.* a code that breaks the Law of Demeter, variable capitalization, or code using the wrong iterator.

The domain specific rules are separated into the following groups:

- *Portability (8).* Seaside runs without modification on 7 different platforms which differ slightly in both the syntax and the libraries they support [14]. Thus, this category targets code patterns specific to some platforms, *e.g.* code that uses dynamic arrays, or some specific methods/classes not portable across different Smalltalk dialects.
- *ANSI compatibility (8).* It targets code that is not ANSI compatible and is also related to the portability of Seaside.
- *Possible bugs (12).* It targets Seaside-specific code that is likely to cause bugs, *e.g.* code in which a given message is not the last in a specific sequence of method calls.

[3]www.pharo-project.org

[4]In Smalltalk arithmetic operators are normal methods, so "arithmetic expressions" are evaluated from left to right without operator precedence, $1+2*3$ is interpreted as (Java like notation) 1.add(2).times(3)=3.times(3)=9

- *Formatting (1).* It targets code in which a specific pattern must be followed, *e.g.* a correct pattern to deprecate an API protocol.

Some of these rules would clearly not be related to bug prevention (for example the spelling group), so we also experimented with the *top rules* as already introduced in Section III-B and are detailed in the next subsection.

Table II presents the number of warnings raised by all rules as well as the percentage of ULoC impacted.

Table II
SEASIDE WARNINGS FOR ALL RULES

Group	#Warnings	ULoC with warnings
All generic	1,118	7.7%
All domain specific	312	2.1%

3) Top Rules: Here we detail the approach used to determine the top rules. Table III shows all the rules that generate some warnings and for which at least one warning coincide with a defect, that is to say the rule for which $TP > 0$. This result confirms previous works, in which a subset of rules performs better than others [7], [9], [6], [10]. Rules prefixed by "GR" are domain specific rules and those prefixed by "RB" are generic rules.

Table III
RULES WITH TP > 0. RULES IN BOLD PERFORMED SIGNIFICANTLY BETTER THAN RANDOM (TOP RULES)

Rule	#Warning	#TP
GRAnsiCollectionsRule	8	1
GRAnsiConditionalsRule	**118**	**18**
GRAnsiStreamsRule	11	1
GRAnsiStringsRule	**40**	**10**
GRDeprecatedApiProtocolRule	56	3
GRNotPortableCollectionsRule	**7**	**4**
RBBadMessageRule	16	1
RBGuardingClauseRule	19	2
RBIfTrueBlocksRule	**7**	**2**
RBIfTrueReturnsRule	**14**	**3**
RBLawOfDemeterRule	**224**	**18**
RBLiteralValuesSpellingRule	232	10
RBMethodCommentsSpellingRule	216	8
RBNotEliminationRule	58	1
RBReturnsIfTrueRule	72	3
RBTempsReadBeforeWrittenRule	**16**	**3**
RBToDoRule	**38**	**6**

In fact, any random predictor, marking random lines with warnings, would, with a sufficient number of attempts, end up with a number of true positives higher than zero, but would not be very useful. Therefore, we can assess the significance of a rule by comparing it to a random predictor [9]. As suggested by [9], [6], this problem can be modeled as follows: the project is viewed as a large repository of lines, with a certain probability (p= #ULoC with defects / #ULoC) of those lines being defect related. A rule marks n lines with warnings. A certain number of these warnings (r) are successful defect predictions. This is compared with a random predictor, which selects n lines randomly from the repository. We can model the random predictor as a Bernoulli process (with probability p and n trials). The number of correctly predicted lines r has a binomial distribution; using the cumulative distribution function $P(TP \leq X \leq n)$ we compute the significance of the rule [6]. In conformance with our other statistical tests, we choose a 5% threshold[5]. When the random predictor has less than 5% probability to give a better result than the rule, we call this one a *top rule*. For example, for Seaside we have $14,416$ ULoCs and 664 with defects, so the probability of randomly picking a line with defect is $p = (664/14416) = 0.046$. For rule GRNotPortableCollectionsRule (Table III), $n = 7$ and the cumulative distribution function $P(4 \leq X \leq 7)$ is 0.0001, therefore we consider it a top rule. The top rules are presented in bold in Table III.

Table IV presents the number of warnings raised by the top rules as well as the percentage of ULoC impacted.

Table IV
SEASIDE WARNINGS FOR TOP RULES

Group	#Warnings	ULoC with warnings
Top generic	299	2.0%
Top domain specific	165	1.1%

V. EXPERIMENT RESULTS

In this section we present the results of our empirical study. We first present the results for all rules and answer RQ1, RQ2, and RQ3. We follow with the results for top rules and answer RQ4, RQ5, and RQ6.

A. Evaluating All Rules

The hypotheses for the Chi^2 test are derived from the one presented in Section IV-B.

RQ1 *Is there a relation between generic warnings and defects?*
H_0^1 *Generic warnings and defects are independent.*
H_a^1 *Generic warnings and defects are related.*

Table V shows the contingency table for generic warnings. The Chi^2 test gives a *p-value* $= 0.65$ (> 0.05 significance level), therefore, we cannot reject the null hypothesis that generic warnings and defects are independent, *i.e.*, the proportions of lines with and without generic warnings are the same in lines with and without defects.

Table VI shows the residuals for generic warnings, the values close to 0 indicate that there is no significant difference between the observed frequencies and the expected ones.

Results for all domain specific rules follow.

RQ2 *Is there a relation between domain specific warnings and defects?*

[5]Note however that this is not a statistical test of significance.

Table V
CONTINGENCY TABLE FOR GENERIC WARNINGS (#ULOCS)

	with defect	without defect	total
with warning	55	1,063	1,118
without warning	609	12,689	13,298
total	664	13,752	14,416

Table VI
RESIDUALS FOR GENERIC WARNINGS

	with defect	without defect
with warning	0.48	-0.10
without warning	-0.14	0.03

H_0^2 *Domain specific warnings and defects are independent.*
H_a^2 *Domain specific warnings and defects are related.*

Table VII shows the contingency table for domain specific warnings. The Chi^2 test gives *p-value* < 0.001. We can reject the null hypothesis with a very small probability of error, we conclude that domain specific warnings and defects are related, *i.e.*, the proportions of lines with and without generic warnings are not the same in lines with and without defects.

Table VII
CONTINGENCY TABLE FOR DOMAIN SPECIFIC WARNINGS (#ULOCS)

	with defect	without defect	total
with warning	37	275	312
without warning	627	13,477	14,104
total	664	13,752	14,416

The effect size is 0.051. Table VIII shows the residuals for domain specific warnings. One can see that condition *with warning and defect* is over-represented (> 2) and is the major contributor to the rejection of the null hypothesis.

We further conclude that defects appear more frequently on lines with domain specific warnings than on lines without warning.

Table VIII
RESIDUALS FOR DOMAIN SPECIFIC WARNINGS

	with defect	without defect
with warning	5.97	-1.31
without warning	-0.88	0.19

We now test whether domain specific warnings are better than generic warnings with respect to defect prevention. An evidence to support such a direction is the fact that we answered negatively to RQ1 and positively to RQ2.

RQ3 *Are domain specific warnings more likely to point to defects than generic warnings?*
H_0^3 *Domain specific and generic warnings have the same PPV.*
H_a^3 *Domain specific warnings PPV is higher.*

From all method histories, 509 had at least one generic warning and 175 had at least one domain specific warning. These will be our two samples in this test, the other methods having no warning have undefined PPV ($TP/(TP+FN) = 0\ /\ 0$).

Applying the Mann-Whitney test for such samples we have *p-value* $= 0.003$. The effect size is 0.1. We can reject the null hypothesis and say that domain specific PPV is higher than generic PPV, methods with domain specific warnings have more chance to have defects on these lines than those with generic warning.

We conclude that it is better to use domain specific warnings to point to defects than generic warnings.

B. Evaluating Top Rules

We also performed the same experiments on top rules.

RQ4 *Is there a relation between top generic warnings and defects?*
H_0^4 *Top generic warnings and defects are independent.*
H_a^4 *Top generic warnings and defects are related.*
RQ5 *Is there a relation between top domain specific warnings and defects?*
H_0^5 *Top domain specific warnings and defects are independent.*
H_a^5 *Top domain specific warnings and defects are related.*

Table IX and X show the contingency tables for top generic and domain specific warnings, respectively. The Chi^2 tests give *p-value* < 0.001 for both. We can reject null hypotheses H_0^4 and H_0^5.

We conclude that top generic warnings and top domain specific warnings are related to defects.

Table IX
CONTINGENCY TABLE FOR TOP GENERIC WARNINGS (#ULOCS)

	with defect	without defect	total
with warning	32	267	299
without warning	632	13,485	14,117
total	664	13,752	14,416

Table X
CONTINGENCY TABLE FOR TOP DOMAIN SPECIFIC WARNINGS (#ULOCS)

	with defect	without defect	total
with warning	32	133	165
without warning	632	13,619	14,251
total	664	13,752	14,416

The effect size is 0.042 for top generic and 0.076 for top domain specific warnings. Table XI and XII show the residuals for top generic and top domain specific warnings. We see that category *with warning and defect* is over-represented (> 2) and is the major contributor to the rejection of the null hypotheses.

We further conclude that defects appear more frequently on lines with top generic or top domain specific warnings than on lines without such warnings.

Table XI
RESIDUALS FOR TOP GENERIC WARNINGS

	with defect	without defect
with warning	4.91	-1.07
without warning	-0.71	0.15

Table XII
RESIDUALS FOR TOP DOMAIN SPECIFIC WARNINGS

	with defect	without defect
with warning	8.85	-1.94
without warning	-0.95	0.20

Finally, we test whether top domain specific warnings are better than top generic warnings with respect to defect prevention. An evidence to support such a direction is the fact that residual of the category *with warning and defect* is higher for the former than for the later.

RQ6 *Are top domain specific warnings more likely to point to defects top generic warnings?*
H_0^6 *Top domain specific and top generic warnings have the same PPV.*
H_a^6 *Top domain specific warnings PPV is higher.*

From all method histories, 77 had at least one top generic warning and 67 had at least one top domain specific warning. These will be our two samples in this test.

Applying the Mann-Whitney test we have *p-value* $= 0.047$. The effect size is 0.14. There is a significant difference between both samples and we can reject the null hypothesis.

We conclude that it is better to use top domain specific warnings to point to defects than top generic warnings.

VI. DISCUSSION

In this section we discuss the results of our experiments. We also present the threats to the validity of these experiments.

A. Evaluating All Rules

We studied the relation between warnings and defects. The outcome of our experiments is that generic warnings are not efficient to identify lines with defects (RQ1). This is coherent with the conclusions of previously published results [7], [9], [6], [10]. The fact that different bug-finding tools and programming languages were considered in other experiments and ours, reinforce the general validity of this conclusion.

This results is due to the great amount of false positives generated by generic rules. It hints at the importance of tailoring coding standard rules to a specific domain, which is confirmed by RQ2, showing that domain specific warnings and defects are dependent for the case study under analysis. In this case, we see a reduction of the amount of false positives.

Since RQ1 was rejected, and RQ2 accepted, the result provided by RQ3 is expected: PPV measured for domain specific rules is higher than for generic rules. We conclude that, for the case study under analysis, generic rules are not effective enough to be used for defect prevention. Domain specific rules give more relevant information on how to avoid bugs and therefore they are effective to be used for defect prevention.

B. Evaluating Top Rules

We also studied the relation between top warnings and defects. With this experiment, we are fairer to both groups of rules, since we select just the most effective rules for defect prevention, thus producing less false positives. The results of RQ4 and RQ5 show that both top generic and top domain specific warnings are related to defects, and thus can be used for defect prevention. This result is also confirmed by previous work, in which a subset of rules performed better than others [7], [9], [6], [10].

Contrary to RQ3, the result of RQ6 was not clear before hand because both RQ4 and RQ5 were accepted. Testing RQ6 shows that top domain specific PPV is (statistically) significantly higher than top generic PPV. Therefore, we can say that it is better to use top domain specific warnings to point to defects than top generic warnings. We conclude that, for the case study under analysis, top domain specific rules are more effective to be used for defect prevention than top generic rules.

C. Threats to Validity

1) Internal Validity: The matching between warnings and defects may be an underestimation: some bug fixes only introduce new code, such as the addition of a previously forgotten check clauses.

Overestimation is less likely: although not all lines that are part of a bug fix may be directly related to the bug, warnings on such lines still point out the area in which the bug occurred. These possible problems are also pointed out by [6], [9] since they also study defect prevention at the level of line of code.

Warnings might point to defects that have not been found. The influence of such dormant defects is minimized in the case of a long-running project as the one analyzed, where most of the defects will have been found.

Finally, we have not tried to identify instances of method renaming to receive the propagation of defects and warnings. If a method *foo* had previously been named *bar*, *bar* will not receive the propagation of defects and warnings from *foo*. However, 475 methods have been renamed during the experiment time frame (943 versions and $1,592$ methods per

version on average). Thus, 0.5 method renaming per version. Therefore, there is a very small amount of method renaming, which is hardly likely to impact on the validity of the results.

2) External Validity: We believe Seaside is a credible case study. It includes a large number of versions (943 collected over a time frame of almost four years), classes and methods representing real-world and non-trivial application, with a consolidated number of users and a relevant history of bugs. Despite this observation, our findings – as usual in empirical software engineering – cannot be directly generalized to other systems, specifically to systems implemented in other languages or to systems from different domains, even if a comparison with previous works [7], [8], [9], [6], [10] (which analyze systems implemented in other languages and from different domains) yielded similar results.

There are two requirements for the used approach that should be considered when replicating this study. The first is the existence of domain specific rules, the second is the possibility to link software repository and issue database. Many studies have successfully extracted such links before [7], [17], [19], [9], [6], suggesting that there is a general habit of clearly identifying bug fixes in many projects. In our experiment, we only considered defects stemming from bug fix commits. Note that in [7] the authors considered defects coming from bug fix and ordinary commits.

VII. Related Work

In recent years, some approaches have been proposed to study the relation between generic warnings and bugs. Boogerd *et al.* [6], [9] empirically assess the relation between violations of coding standard rules raised by MISRA C and faults, using coding standard rules for embedded C development on industrial cases. The authors have found that only 10 out of 88 rules for the case study presented in [9], and 12 out of 72 rules for the case study presented in [6] were significant predictors of fault location. From the set of found rules, both case studies agree only on one rule. These results suggest the importance of tailoring a coding standard to a specific domain, as the observed violation severity differs between projects [9]. This reinforces our results. Although the idea of our research is similar to their research, our focus is different. While the authors used generic rules to check the relation between warnings and bugs, we used and compared generic and domain specific rules.

Basalaj *et al.* [10] studied the link between QA C++ warnings and faults for snapshots from 18 different projects and found a correlation for 12 out of 900 rules. Wagner *et al.* [22] evaluated two Java bug-finding tools (FindBugs and PMD) on two different software projects, to evaluate their use in defect detection. Their study could not confirm this possibility for their two projects. More recently, Couto *et al.* [8] also showed that, at the level of methods, overall there is not a correspondence between the warnings raised by FindBugs and the methods changed to remove defects.

Another research close to our work is proposed by Kim and Ernst [7]. It aims at improving the ranking mechanism for warnings by mining software histories. For that the authors make use of *annotation graphs* [18], [19] to build the marking approach. Although they use the software history as input to the proposed algorithm, the empirical data reported take into account a simple version, differently from our approach that takes into account the whole history of the system. The approach was evaluated using three Java bug-finding tools (FindBugs, JLint and PMD) on three open source Java projects. Even though it is not the main goal of their work, they showed that a small subset of rules perform better than others in the context of bug prevention.

We note that even though several bug-finding tools are analyzed in different languages, related work agrees to conclude that, overall, bug-finding tools do not prevent the introduction of bugs in software. But we are not aware of any other work that study the relation between domain specific rules and defects.

In the context of domain specific rules, Renggli *et al.* [14] advocate their use to check domain-specific practices. Their empirical validation demonstrates that domain-specific program checking significantly improves code quality when compared with generic program checking [14]. The demonstration is done with two different rule sets working at different levels of domain abstraction in two long term evolution case studies (Seaside and Magritte) written in Smalltalk. In the context of generic rules, Araujo *et al.* [23] study the relevance of the warning reported by FindBugs and PMD in several Java systems. They conclude that better relevance (less false positives) can be achieved when FindBugs is configured in a proper way, *i.e.*, when the tool is configured to report warnings that make sense for the system under analysis. This matches our experiment with *top rules*. However, relating warnings and defects is also beyond the scope of their research.

VIII. Conclusion

To the best of our knowledge, this work is the first to study the use of domain specific rules for defect prevention. In this paper, we performed a systematic study to investigate the relation between generic or domain specific warnings and observed defects. The study was performed on Seaside, a real software, that has been used and maintained for years and for which domain specific rules have been created. Two groups of research questions were created to assess whether domain specific rules would be better bug predictors than generic rules. The first questions were about generic and domain specific rules, the second considered the top rules, a more focused set of generic and domain specific rules. All the results reported in this works were statistically significant, and not due to chance.

We conclude that, for the case study under analysis, generic rules were not effective enough to be used for defect prevention. This was also reported in previous publications. Domain specific rules give more relevant information on how to avoid bugs and therefore they are more effective to be used for defect prevention. Moreover, the top domain specific rules were more effective to be used for defect prevention than top generic rules. Therefore, the results showed that domain specific rules provide a better defect prevention than generic ones. With the results reported in this work, we expect domain specific rules to be created and used by developers in complement to generic ones for defect prevention.

As future work, we plan to expand this research to other systems. Yet finding systems with domain specific rules implemented is not an easy task, therefore we plan to invest in the creation of such rules according to the domain of some systems and based on experts opinions. We also plan to compare warnings and defects taking into account categories of rules, in complement to the overall and top rules comparison provided in this research.

ACKNOWLEDGMENT

This research has been supported by grants from Agence Nationale de la Recherche (ANR-2010-BLAN-0219-01).

REFERENCES

[1] D. Hovemeyer and W. Pugh, "Finding Bugs is Easy," in *Object Oriented Programming Systems Languages and Applications*, 2004, pp. 132–136.

[2] D. Engler, B. Chelf, A. Chou, and S. Hallem, "Checking system Rules Using System-specific, Programmer-Written Compiler Extensions," in *Symposium on Operating System Design & Implementation*, 2000, pp. 1–16.

[3] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata, "Extended Static Checking for Java," in *Conference on Programming Language Design and Implementation*, 2002, pp. 234–245.

[4] S. C. Johnson, "Lint, a C Program Checker," in *Unix Programmers Manual*, 1978, pp. 292–303.

[5] D. Roberts, J. Brant, and R. Johnson, "A Refactoring Tool for Smalltalk," *Theory and Practice of Object Systems*, vol. 3, pp. 253–263, 1997.

[6] C. Boogerd and L. Moonen, "Assessing the Value of Coding Standards: An Empirical Study," in *International Conference on Software Maintenance*, 2008, pp. 277 –286.

[7] S. Kim and M. D. Ernst, "Which Warnings Should I Fix First?" in *European Software Engineering Conference and the ACM SIGSOFT Symposium on The foundations of Software Engineering*, 2007, pp. 45–54.

[8] C. Couto, J. E. Montandon, C. Silva, and M. T. Valente, "Static Correspondence and Correlation Between Field Defects and Warnings Reported by a Bug Finding Tool," *Software Quality Journal*, pp. 1–17, 2012.

[9] C. Boogerd and L. Moonen, "Evaluating the Relation Between Coding Standard Violations and Faults Within and Across Software Versions," in *Working Conference on Mining Software Repositories*, 2009, pp. 41–50.

[10] W. Basalaj and F. van den Beuken, "Correlation Between Coding Standards Compliance and Software Quality," Programming Research, Tech. Rep., 2006.

[11] T. Kremenek, K. Ashcraft, J. Yang, and D. Engler, "Correlation Exploitation in Error Ranking," in *Symposium on Foundations of Software Engineering*, 2004, pp. 83–93.

[12] S. Kim, K. Pan, and E. E. J. Whitehead, Jr., "Memories of Bug Fixes," in *International Symposium on Foundations of Software Engineering*, 2006, pp. 35–45.

[13] Z. Li, L. Tan, X. Wang, S. Lu, Y. Zhou, and C. Zhai, "Have Things Changed Now? An Empirical Study of Bug Characteristics in Modern Open Source Software," in *Proceedings of the 1st workshop on Architectural and system support for improving software dependability*, 2006, pp. 25–33.

[14] L. Renggli, S. Ducasse, T. Grba, and O. Nierstrasz, "Domain-Specific Program Checking," in *Objects, Models, Components, Patterns*, 2010, pp. 213–232.

[15] T. J. Ostrand, E. J. Weyuker, and R. M. Bell, "Where the Bugs Are," in *ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2004, pp. 86–96.

[16] A. Mockus and L. G. Votta, "Identifying Reasons for Software Changes using Historic Databases," in *International Conference on Software Maintenance*, 2000, pp. 120–130.

[17] J. Śliwerski, T. Zimmermann, and A. Zeller, "When do Changes Induce Fixes?" in *International Workshop on Mining Software Repositories*, 2005, pp. 1–5.

[18] T. Zimmermann, S. Kim, A. Zeller, and E. J. Whitehead, Jr., "Mining Version Archives for Co-changed Lines," in *International Workshop on Mining Software Repositories*, 2006, pp. 72–75.

[19] S. Kim, T. Zimmermann, K. Pan, and E. J. J. Whitehead, "Automatic Identification of Bug-Introducing Changes," in *International Conference on Automated Software Engineering*, 2006, pp. 81–90.

[20] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in Software Engineering: An Introduction*. Kluwer Academic Publishers, 2000.

[21] S. Ducasse, A. Lienhard, and L. Renggli, "Seaside: A Flexible Environment for Building Dynamic Web Applications," *IEEE Software*, vol. 24, pp. 56–63, 2007.

[22] S. Wagner, F. Deissenboeck, M. Aichner, J. Wimmer, and M. Schwalb, "An Evaluation of Two Bug Pattern Tools for Java," in *International Conference on Software Testing, Verification, and Validation*, 2008, pp. 248–257.

[23] S. S. Joao Araujo Filho and M. T. Valente, "Study on the Relevance of the Warnings Reported by Java Bug-Finding Tools," *Software, IET*, vol. 5, no. 4, pp. 366–374, 2011.