Blending and Reusing Rules for Architectural Degradation Prevention

Alessandro Gurgel, Isela Macia, Alessandro Garcia, Arndt von Staa
OPUS Group, Informatics Department
Pontificial Catholic University of Rio de Janeiro, RJ, Brazil
{agurgel, ibertran, afgarcia, arndt}@inf.puc-rio.br

Mira Mezini, Michael Eichberg, Ralf Mitschke
Technische Universität Darmstadt
Darmstadt, Germany
{mezini,eichberg, mitschke}@informatik.tu-darmstadt.de

Abstract

As software systems are maintained, their architecture often degrades through the processes of architectural drift and erosion. These processes are often intertwined and the same modules in the code become the locus of both drift and erosion symptoms. Thus, architects should elaborate architecture rules for detecting occurrences of both degradation symptoms. While the specification of such rules is time-consuming, they are similar across software projects adhering to similar architecture decompositions. Unfortunately, existing anti-degradation techniques are limited as they focus only on detecting either drift or erosion symptoms. They also do not support the reuse of recurring anti-degradation rules. In this context, the contribution of this paper is twofold. First, it presents TamDera, a domain-specific language for: (i) specifying rule-based strategies to detect both erosion and drift symptoms, and (ii) promoting the hierarchical and compositional reuse of design rules across multiple projects. The language was designed with usual concepts from programming languages in mind such as, inheritance and modularization. Second, we evaluated to what extent developers would benefit from the definition and reuse of hybrid rules. Our study involved 21 versions pertaining to 5 software projects, and more than 600 rules. On average 45% of classes that had drift symptoms in first versions presented inter-related erosion problems in latter versions or vice-versa. Also, up to 72% of all the TamDera rules in a project are from a pre-defined library of reusable rules. They were responsible for detecting on average of 73% of the inter-related degradation symptoms across the projects.

Categories and Subject Descriptors D.2.11 [Software Architectures]: languages

Keywords architectural degradation; design rules, reuse

1. INTRODUCTION

Architectural degradation is a long-standing problem in software engineering. The architecture of software systems is well-known to increasingly degrade through the maintenance and evolution of the source code [7][14][16][28][37]. Hochstein and Lindvall [14] introduced the term architectural degradation to refer to this continuous quality decline. Symptoms of architectural degradation arise through the processes of architectural erosion and drift [14][28]. Erosion occurs when constraints governing the dependencies between architecture components are violated [28]. Drift symptoms imply the violation of intra-component constraints [28].

In spite of their differences, the erosion and drift processes are often intertwined [14][17]. Violations of component constraints may foster the later introduction of interaction violations or vice-versa, thus, the same or related modules in a program become the locus of both drift and erosion symptoms. The detection of a specific drift symptom may help to reveal erosion symptoms or vice-versa. If any of them remains undetected, it may provoke the emergence of the other one over the system's history [14][18][28]. Hence, the longevity of software projects largely depends on the early detection and repair of architectural degradation problems. Several techniques have been devoted to support architectural degradation detection [1][7][24][27][35][37]. However, there are two main problems with the current state of the art.

First, existing approaches promote the exclusive detection of either erosion or drift symptoms. The detection of either erosion or drift symptoms may not prevent the increasing decay of the software architecture [14][28]. For instance, the removal of the former symptoms may not imply the amelioration of the latter. On the contrary, the strict focus on erosion detection may imply that architects perceive severe drift symptoms too late, when it is hard or costly to address them. The converse is also true. Given the simultaneous occurrence of erosion and drift symptoms [14][18], architects should be able to elaborate hybrid strategies for detecting both forms of degradation using a unified set of abstractions.

Second, to the best of our knowledge, existing approaches only support the specification and checking of rules for particular systems and do not provide any mechanism to reuse them. As a consequence, the specification of such architectural rules becomes a repetitive task, as rules are often similar across multiple projects from the same domain or the same company. In addition, there is recent evidence, gathered from industry [41], presenting convincing results for reusing of anti-drift rules in 7 systems from a same domain. Ideally, architects should be able to reuse anti-drift and anti-erosion rules across projects adhering to similar architecture decompositions. This paper addresses these two problems by proposing TamDera 1, a new Domain-Specific Language (DSL), for specifying rules to detect architectural degradation (Section 4).
The proposed language has two distinguishing features: (i) support for specifying and blending rules for architectural erosion and drift detection in a unified way, and (ii) support for hierarchical and compositional reuse of such rules. It applies usual concepts of programming languages such as inheritance to enable comprehensibility, reusability and maintenance of architectural specifications. For instance, TamDera allows architects to define abstract anti-degradation rules that, by means of inheritance, are later specialized in concrete projects. The language was designed to support the detection of architectural degradation symptoms in the source code. TamDera allows to define architectural abstractions and mapped them to modules in the source code. The detection relies on static analysis to extract structural dependencies of implementation modules. This facilitates the detection of architectural degradation symptoms during the software build (Section 4.4). A tool implementation to support the language usage and rule enforcement is presented in this paper (Section 4.4).

As a second contribution of this paper, we evaluate the usefulness of supporting blending and reuse of architectural rules with TamDera. The study involves 21 releases of 5 projects, and more than 600 anti-degradation rules (Section 5). The findings revealed that developers could benefit from the TamDera approach. First, our evaluation provided evidence, observed in multiple projects, on the usefulness of using TamDera to detect the co-occurrence of erosion and drift symptoms. The results revealed that the simultaneous occurrence of drift and erosion symptoms – as detected by hybrid TamDera rules – in the same code module along the project histories was frequent. On average, 45% of the classes that had drift symptoms in early versions presented erosion problems in later versions and vice-versa. TamDera facilitated the early detection of these symptoms by providing a unified set of abstractions for architectural erosion and drift detection. Second, our analysis pointed out several cases where the exclusive detection of architectural degradation symptoms during the software build (Section 4.4). As a result, dependencies between code modules realizing the Controller and Model components are accidentally introduced, thereby leading to dependency violations.

2. Background

This section introduces terminology associated with architectural degradation. It also illustrates how symptoms of architectural erosion and drift co-occur through concrete scenarios. These example scenarios will subsequently be used throughout the paper.

2.1 Architectural Degradation Symptoms

Software architecture is concerned with the selection of architecture components and their dependencies as well as with constraints on both of them [28]. Components are architectural entities that encapsulate a subset of the system's functionalities [34]. A component of the architecture description is realized by one or more modules in the implementation. The term *module* is used to represent code elements, such as a package, an (implementation-) interface or a class, which contribute to the implementation of a coherent unit of functionality. In certain cases, inner elements of a module can even contribute to the implementation of different architectural components [34]. *Inner module elements* refer either to methods or fields of a class, or method declarations of an implementation-level interface.

**Erosion** occurs whenever an implementation decision violates one or more component dependency constraints in the specified architecture. In other words, an erosion symptom is the violation of a constraint at the level of a component dependency. A simple example is an unintended dependency established between two components. **Drift** occurs whenever an implementation decision leads to violations of component constraints in the specified architecture. Typical examples of component constraints are related to modularity principles, such as "narrow component interface" or "single responsibility principle" [28].

The processes of architectural erosion and drift, as well as their relationship, are illustrated below using a motivating example. From hereon, architectural degradation, architectural drift and architectural erosion are also referred to simply as degradation, drift and erosion.

**Example of erosion.** Figure 1 exemplifies an erosion symptom in the MobileMedia system [8]. This system realizes the architecture pattern Model-View-Controller (MVC)[4]. Each component is realized as a set of modules (classes) in the source code (only a few of them are shown in the figure). The erosion problem concerns an architecture constraint governing the exception handling policy: exceptions are incorrectly propagated through module interfaces across system components. The class BaseController defined in the Controller component e.g., invokes the Data service provided by AlbumData and ends up handling exceptions (e.g. PersistenceException) thrown by AlbumData, which should have been handled within the Model component. As a result, dependencies between code modules realizing the Controller and Model components are accidentally introduced, thereby leading to dependency violations.

![Figure 1. Erosion and drift symptom in MobileMedia (white) and HealthWatcher (gray) architectures](image-url)
Example of drift. Let us consider again the class BaseController in Figure 1. It exhibits the Large Class anomaly, as it defines many methods and realizes various non-cohesive functionalities of the system (e.g. video deletion and photo sorting). It also exhibits the Ambiguous Interface symptom [11], as it provides an over-generalized component interface - handleCmd - for handling all commands. This means that the BaseController module (and, therefore, the Controller component) is aggregating several responsibilities from different service requests that should be implemented independently. To overcome the drift symptom in a later version, the BaseController class was decomposed into smaller classes; each of them being in charge of realizing specific controller responsibilities. Thus, the single interface of the Controller component was decomposed in multiple specific interfaces. These examples also illustrate how architecture degradation symptoms could be introduced in anomalous code elements.

Drift symptoms are more difficult to detect than erosion symptoms. The key reason is that code anomalies are not always indicators of drift symptoms [17]. They contribute to drift only when they lead to violations of component constraints in the architecture specification [17][18]. For instance, certain Large Classes and Long Methods, which are often detected by metrics such as LOC, CAM and CC [15], may occur in modules that are not related to any component constraints.

2.2 Characteristics of Erosion and Drift

Erosion and drift symptoms tend to be intertwined. Drift symptoms often foster the later introduction of erosion symptoms and vice-versa. Drift symptoms impair code comprehension, causing developers to unconsciously introduce dependency violations in their programs [28]. The left-hand side of Figure 1 depicts an example of the close relationship between erosion and drift symptoms in MobileMedia. The class BaseController implements different services through its over-generalized interface handleCmd. The amount of services exposed by this interface significantly increased throughout the system evolution. This interface bloat in turn forced BaseController to handle exceptions propagated from several non-related components. The handling of such exceptions should not be a responsibility of the Controller component according to the MVC decomposition, thereby characterizing the occurrence of several dependency violations.

This example illustrates a direct relation between erosion and drift phenomena. Hence, in order to detect the degradation, architects should consider blending the detection of erosion and drift symptoms into the same architecture specification. In this way, developers would be aware of both degradation processes by just considering one architecture document. Therefore, this simultaneous detection would prevent developers from introducing erosion symptoms due to reminiscent drift symptoms and vice-versa. Furthermore, intertwined occurrences of drift and erosion symptoms are also a sign of severe stages of degradation [17][18].

Recent studies revealed that code modules exhibiting both kinds of degradation symptoms tend to be the locus of more severe architecture instabilities in a project history [18][20].

Similar degradation symptoms can infect several projects. It has been noticed that similar degradation symptoms can infect several projects. This occurs, for instance, when multiple projects in the same company share similar constraints either to particular components (e.g. frameworks or libraries) or to dependency among them (i.e. architectural style rules). Projects from different companies can also share similar architecture constraints. This occurs when they follow similar architecture designs. To illustrate this phenomenon, the right part of Figure 1 depicts the architecture of the HealthWatcher system, which follows the Layer style [12]. Certain modules of HealthWatcher suffer from similar erosion and drift symptoms as those occurring in modules of MobileMedia. For instance, SearchComplData introduces dependency violations through expectation propagation behavior, similar to BaseController in MobileMedia (Section 2.1). Also, it refines drift symptoms related to the implementation of independent responsibilities, such as GUI and Persistence. It contains large methods with duplicated code. In this way, SearchComplData has a similar code structure to BaseController. Unlike BaseController, it does not suffer from interface bloat. Furthermore, these classes implement components with different responsibilities (i.e. Controller and GUI).

The re-occurrence of the same degradation symptoms in Figure 1 illustrates the need for reusing architectural constraints, instead of defining them from scratch. Architects would benefit from a single reusable abstraction that groups constraints for detecting erosion (anti-erosion) and drift (anti-drift). For instance, an architectural specification can be established to group correlated anti-erosion and drift rules that constrain MobileMedia component elements. This specification encompasses anti-erosion constraints for the exception handling policy and tight coupling between non-related components. Also, the same abstraction groups anti-drift constraints for detecting drift symptoms related to interface bloat, such as large methods in BaseController. Therefore, the same abstraction can be reused in HealthWatcher to constrain the dependencies between non-adjacent layers, the exception handling behavior and size boundaries for GUI components such as SearchComplData. As we can notice, the reuse of hybrid specification of anti-erosion and anti-drift rules can help developers to reduce the effort of specifying such rules in each project. These specifications are also referred to in this paper as hybrid rules.

In addition to the reuse, architects would also benefit from specification mechanisms to specialize previously-defined anti-erosion and anti-drift constraints and associate them with the same reusable abstraction. These mechanisms would allow architects to subtly adjust specific constraints to each system context. As an example, anti-drift rules that are based on size boundaries, such as those applicable to BaseController, should be adjusted to other systems (e.g. SearchComplData in HealthWatcher).

3. Related Work

This section outlines previous work aimed at supporting the detection of architectural degeneration symptoms. We refer to the motivating example (Section 2.2) to illustrate limitations of anti-erosion and anti-drift techniques.

Anti-erosion techniques. The existing anti-erosion techniques provide mechanisms to: (i) explicitly define the intended architecture of a system, including the description of component dependency rules, and (ii) check if the system’s implementation is in conformance to the intended design. Sangal et al. [29] and Terra et al. [35] proposed DCL, a tool that represents relationships between code elements for a single project and detects violations of such dependencies. Eichberg et al. [7] presented a tool that allows the definition and checking of anti-erosion rules. They govern the relationships between the architectural elements of a software system. Marwan and Aldrich [24] developed an embedded language for documenting the system’s architecture in the source code and checking its conformance with a prescribed architecture. Morgan et al. [27] defined a domain-specific language to specify and check anti-erosion rules in the system implementation. Ubayashi et al. [37] presented a programming-level interface to represent the intended architectural design and detect

However, all these techniques are limited to only support the detection of erosion symptoms. Software projects supported by these tools are susceptible to the progressive introduction of drift symptoms, which can in turn foster the later introduction of more severe degradation symptoms (Section 2.2). Then, when a dependency violation is hopefully detected with these techniques, it is probably too late, for instance, to conduct architecturally relevant refactorings. In the best-case scenario, developers would have to resort to the use of independent techniques for preventing drift.

Furthermore, the techniques presented above rely on anti-erosion rules that are strictly specified for a single system and cannot be reused in other system contexts. For instance, architects cannot maintain a uniform base of anti-erosion rules to constrain the handling of specific exceptions to particular components. This solution would be useful as GUI classes cannot handle Data exceptions in HealthWatcher and neither Controller ones are allowed to handle Model exceptions in MobileMedia (Figure 1).

**Anti-drift techniques.** Other techniques, on the other hand, are limited to only identify symptoms of drift in system implementations. Consequently, developers can introduce unacceptable dependencies between components (Section 2.2). These techniques often refer to the evaluation of structural module properties that may be related to drift symptoms. The problem is that – since they only exploit source code information to detect anomalous code elements [17] – they do not offer means to enable developers to accurately identify which anomalous modules are architecturally relevant. Another limitation is the fact that these properties are strictly specified for a particular system. Therefore, they cannot be reused in other systems even though few adjustments are usually required to detect similar architectural anomalies in different contexts [26] (Section 2.2).

Marinescu et al. [22] presented a tool that relies on detection strategies [21] (i.e. combinations of code metrics) to identify anomalous code elements; they may be responsible for introducing drift symptoms. Moha et al. [26] presented a methodology to detect anomalous code structures by combining metric-based evaluations to structural module properties. Mara et al. [20] proposed a tool that enables the definition and application of both conventional and history-sensitive detection strategies. However, these techniques are not able to accurately identify which anomalous code elements are related to component dependency problems [17], although the first ones are often the source of latter ones [17][18]. This limitation is due to the fact that they only exploit source code information to detect anomalous code elements [17]; they do not offer means to enable developers to identify which anomalous modules are architecturally relevant.

**Taming drift and erosion symptoms.** To the best of our knowledge, there is no technique that supports detection of both erosion and drift symptoms. This is harmful to architectural maintenance as it is likely to be often the cause-effect relationships governing these symptoms (Section 2.2). State-of-art work often force developers to learn, at least, two different techniques for dealing with each kind of degradation process. As a consequence, rules for detecting drift and erosion symptoms are specified and analyzed independently, even though they rely on the same architectural abstractions. This creates an additional dependency between the two independent architectural specifications (i.e. anti-erosion and anti-drift) and the enclosed architectural element. As a result, whenever the architectural element changes, both specifications require modifications.

### 4. The TamDera Language

The proposed language, named **TamDera** (Section 4.1), enables the detection of both symptoms of degradation in the source code. Developers can define and blend anti-erosion and anti-drift rules to produce hybrid, composed rules (Section 4.2). **TamDera** also supports the reuse of these rules in multiple contexts (Section 4.3). **TamDera** is a complementary approach to conventional Architecture Description Languages (ADLs) [37] as it focuses on detecting architecture degradation symptoms in the source via the specification of anti-erosion and anti-drift rules. Therefore, as opposed to ADLs, the rules need to refer to **sets of source code elements** that realize architectural concepts. These concepts could be first described, for instance, in ADL descriptions.

In the following, we introduce a partial description of the **GUI** and **Data** components (Figure 1). It will be used to illustrate the basic abstractions and mechanisms of **TamDera**. We focus on anti-erosion rules (AER) and an anti-drift rule (ADR) obtained from the GUI and Data design descriptions:

"The GUI component's purpose is limited to handle user input and display data information to users. It delegates user requests to the Business component and displays the retrieved data information. In order to avoid this component from addressing other responsibilities, GUI classes are not allowed to directly access services provided by the Data component."

"The Data component provides interfaces for data manipulation and transaction management. It also realizes the handling of exceptional events raised by persistence and transaction management actions. These exceptions are remapped and signalled as general data-related exceptions to the upper layers."

Based on given descriptions we can infer the following architecture rules. First, an anti-erosion rule AER1 establishes that GUI classes cannot directly access services from the Data component. Second, an anti-drift rule ADR1 should impose upper modularity-related boundaries, for instance on the size and cyclomatic complexity of GUI classes; the goal is to avoid the accidental complexity coming from the implementation of several different responsibilities. For illustration purpose, we use size and cyclomatic complexity metrics, but others can also be used such as cohesion. The rules AER1 and ADR1 are combined to prevent the GUI component from realizing more responsibilities and introducing more dependencies than expected. The anti-erosion rule AER2 establishes that only Data classes are able to handle persistence and transaction exceptions.

### 4.1 Concepts and Mappings

Architects usually have a wide range of concerns that are represented by elements in architectural specifications, such as component models or ADL descriptions. Therefore, at the architectural level, each concern is realized by one or more elements of architectural specifications, such as components and interfaces. However, those architecturally-relevant concerns are often realized by multiple module elements in the source code. Therefore, it is not trivial to associate anti-degradation rules with the group of counterpart elements realizing architectural concerns in the source code. In order to support this activity, **TamDera** relies on two main constructs: **architectural concept** and **concept mapping**. They are described below.

**Definition 1 (Architectural concept).** An architectural concept is a relevant concern in the mind-set of software architects. It
identifies the set of module elements that realize this concern and, therefore, share uniform dependency and component constraints.

Module elements realizing a concept can be classes, interfaces, methods, or fields of a program. An architectural concept consists of anti-degradation rules. These rules should be respected by aggregate sets of module elements comprising the concept. TamDera is designed to support the static checking of anti-degradation rules on these elements. TamDera is not intended to increase flexibility by supporting description of rules relying on dynamic information (e.g., executions of method X must call method Y). The focus is to support the continuous detection of statically-checked rules to be integrated with the system build (Section 4.4).

The keyword concept is used to define an architectural concept with a unique name. Thus, different component constraints, such as size and complexity boundaries, are applied to different modules aggregating various architectural concepts. For instance, architects can specify less-restrictive size boundaries to components that implement core functionalities and which are expected to have a considerable size.

Graphical user interface (GUI), data management and data exceptions (Section 4) are examples of architecturally-relevant concerns captured by elements in the architecture specification. For illustration, Listing 1 shows a TamDera concept for these three architectural concepts. These concepts are relevant to the AER1, AER2 and ADR1 rules mentioned in Section 4. GUIHW (lines 01-03) and DataHW (lines 05-07) concepts refer to elements that respectively realize the GUI and Data components, whereas the DataHWException (lines 09-11) concept denotes specific exception classes pertaining to the Data component (Figure 1).

Listing 1
01: concept GUIHW { 02:   parent: "Command" 03: } 04: 05: concept DataHW { 06:   name: "healthwatcher.data." 07: } 08: 09: concept DataHWException { 10:   name: ".*PersistException | TransacException" 11: }

Definition 2 (Concept mapping). A concept mapping is a property of an architectural concept that defines which module elements realize the architectural concept.

TamDera supports concept mapping through regular expressions that identify properties shared by module elements realizing the concept. Examples of these properties are common names (suffixes, prefixes, and package names), or a common parent (super class or interface) of code elements. The definition of common properties governing element names is made using the keyword name (Listing 1). The name-based mapping receives a string (i.e. a regular expression) as input and retrieves all source code elements whose name matches it. Listing 1 illustrates the use of the name-based mapping. The concept DataHW is associated with all the code elements included in the package healthwatcher.data and in its sub-packages. Parent-based mapping is denoted by the parent keyword; it takes the common parent name as a parameter and maps all classes that extend or implement the common parent to the concept. For example, the concept GUIHW in Listing 1 is mapped to all classes whose parent class is named Command. Parent-based mapping is particularly interesting for programs with well defined interfaces. For instance, design pat-

<table>
<thead>
<tr>
<th>Dependency</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A invoke B</td>
<td>A method of A calls a method of B</td>
</tr>
<tr>
<td>A create B</td>
<td>Some method of A creates an object instance of a class of B</td>
</tr>
<tr>
<td>A declare B</td>
<td>The type of a field variable of A is a class of B</td>
</tr>
<tr>
<td>A derive B</td>
<td>A class of A extends or implements a class of B</td>
</tr>
<tr>
<td>A handle B</td>
<td>A code element of A has a catch block that handles any class exception of B</td>
</tr>
<tr>
<td>A depend B</td>
<td>A code element of A has some kind of dependency on a code element of B</td>
</tr>
</tbody>
</table>

Table 1: Dependency types considered by TamDera
terms such as Abstract Factory [10] rely on interfaces to structure their solutions.

4.2 Blending Anti-Erosion and Anti-Drift Rules

TamDera allows architects to blend anti-erosion and anti-drift rules in the specification of architectural concepts. The mechanisms to describe both forms of rules are described below.

Definition 3 (Anti-erosion rule). An anti-erosion rule is a construct that establishes a dependency constraint among architectural concepts.

Anti-erosion rules establish both mandatory and unacceptable dependency constraints among concepts. As a consequence, they prevent the violation of these dependency constraints. Each anti-erosion rule has a source and a target architectural concept. They refer to concepts by their names. The source concept encompasses code elements that are the source of an established dependency. The latter are the concepts whose dependencies with the source concepts are constrained. Table 1 summarizes the dependency types currently supported, which can be extended (Section 4.4).

In this table, A indicates a set of source concepts and B refers to target ones. Some types are not applicable to certain module elements, e.g., declare is only applicable to classes.

TamDera provides three constructs to establish anti-erosion rules: cannot, only-can and must. The cannot construct establishes that source concept elements are prohibited to have a specific dependency type with any target concept element. Listing 2 illustrates the AER1 (line 01) from the HealthWatcher architecture (Figure 1). It uses the construct cannot and the relationship invoke to prohibit the access from GUIHW (source concept) code elements to services provided by DataHW (target concept) elements. The only-can construct (line 03) establishes that only code elements from the source concept can have a specific dependency type with code elements from the target concepts. For instance, AER2 verifies whether there is a module in the code, which is not realizing the Data layer but which is undesirably handling (i.e. catching) an exception comprised by the concept DataModelHWException. The last rule (lines 05) uses another construct to state other aspects of the GUI's design description; for conciseness we omit their explanation and refer to [33] for a full description. The three aforementioned constructs are similar to constructs supported by DCL [35].

Definition 4 (Anti-drift rule). An anti-drift rule defines a component constraint that is relevant to an architectural concept.

TamDera allows architects to define strategies [21] for preventing drift in the form of anti-drift rules nested in architectural concept bodies. They rely on constraints formulated with metrics and thresholds for capturing deviations from modularity principles in each component. Structural metrics represent the most popular strategy to detect drift symptoms [15][20][26][27]. TamDera currently supports several metrics quantifying size, complexity, cohesion and coupling attributes. The language can be extended to support new metrics and strategies (Section 4.4).
For illustration, Listing 3 shows rules aiming to detect drift symptoms in the HealthWatcher architecture. In particular, the concept `GUIHW` has anti-drift rules to constrain the size (LOC) and the cyclomatic complexity (CC) of its code elements (ADR1). These rules use thresholds (i.e. 100 and 5) to determine limit values. These specific metrics were selected for illustrative purpose and other metrics could be used for detecting similar or different drift symptoms presented in the GUI modules of the HealthWatcher system (Section 2.2). The violation of drift rules may imply that developers need to reason about producing and checking anti-erosion rules (Section 2.2).

**Listing 3**

```java
01: concept GUIHW{
02:  parent: "Command"
03:  LOC < 100
04:  CC < 5
05: }
```

### 4.3 Reusing Anti-Degradation Rules

**TamDera** offers a compositional reuse mechanism for anti-drift rules as well as a hierarchical mechanism for architectural concepts, which in turn enables the hierarchical reuse of anti-drift rules. In addition, it supports the modular specification of rules and concepts in specification files that can be reused across multiple projects. This enables single-project and multi-project reuse of anti-drift rules.

Compositional reuse enables grouping anti-drift rules into a named set (anti-drift constraint set).

**Definition 5 (Anti-drift constraint set).** A constraint set establishes a set of anti-drift rules. The definition of a concept can refer to a constraint set in order to reuse its anti-drift rules.

Architects use the keyword `constraintset` to specify a set of reusable anti-drift rules in **TamDera**. In general the set of anti-drift rules of an architectural concept includes: (i) the rules explicitly defined within its body (Section 4.2) and (ii) those associated with the referenced constraint sets. Listing 4 illustrates the definition and reuse of a constraint set by the example of the `constraintset InheritanceOveruse` (lines 01-03), which constrains the depth of hierarchy class trees via the metric DIT (depth of inheritance tree) [15]. The constraint avoids that a piece of coherent functionality gets artificially decomposed into several hierarchy classes. The constraint set is reused in the GUI and Controller concept definitions (lines 05 and 09 respectively). For conciseness, we omitted the GUI and Controller anti-drift rules, which are composed with the `InheritanceOveruse` rules.

**Listing 4**

```java
01: constraintset InheritanceOveruse {
02:  DIT < 5
03: }
04: concept GUI{
05:  InheritanceOveruse
06: }
07: }
08: concept Controller{
09:  InheritanceOveruse
10: }
```

**TamDera** supports the reuse of previously defined concepts by means of an inheritance mechanism. The rationale behind hierarchical reuse is that concepts that play similar architectural roles in different projects should be subject to similar anti-degradation rules, i.e., they should belong to the same concept hierarchy tree. This is also the case when, within a single project, architectural concepts share similar structural constraints with subtle differences, such as threshold adjustments.

A concept inheritance establishes a hierarchy relationship between a super-concept and a sub-concept. The concept inheritance mechanism in **TamDera** promotes the reuse of anti-drift rules from the super-concept to the sub-concept. Figure 2 presents the definition of a super concept `GUI` (on the top of the figure). It defines three anti-drift rules (R1, R2, R3). They are in charge of realizing the rule ADR (Section 4). This concept is extended by `ViewMM` (on the left of the figure - lines 02-07), which implicitly inherits all `GUI` rules through the inheritance mechanism. Therefore, all module elements mapped to `ViewMM` must satisfy these rules.

**TamDera** concepts can be abstract or concrete. Unlike concrete concepts, abstract concepts do not specify a concept mapping (Section 4.1). For instance, the abstract concept `GUI` (Figure 2) has two concrete sub-concepts (`ViewMM` - left and `GUIHW` - right) that define a concept mapping. For evident reasons, only concrete concepts are checked during the anti-degradation rule conformance (Section 4.4).

**TamDera** allows users to modularize the specification of concepts and anti-degradation rules in several architectural models.

**Definition 6 (Architectural model).** An architectural model is an abstraction that defines a specification file modularizing the definition of architectural concepts and anti-degradation rules.

Figure 2 presents three such models: `abstract_rules` (top), HealthWatcher (right) and MobileMedia (left). The `abstract_rules` module defines an abstract concept `GUI` (lines 01-05) and an anti-erosion rule that checks the conformance of AER1. The other models specify rules to constrain the architecture of HealthWatcher and MobileMedia.

![Figure 2: Reuse of anti-degradation rules using TamDera](image)

The same architectural model (i.e., specification file) can be used in several projects, thus promoting the reuse of both anti-erosion and drift rules across multiple projects (Section 2.2). For instance, the concept `GUI` from `abstract_rules` and the anti-erosion rule `R4` are inherited by both `healthwatcher` and `mobilemedia` models through the `import` construct, which allows the inheritance of concepts and all anti-erosion rules defined in a super architectural model (i.e. imported) to a base one. The base architectural model can define sub-concepts of the inherited concepts from the super architectural model. For example, the concepts `GUIHW` and `ViewMM` reuse `GUI` and its reusable anti-drift rules. This allows architects to specify a library of reusable concepts and promote their reuse in multiple projects.
Extending anti-degradation rules. TamDera also enables to override and extend anti-drift rules in sub-concept definitions. Thus, architects can adjust thresholds reused from super concepts according to their needs. This is particularly interesting to enable developers in better coping with particular characteristics of a system. The rationale behind this mechanism is to provide the flexibility to reuse or not (i.e. override) anti-drift rules from parent concepts without necessarily modifying their definitions. Sub-concepts extend a reused anti-drift rule via the declaration of an anti-drift rule using the same metric and the same operator, but specifying different thresholds.

As an example, Figure 2 presents the definition of the concept GUIHW (right side), which has the GUI as super-concept. GUIHW overrides anti-drift rules from GUI imposing more restrictive boundaries for the lines of code (line 04 - R1+) and cyclomatic complexity (line 05 - R2+) of its code elements. These rules were overridden to capture existing drift symptoms that occur in GUI classes that have less than 200 lines of code.

In addition to overriding inherited rules, a base module can also define new anti-erosion rules that are only applicable to its sub-concepts. For instance, GUIHW establishes a new anti-erosion rule (line 9), requiring that GUIHW elements extend elements denoted by the concept AbsCond.

4.4 Implementation Issues

We have implemented the TamDera approach to check the conformance of anti-degradation rules in software systems. In a nutshell, all anti-degradation rules are translated to Prolog queries that use static source code information stored in the knowledge base to check their conformance. In particular, we decided to focus on Java systems to evaluate the feasibility of our approach. This allows us to reuse static analysis platforms for Java programs. The tool was implemented upon the Eclipse platform using XText [38], which is a framework for developing parsers and editors. XText provides features, such as generating editors with syntax coloring and code completion. This facilitates the specification of concepts and their anti-degradation rules.

Design overview. The TamDera tool uses Prolog [32] to statically check the conformance of anti-degradation rules. Prolog has been successfully used as a scalable engine to check structural dependencies related to erosion symptoms [7]. There is recent evidence [7] supporting the use of Prolog in the incremental build process of the system. As a consequence, the system can be continuously checked when the source code is modified [7].

Structural properties of module elements, their dependencies and metrics' values are directly represented as Prolog terms (i.e. knowledge base [5]). Thus, a single representation is used for detecting erosion or drift symptoms. This can also foster the tool integration with other programming languages as we can develop translators to represent architectural relevant information as Prolog facts. This characteristic is particularly interesting as recent studies have shown that most software projects nowadays are implemented in four different programming languages [37].

The tool reuses the Bytecode Analysis Toolkit (BAT) [7],[39], which receives the system binaries as parameter and retrieves a Prolog-based representation of the system. On the other hand, we use the metric files generated by Together [36] to check the anti-drift rules. It contains metric results based on structural properties (e.g. size and coupling) for each module element in the system under analysis.

Architectural conformance. The tool receives the system specification file whose architecture implementation is being checked as a parameter. It parses the (main) architectural model and also the ones which are referred to through the import key-word (Section 4.3). In this sense, it evaluates all anti-degradation rules taking into consideration the reuse mechanisms (Section 4.3). Also, it evaluates the concept mappings and stores them in the knowledge base. Our engine uses this information to identify concept code elements that violate any anti-degradation rule. TamDera generates an output report which describes each source code element that broke the corresponding anti-degradation rule. Consistency among rules. The tool also checks the consistency of anti-degradation rules in architectural models (Section 4.3). In fact, current techniques allow users to unconsciously define an inconsistent set of rules which impair the architectural conformance task [35]. For instance, users can define the rules: A must-involve B and A cannot-involve B. They impose contradictory constraints to the implemented architecture. Similarly to the checking of anti-degradation rules, the tool stores the information about each rule in the knowledge base and checks the consistency among the rules. For instance, we define a function to check if there are two anti-erosion rules that refer to the same concepts (A and B) and dependency types (invoke) but one uses a must relationship while the other uses a cannot one.

5. Evaluation

The usefulness of the TamDera language is largely dependent on how frequent the detected symptoms of erosion and drift are inter-related in a program (Section 2.2). The relation of these symptoms can be revealed in two ways (Section 5.4.1). First, their detection rules are logically blended; i.e. associated with the same concept (Section 4.2) in the same architecture model. Second, a drift symptom detected by TamDera rules in a program version is perceived to provoke an erosion symptom in a later version. Our evaluation is also intended to assess the adequacy of TamDera to promote the reuse of anti-degradation rules. Therefore, we defined two research questions that drive our evaluation: (i) how significant is the number of inter-related erosion and drift symptoms detected with TamDera rules, and (ii) to what extent anti-erosion rules and anti-drift rules can be reused in one or more projects.

5.1 Target Systems

We selected software systems for which either the intended architecture specification or the original architectures are available. Otherwise, we were not able to investigate the veracity of the rules as well as the degradation symptoms being detected. We also looked for systems adhering to architecture decompositions sharing the same architectural styles and design patterns where opportunities for reusing rules could be explored. At the same time, those systems needed to be from different domains and designed by different developers. The goal was to check whether recurring rules could be actually reused even in extreme cases, where the dominant application domains and developers' backgrounds were different. We also intended to select systems that underwent severe degradation stages, but were continued and redesigned in follow-up projects.

Based on these criteria, we selected three systems: MobileMedia [8], HealthWatcher [12], and MIDAS [19]. However, we took into consideration five projects. The reason is that the original Java projects of the first two systems (Section 2.2) manifested major symptoms of degradation over time. Then, two new follow-up projects [13],[30] started. They consisted of significant architecture re-structuring of both MobileMedia and HealthWatcher systems. The systems were partially re-designed with aspeccal decompositions and re-implemented with AspectJ [3]. We considered both groups of projects in our evaluation to check if the original design rules from the first project could be reused in the second project. We considered cases where the rules
were fully reused (as in the original project) and those that were refined to satisfy the particularities of the second project.

Despite of being projects designed by distinct architects they share — in many cases (e.g. Section 2.2) — similar design decisions. HealthWatcher [12] is a web system used for registering complaints about health issues in public institutions. MobileMedia is a product line that manages different types of media on mobile devices. MIDAS is a lightweight middleware for distributed sensor applications [19]. These projects were previously used in studies of degradation and refactoring [17][18][19]. This was beneficial to our study as we were also able to access their degradation symptoms from previous independent reports. These documents were helpful to evaluate the adequacy of TamDera rules to detect the reported degradation symptoms.

5.2 Study Procedures

The study relied on the use of all TamDera’s mechanisms (Section 4) to specify the anti-degradation rules. The process was conducted in two major phases:

Phase 1: Identification of architectural concepts. We accessed the available documentation to support the identification of architecturally-relevant concepts in each project. This activity was driven by the use of high-level component models that were available for all the five projects [33]. There were also specific models for certain versions where the intended architecture decomposition was modified. The subject systems make use of several architectural styles, such as MVC, Layers and Aspects Design. They also implement several design patterns that are often used to realize architecture decompositions, such as Chain of Responsibility and Façade [10]. We also referred to the documentation of these patterns to guide the specification of architectural concepts. As these pattern elements often rely on abstract classes [10], we naturally mapped the architectural concepts to these classes. Finally, we performed a peer revision with the original architects of each system. The goal was to ensure that the list of concepts and mappings (Section 4.1) specified with TamDera were good enough to represent the key decisions of the intended architectures.

Phase 2: Iterative improvement of anti-degradation rule specifications. We also referred to the aforementioned component models in order to specify some of the dependency constraints (i.e. anti-erosion rules). The documentation of styles and patterns were carefully examined to specify the rules for each concept identified in Phase 1. For instance, the responsibilities and characteristics of style and pattern elements were used to specify the anti-drift rules. The system developers also validated and provided us with a list of suggestions to enhance rule definitions based on their architecture knowledge. All the concepts and their corresponding rules are available at the study website [33]. In a final step, we identified opportunities to make the list of concepts and rule specifications more generic. The goal was to possibly enable their reuse across multiple projects. However, this generalization process was performed without acquiring specific knowledge of the 5 projects.

5.3 Assessment Settings

Our study evaluated the occurrence of inter-related erosion and drift symptoms as well as the reuse of their corresponding rules across the 5 projects (Section 5.1). First, we analysed the significance of co-occurring erosion and drift symptoms detected with TamDera. This analysis was supported by comparing (Section 5.4.1): (i) the percentage of code elements containing both forms of degradation symptoms, with (ii) the number of code elements containing at least an erosion symptom or a drift symptom. This comparison was important to check to what extent the TamDera’s rules were useful to diagnose the simultaneous occurrence of drift and erosion symptoms. Second, the procedures to assess the reuse of TamDera rules are described below. Due to space constraints, more details about the study settings are available at [33]. As TamDera has the distinguishing feature of detection both erosion and drift symptoms, we were not able to directly compare the language with other DSLs that describe only anti-erosion rules or only anti-drift rules (Section 3). Our study does not explicitly evaluate the effort to specify rules. However, the reuse assessment evaluates the degree of rules reused in single and multiple projects, thereby identifying scenarios where specification effort can be reduced.

Reuse assessment. We assessed if the TamDera mechanisms promote significant degree of reuse across multiple projects. The reuse assessment relied on quantifying the anti-degradation rules that were reused and contrasts this number with those rules defined from scratch. A rule was considered to be reused if it was used as originally specified. The reuse measure was derived by calculating the percentage of rules that are reused out of the total of them (i.e. both reused and non-reused rules). For a single project, we took into consideration the rules within the project file and the reused rules from the abstract rules file (Section 4.3). As an example, consider the HealthWatcher specification in Figure 2, which reuses 2 rules from the super concept gui (R3 and R4), overrules two rules (R1+ and R2+) and defines a new anti-erosion rule (R5+). Hence, the total number of rules is 5, of which 2 are reused, resulting in a reuse degree of 40% (2 out of 5 rules).

Effective reuse assessment. Then, we identified the reused rules that actually detected architectural design problems. These rules are named effective reused rules. We counted the number of classes containing degradation symptom(s) to evaluate the effective reuse of rules. Then, we distinguished the symptoms that were detected by reused rules from those defined from scratch. Hence, the effective reuse was evaluated as the percentage of degradation symptoms identified by the reused rules out of the total of degradation symptoms (i.e. including those identified by non-reused rules). For illustration, consider the numbers (red values) on the right-hand side of the rules in Figure 2, which correspond to the number of erosion and drift symptoms detected by the corresponding rule. In the case of HealthWatcher, there are 2, 5, 6, 0, 1 degradation symptoms detected by R1, R2, R3, R4, R5 respectively. Hence, the effective reuse is 42.8% (6 out of 14 rules), as 6 degradation symptoms were detected by the reused rule R3. The multi-project effective reuse was evaluated in a similar way, however, by considering only rules that detect degradation symptoms in at least two different projects.

5.4 Study Results

All the results of our study are available at the complementary website [33]. Our evaluation was based on the architectural specification files for 8 versions of HealthWatcher, 7 versions for MobileMedia and 2 versions of MIDAS. Also, we considered the specification files of the first and forth Aspect versions of MobileMedia and HealthWatcher (Section 5.1). So, we analyzed 21 versions of TamDera specifications in total. The evolution history of all the systems underwent architecturally relevant changes. The MobileMedia evolution was guided through the addition of new features [8], whereas the HealthWatcher history mostly encompassed refactorings of specific modules in order to adopt architecturally relevant design patterns [12]. The MIDAS versions are those before and after restructurings to improve the system’s modularity and adaptability [19]. Therefore, the anti-degradation
rules of the target projects had suffered modifications over their evolution.

We chose to focus on the data of the versions 1, 4 and 8 of HealthWatcher, the versions 1, 4 and 7 of MobileMedia, the versions 1 and 4 of the aspectual implementation of both systems, and the two versions of MIDAS. We name these versions as HWv1, HWv4, HWv8, MMv4, MMv7, HA1, HA4, MA1, MIDASREF, and MIDASAPT, respectively. These versions are those that suffered from the most widely-scaled changes in both implementation and architecture levels along the system’s evolution. Therefore, they help us to better illustrate the study results. The analysis encompassed more than 600 anti-degradation rules and more than 300 concept specifications. Table 2 summarizes the amount of concepts, rule types, and the amount of classes in the code that actually manifested degradation symptoms in each analysed version. The number of anti-degradation (ADG) rules is the tally of anti-erosion (AE) and anti-drift (AD) rules.

Table 2. Characteristics of systems’ specification files

<table>
<thead>
<tr>
<th></th>
<th>MM</th>
<th>MA</th>
<th>HW</th>
<th>HA</th>
<th>MIDAS</th>
</tr>
</thead>
<tbody>
<tr>
<td>concepts</td>
<td>1</td>
<td>4</td>
<td>1</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>Anti-Erosion rules</td>
<td>22</td>
<td>24</td>
<td>28</td>
<td>30</td>
<td>24</td>
</tr>
<tr>
<td>Anti-Drift rules</td>
<td>25</td>
<td>25</td>
<td>29</td>
<td>33</td>
<td>34</td>
</tr>
<tr>
<td>Anti-Deg. rules</td>
<td>47</td>
<td>48</td>
<td>54</td>
<td>68</td>
<td>67</td>
</tr>
<tr>
<td>classes with Deg.</td>
<td>9</td>
<td>14</td>
<td>8</td>
<td>12</td>
<td>43</td>
</tr>
<tr>
<td>Deg.=degradation; A=After; B=Before</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

5.4.1 Co-occurring Erosion and Drift Symptoms

Simultaneous occurrences of erosion and drift symptoms. We evaluated the simultaneous occurrence of drift and erosion symptoms detected with TomDera in the same modules. Figure 3 shows the results. The percentage of the symptoms was computed based on the total of degradation symptoms for each version described in Table 2 (last row). The histogram presents the percentage of classes containing only erosion symptoms (ES), only drift symptoms (DS) and both of them (DGS). On average 45% of the HealthWatcher and MobileMedia classes, which exhibit degradation symptoms, contain both erosion and drift symptoms. MIDAS was an exception for the reasons discussed below. The high rate of drift and erosion co-occurrences is a first indicator of the usefulness of TomDera to support: (i) the retrospective analysis of cause-effect relationships involving drift and erosion symptoms affecting the same modules in the code, and (ii) the early detection and removal of drift (or erosion) symptoms in order to prevent the later occurrence of other erosion (or drift) symptoms caused by the former ones in later versions.

The symbiosis of erosion and drift detection. It could be that an extent of these co-occurring drift and erosion symptoms were just accidentally affecting the same module, but has no conceptual or historical relation. However, we observed that, on average, 85% of co-occurring symptoms were revealed by TomDera rules bound to the same architectural concept. These symptoms are referred to as concept-related. For instance, the rules AER1 and ADR1 (Section 4) rely on the GUI concept and detect both degradation symptoms that occur in SearchCompiData (Section 2.2). We also observed that, in many cases, a pair of drift and erosion symptoms was concept-related, but they were not necessarily occurring in the same modules. These are typically the case of rules for concepts related to patterns and styles (Section 5.4.2). For instance, each layer from the HealthWatcher architecture is described as an independent concept. These concepts have in common a super-concept, which establishes general anti-drift rules for coupling and size. They also involve anti-erosion rules that constrain the dependencies between non-adjacent layers. Despite these rules being concept-related, they detect erosion and drift rules that occur in different modules (i.e. modules that belong to different layers such as GUI and Data). It would be difficult and time-consuming to detect these concept-related symptoms through the use of individual techniques (Section 3) for drift and erosion.

Erosion detection alone does not prevent degradation. If we also take MIDAS into consideration, the simultaneous occurrence of erosion and drift symptoms decreases from 45% to 35%, which is still significant (Figure 3). MIDAS was developed using a middleware environment in charge of strictly enforcing the conformance of its implementation to the intended architecture [19]. It means that no dependency violation (i.e. erosion symptom) would remain in the code and, therefore, this system’s versions did not exhibit any erosion symptom (Figure 3).

However, the quality of MIDAS architecture had progressively declined until the point where a major restructuring was required [19]. The reason was that several components of MIDAS were progressively exhibiting drift symptoms: they increasingly lost their original conceptual coherence (i.e. purpose) as their implementations had evolved to provide multiple services. In other words, they were increasingly manifesting anomalies related to the “single responsibility” principle [23]. The MIDAS architecture significantly decayed due to the continued incidence of drifts [11]. Even though the developers were concerned with erosion prevention, the MIDAS architecture became susceptible to degradation through an architectural drift process. Hence, this scenario reinforces the importance of the early detection of both degradation symptoms provided by TomDera (Section 4). More importantly, we observed in the MIDAS case that the rules would be beneficial to diagnose the following fact: the enforcement of anti-erosion rules might be the actual cause of drift rule violations. This could be easily observed via TomDera specifications when both rules are bound to the same architectural concept.

Drift and erosion symptoms throughout systems’ evolution.

We also observed other interesting cases. Our analysis revealed that erosion symptoms in early versions often result in erosion problems in later versions and vice-versa. For instance, in HealthWatcher, several method declarations were signalling exceptions to other components, but those exceptions were supposed to be internally handled. These were cases of erosion symptoms. Those exception declarations were placed in methods in parent classes and those erosion problems in turn caused drift symptoms in children classes. The latter classes were forced to log the occurrence of these exceptions and throw them as defined in the parent class. This situation increased the internal complexity of children classes as well as their coupling degree with neighbouring components. Also, there were cases where drift
symptoms in early versions were the source of later violations in the project history. For instance, the number of responsibilities realized by the Controller component increased through successive versions. In later versions, this responsibility overload required the Controller to access information from different components, thereby establishing unintended dependencies. The analysis revealed that 66% of drift symptoms in Controller classes were sources of later dependency violations.

**Identifying and removing co-occurring symptoms is not trivial.** When both kinds of symptoms infect the same module, someone could expect that by removing one symptom, the other will be easily detected and fixed as well. For instance, when removing of unacceptable access of SearchComplData (erosion symptom) to Data services, someone could observe that the class is inadequately addressing other responsibilities, such as handling data-specific objects (drift symptom). This expectation motivated us to investigate how often erosion and drift symptoms that infected the same module were simultaneously fixed. However, this behavior was not observed in more than 61% of all the co-occurrences detected in the target systems. For instance, in the AspectJ project of HealthWatcher all the erosion symptoms in the GUI classes were addressed through the modularization of Persistence and Transaction exceptions. This refactoring reduced the number of responsibilities that GUI classes were undesirably dealing with. However, GUI classes remained infected by drift symptoms as they introduce a tight coupling degree between GUI and Business layers. This co-occurring problem could be detected by TamDera as the rules for detecting both problems would be defined in the same GUIHW concept.

There were also cases where the removal of drift problems did not imply the detection and fix of related erosion symptoms. For instance, around 83% of all the drift problems in the Controller classes of MobileMedia were addressed by decomposing them in micro controllers. Thereby, each specific controller was responsible for dealing with a specific functionality. However, after this architecturally-relevant refactoring, the erosion symptoms persisted in the code as Controllers continued to deal with exceptions propagated by the Data component (Section 2.2). These scenarios provide interesting evidence that: (i) the detection of an erosion problem does not imply that it is easy to identify a concept-related drift problem occurring in the same code module and vice-versa, and (ii) relying on techniques for detecting just one kind of degradation symptom (Section 3) are not enough to enable developers to prevent architectural degradation.

### 5.4.2 Significant reuse of rules

**Significant reuse of rules.** The second study goal was to analyze the potential reuse of TamDera rules in different contexts. We elaborated sources of reusable rules specifying architectural constraints (Section 5.2). These rules were reused in the target projects. Table 3 presents the amount of reused rules for each system. Similarly to Table 2, we distinguish the amount of anti-erosion, anti-drift and anti-degradation rules. An analysis of the last row of Table 3 reveals that 72% of the specified rules were reused on average, taking into consideration the total number of specified rules for the all systems. This finding suggests that architects can significantly save resources on the development and maintenance of architectural rules shared by several projects. Changes applied to shared rules are propagated through the reuse mechanisms of TamDera (Section 4.3) to multiple projects.

**Reuse of style and pattern constraints.** We observed that a large extent of the reused rules, specified in reusable concepts, was related to architectural styles and design patterns (Section 5.2). The definition of each single style or pattern is often formed by a cohesion set of component (anti-drift) and dependency (anti-erosion) constraints. For instance, the Controller classes of MobileMedia realize the design pattern Chain of Responsibility (CoR) [10]. This pattern reduces the coupling between the sender of a request to its receiver by delegating the request handling to multiple objects. Listing 5 presents part of a reusable rule associated with the CoR pattern. They define drift and erosion rules for code elements realizing the Handler concept. Those elements are architecturally relevant as they handle requests coming from other components. In Listing 5, the client interface as well as other concepts and rules of the CoR were removed from Listing 5 for conciseness. First, it defines an anti-drift rule constraining the coupling strength of each concrete class handling the request (ConcreteHandler, lines 01-04) through the metric CBO (coupling between objects) [15]. The coupling threshold is represented by the constant LOW_COUPLING. Second, there is also a reusable anti-erosion rule to prohibit direct calls of clients to concrete handlers (line 05). More specifically, those clients are realizing other architectural concepts and should access a specific interface to send their requests. This example shows how drift and erosion rules of a single pattern are mutually-related (Section 2.2): while the former ones enforce structural properties of modules realizing pattern concepts, the latter ones constrain their dependency with other architecturally-relevant concepts of the system.

<table>
<thead>
<tr>
<th>Table 3: Reuse of architecture rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>MM</td>
</tr>
<tr>
<td>---</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>Anti-Erosion rules</td>
</tr>
<tr>
<td>Anti-Drift rules</td>
</tr>
<tr>
<td>Anti-Deg. rules</td>
</tr>
<tr>
<td>Anti-Deg. rules (%)</td>
</tr>
</tbody>
</table>

B=Before; A=After; Deg.=Degradation

**Significant detection of degradation symptoms by reused rules.** We observed that a significant number of erosion and drift symptoms were detected by reused rules (72% of original rules) in all the 5 projects. They were responsible for detecting on average 73% of the existing symptoms. Table 4 illustrates the effective reuse (Section 5.3) for each system version. These measures represented a balance between the reused rule percentage and the symptoms detected by them. Therefore, the reused rules had similar efficiency to detect architectural deviations in comparison to the non-reused rules unique to each project.

**Listing 5**

```java
01: constraint set ConcreteHandler {
02:    thresholds: LOW_COUPLING
03:    CBO < LOW_COUPLING
04: }
05: Client cannot invokes ConcreteHandler
```

**Overriding anti-drift rules.** There was a need to subtly override reused rules in 11% of the cases [33]. For instance, the concept GUIHW overrides anti-drift rules from GUI to impose more restrictive constraint boundaries (Section 4.3). These boundaries are used to capture symptoms in particular HW GUI elements. In such scenarios, we decided not to modify these rules in the GUI super-concept. Otherwise, it would potentially generate false positives in the MM analysis. In this context, false positives are related to the use of detection strategies to identify drift symptoms. It represents modules that violate drift rules but do not have the associated drift symptoms. In fact, these restrictive boundaries are not applicable for the ViewMM code elements (Figure 2). Rule overriding (Section 4.3) was often useful to avoid false positives.
and false negatives, in addition to capture particular symptom intricacies of a project. It was also particularly interesting for addressing adjustments required in the aspect-oriented refactorings of the MM and HW architectures (Section 5.1). For instance, concept mappings need to be often overridden to consider: (i) the inclusion of new code elements, and (ii) the renaming or removal of certain classes. Thresholds of drift rules also needed to be replaced in specializations of architectural concepts.

### Table 4: Detection of degradation symptoms by reused rules

<table>
<thead>
<tr>
<th>Deg. Symptoms</th>
<th>MM</th>
<th>MA</th>
<th>HW</th>
<th>HA</th>
<th>MIDAS</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>B</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>A</td>
</tr>
</tbody>
</table>

**DGRR** = degradation symptoms detected by reused rules; B=Before; A=After; Deg.=Degradation

**Detection of the same degradation symptom in multiple projects.** The reused rules were effective in the detection of the same degradation symptom in multiple projects. We selected a representative set of pairs of system versions, which were sharing reusable rules. Figure 4 presents the results for each of those selected pairs (represented in the x-axis). For instance, the first pair is formed by the first versions of the HW and MM projects. The rules that detect degradation symptoms in both projects are called *common reused rules* in Figure 4. Their percentage (dark grey bar) is computed from the total number of rules defined for the pair of versions. We assessed the percentage of degradation symptoms detected by them, the so-called *similar symptoms*. The analysis reveals that 34% on average of the rules were effectively reused to detect degradation symptoms that occur in both HW and MM projects. Examples of these rules are: (i) anti-erosion rules constraining component dependencies imposed by architectural styles (e.g. R4 - Figure 2), (ii) anti-drift rules to constrain size, complexity and coupling of particular components/elements such as View and Model (e.g. R3 - Figure 2), and (iii) both anti-erosion and anti-drift rules associated with architecturally-relevant design patterns such as the Command [10].

### 5.5 Threats to Validity and Study Limitations

Threats to external validity are related to the choice of target applications and the generalization of results. We tried to mitigate this threat by selecting applications from different domains and developed by different teams of architects and developers. In fact, they had already been used in previous empirical studies [8][11][17][18][20] with similar purposes. The choice of applications also considered the criteria described in Section 5.1.

We identified two major issues that threaten the construction validity of our study: the specification of architectural concepts and their rules, as well as the identification of architectural degradation symptoms. To mitigate this treat, we specified the concepts referring to components explicitly present in high-level diagrams for all projects (Section 5.2). We also performed a detailed revision with the original systems’ architects to guarantee that the defined concepts capture the constraints associated with the intended architecture. We only considered those architecture concepts and constraints that were confirmed by systems’ architects.

We mitigated the second issue by using reports about architectural degradation and related refactorings in each application as an “oracle” to retrieve the degradation symptoms. These reports were produced in previous empirical studies based on such systems. They provided helpful information regarding the intended architecture design of target systems and the location of the architect-

![Figure 4: Effective reuse of common degradation symptoms](image-url)

The main issues that threaten the conclusion validity of our study are the representativeness of the characteristics and number of used systems. We recognize that we cannot extrapolate much the results of our study to a wide range of projects. Even though those systems share some similar architecture design decisions, they follow different architecture styles and are from different domains. We are aware that the reuse of anti-degradation rules may be affected because of considering systems adhering to similar styles and patterns. However, we followed a pragmatic approach, i.e. we believe that no reuse can be promoted if there is no effort upfront to anticipate general rules applicable to architectures following similar design decompositions.

As far as the number of analyzed programs is concerned, we relied on 21 versions of 5 different systems. Of course, a higher number of systems and versions is always desired. However, the analysis of a bigger sample in this first study would be impracticable for different reasons. First, the relationship between code modules and degradation symptoms need to be confirmed by architects. Second, systems with all the required information and stakeholders available to perform this study are rather scarce. Then, our sample can be seen as appropriate for an exploratory evaluation.

In terms of study limitation, the study assessment does not explicitly evaluate the effort need to specify anti-degradation rules. Effort is an attribute that can only be measured via controlled experiments which deviates from the assessment focus at this point (i.e., evaluates TamDera’s novel features). However, our study pointed out how the effort to specify rules can be potentially reduced through the reuse mechanism (Section 5.4.2).

A false positive is characterized when an anti-degradation rule is violated by a module that does not have any degradation symptom. On the other hand, a false negative is characterized when rules are not enough to pinpoint a degradation symptom. In respect to anti-drift, it is well known from literature that false positives and negatives largely depends on the ability of architects to select the metrics and define thresholds based on the particular characteristics of the program [18]. Using TamDera’s reuse mechanisms, architects can combine metrics and reuse or redefine thresholds for system adhering similar architectural decompositions. As far as anti-erosions are concerned, false negatives might occur if the language is not expressive enough to express certain form of constraints governing architectural components. At the moment, TamDera does not support the specification of rules related to dynamic information, but the language can be extended to incorporate new constructs to specify object instances and flows of method execution (e.g., similar to AspectJ’s cflow construct). In our evaluation, we observed an insignificant set of false positives and negatives. The system original developers reviewed and provided suggestions to iteratively enhance the rule specifications during the first steps of the study (Section 5.2).
6. Concluding Remarks

This paper presents TamDera, a language that supports blending of anti-drift and anti-erosion rules to detect both symptoms of architectural degradation. TamDera enables the compositional and hierarchical reuse of anti-degradation rules. We evaluated the usefulness of our language by detecting degradation symptoms in 21 versions of 5 different projects.

Our evaluation showed that TamDera could help architects and developers to save time and resources. We found that 72% of the anti-degradation rules were reused from existing ones. The analysis also revealed that such reused rules were responsible for identifying on average of 73% of all erosion and drift symptoms in the target projects (Section 5.4.1). More interestingly, there were a broad range of scenarios confirming that individual techniques for drift or erosion would not be sufficient or efficient to support degradation prevention. In cases where both symptoms were affecting the same module in the code, developers detected and removed only one symptom (Section 5.4.1). In addition, many inter-related drift and erosion symptoms do not necessarily affect the same modules in the code (Section 5.4.2), which make them difficult to detect together using existing techniques (Section 3).

Our study pointed out how the effort to specify rules can be potentially reduced thanks to: (i) the degree of reuse achieved once a generic rule is defined, and (ii) the elimination of the need for learning at least two different languages (for anti-degradation rules). In our agenda, we plan to conduct controlled experiments and case studies with our industry partners to evaluate the effort of using TamDera to describe anti-degradation rules.

References

[10] Gamma, E. et al. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley.