4
Evaluation

Existing approaches for preventing architectural degradation promotes the exclusive detection of either erosion or drift symptoms (Section 2.5). The design of **TamDera** is based on the assumption that by supporting the detection of both erosion and drift symptoms, developers may better prevent the increasing decay of software architectures (Chapter 3). The use of **TamDera** can help to avoid the strict focus on erosion detection, which often imply that architects perceive severe drift symptoms too late, when it is hard or costly to address them.

In this context, the usefulness of the **TamDera** language is largely dependent on how frequent; in fact, the same location in the program becomes the locus of inter-related erosion and drift symptoms. If these symptoms are somehow related, it is likely that architects will benefit from hybrid detection rules supported by **TamDera**. The relation of these symptoms can be revealed in two ways: (i) their **TamDera** anti-degradation rules are logically blended, i.e., associated with the same architectural concept in the architecture model, or (ii) a drift symptom encountered in a version is perceived to provoke an erosion symptom in a later version. The usefulness of **TamDera** is also dependent on its adequacy to promote reuse of anti-degradation rules.

Therefore, we defined two research questions that drove our evaluation and are addressed in this Chapter:

(i) how significant is the number of modules in evolving systems exhibiting inter-related erosion and drift symptoms?

(ii) to what extent can anti-erosion rules and anti-drift rules be reused by one or more projects?

The following subsections describe the target applications (Section 4.1), the study procedures (Section 4.2), the evaluation settings (Section 4.3), the results (Section 4.4), and threats to validity (Section 4.5). Section 4.6 summarizes the study results.

4.1. Target applications

We selected applications for which either the intended architecture specification or the original architects are available. Otherwise, we are not able to investigate the veracity of both kinds of anti-degradation rules and the architectural degradation symptoms being detected. We also looked for systems adhering to architecture decompositions that shared architectural styles and design patterns. The goal was to identify a subset of systems where opportunities of reuse of rules could be explored. At the same time, those systems need 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 domains and developers' backgrounds were different. We also selected systems that underwent severe degradation stages and were continued and redesigned in follow-up projects.

Based on these criteria, we first selected three systems: MobileMedia (FIGUEIREDO et al, 2008), HealthWatcher (GREENWOOD et al, 2007), and MIDAS (MALEK et al, 2007). However, we took into consideration five projects. The reason is that the original Java projects of the first two systems manifested major symptoms of architectural degradation over time (Section 1.2). Then, two new follow-up projects (SOARES ET AL, 2007; HEATHWATCHER, 2012) started and consisted of significant architecture re-structuring of both systems. The systems were partially re-designed with aspectual decompositions and re-implemented with AspectJ (ASPECTJ, 2012). We considered both groups of projects in our evaluation to check if unchanged or refined design rules could be reused. Despite of being projects designed by distinct architects, they share, in many cases (e.g., Section 1.2), similar design decisions. HealthWatcher 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 (GARCIA et al, 2009). These projects were previously used in studies of architectural degradation and refactoring (GARCIA et al 2009; DANTAS et al 2011; MACIA et al 2012). Therefore, we were able to access their degradation symptoms from previous reports and evaluate the adequacy of **TamDera** rules to detect them.

4.2. Study procedures

The study used **TamDera**'s abstractions and mechanisms, including the notions of concepts, concept mapping, concept inheritance, constraint sets, anti-erosion and anti-drift rules, and rule overriding (Section 3.1). The study was conducted in three major phases:

Phase 1: Identification of architectural concepts. We accessed the available documentation to support the identification of architecturally-relevant concepts in each project. High-level architecture models were available for all the five projects (TAMDERA, 2012). There were also specific models for certain versions where the intended architecture was modified. The subject systems make use of several architectural styles, such as MVC, Layers (Section 2.3) and Aspectual Design (SOARES et al, 2007). They also implement several design patterns that are often used to realize architecture decompositions, such as Chain of Responsibility and Façade (GAMMA et al, 1995). We also referred to these patterns to guide the specification of architectural concepts. As the pattern roles often rely on abstract classes (GAMMA et al, 1995), we naturally mapped the architectural concepts to these classes. Finally, we performed a peer revision with the original architects of each system to guarantee that the concepts 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 architecture models to specify some of the interaction 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 elements and design pattern roles 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 were made available at the study website (TAMDERA, 2012). In a final step, we generalized rule specifications, so that we could reuse them across the 5 projects.

Phase 3: Assessment. We analyzed the co-occurrences of both erosion and drift symptoms. Then, we investigated the relation of the occurrences of these processes throughout the later versions of each system. In addition, we assessed the reuse of hybrid rules for preventing architectural degradation in each project and also how these rules detect the same kind of degradation symptom across multiple projects. The goal was to assess if the **TamDera's** mechanisms promote a significant degree of rules' reuse and if these rules are efficient to detect the degradation symptoms.

4.3. Evaluation 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 4.1). First, in order to analyze the significance of co-occurring erosion and drift symptoms (Section 4.4.1), we compared : (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. The idea was to check the proportion of modules possibly manifesting inter-related drift and erosion symptoms, when compared to the total number of modules containing any kind of symptom. The procedures to assess the reuse of **TamDera** rules are next described. Finally, a full description of the study settings is available at (TAMDERA, 2012).

Reuse assessment. The reuse assessment relied on quantifying the anti-degradation rules that were reused, and contrasting this number with those rules defined from scratch. This reuse measure was calculated by 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 3.1.4.2). As an example, consider the HealthWatcher specification in Figure 8, which reuses 2 rules from the super concept `GUI` (R3 and R4), overrides 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 (highlighted values) on the right-hand side of the rules in Figure 8, 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.

4.4.
Study results

Our evaluation was based on the analysis of the architectural specification files produced for 8 versions of HealthWatcher, 7 versions for MobileMedia and 2 versions of MIDAS. We also considered the specification files of the first and fourth AspectJ versions of MobileMedia and HealthWatcher. So, we analyzed 21 versions of TamDera specifications in total. The files for two subsequent versions of the same system are different only when there were one or more changes to the architectural rules.

The evolution history of all systems underwent architecturally relevant changes. The MobileMedia evolution was guided through the addition of new features (FIGUEIREDO et al, 2008), whereas the HealthWatcher history mostly encompassed refactorings of specific modules in order to adopt architecturally relevant design patterns (GREENWOOD et al, 2007). The two versions of

MIDAS are those before and after restructurings to improve the system's modularity and adaptability (GARCIA et al, 2009). Therefore, the anti-degradation rules of the target projects had suffered modifications over their evolutions.

In order to present the results, we chose to focus on the data of: (i) versions 1, 4 and 8 of HealthWatcher, (ii) versions 1, 4 and 7 of MobileMedia, (iii) versions 1 and 4 of the aspectual implementation of both systems, and (iv) the two versions of MIDAS. These versions are those that suffered from the most widely-scoped changes in both implementation and architecture levels along the system's evolution. Therefore, they help us to better illustrate the study results. We name these versions as HW_1, HW_4, HW_8, MM_1, MM_4, MM_7, HA_1, HA_4, MA_1, MA_2, $MIDAS_{BEF}$ and $MIDAS_{AFT}$, respectively. The analysis encompassed more than 600 anti-degradation rules and more than 300 concept definitions. Table 2 summarizes the amount of concepts, rule types, and the amount of classes in the code that actually manifested degradation symptoms in each analyzed version. The number of anti-degradation (ADG) rules is the tally of anti-erosion (AE) and anti-drift (AD) rules.

Table 2. Properties of TamDera system specifications

	MM			MA		HW			HA		MIDAS	
	1	**4**	**7**	**1**	**4**	**1**	**4**	**8**	**1**	**4**	**AFT**	**BEF**
# of concepts	22	24	24	28	30	24	45	76	27	49	26	28
# of AE rules	22	23	32	25	33	34	50	66	36	52	20	20
# of AD rules	25	25	27	29	35	33	44	51	37	52	22	26
# of ADG rules	47	48	59	54	68	67	94	117	73	104	42	46
# of classes where occur DG	9	14	16	8	12	43	49	60	36	41	49	45

AE = anti-erosion; AD = anti-drift; ADG = anti-degradation; DG = degradation

4.4.1. Co-occurring erosion and drift symptoms

Simultaneous occurrences of erosion and drift symptoms. We evaluated the simultaneous occurrence of drift and erosion symptoms in the same modules. This provided a first evidence on the likelihood of those symptoms being somehow inter-related. Figure 11 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 further discussed in this section.

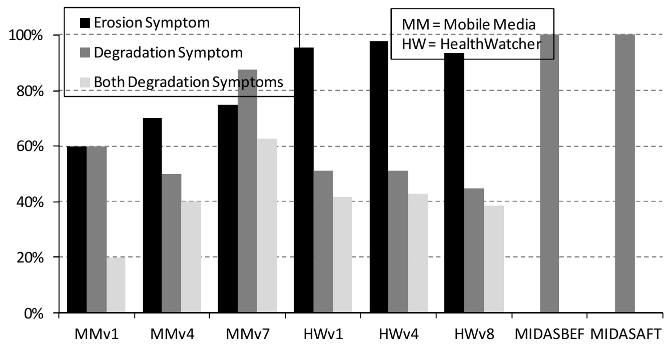


Figure 11. Analysis of co-occurring erosion and drift symptoms

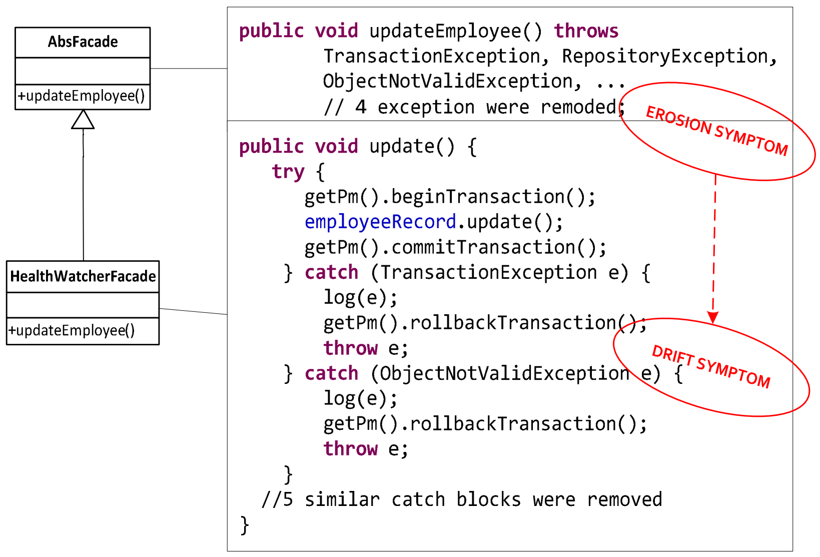
In Figure 11, we also observe that MM_1 presents a lower DGS measurement in comparison to the other ones. Particularly, only the class `BaseController` (Section 1.2) had both degradation symptoms. It implements the entire management of all media types. Throughout the MobileMedia evolution, new controllers were created and the management media types were shared by the controllers. As a consequence, the number of classes containing both symptoms increased along the system evolution as more controllers had both degradation symptoms. On the other hand, the HealthWatcher measures were practically the same across the versions. It encompassed several *GUI* classes, such as `SearchComplData` (Section 1.2) and Façade classes that are used as entry-points for different layers. They contained drift symptoms, such as large methods associated with the accidental handling of exceptions from non-related components (Section 1.2).

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 rules bound to the same architectural concept. These symptoms are referred to as *concept-related*. For instance, rules AER1 and ADR1 (Section 3.1.1) rely on the *GUI* concept and detect both degradation symptoms that occur in `SearchComplData` (Section 1.2). 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 hybrid rules for concepts related to patterns and styles (Section 4.4.2). Therefore, it would be difficult to detect these concept-related symptoms through the use of individual techniques (Section 2.5) 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 11). MIDAS was developed using a middleware environment in charge of strictly enforcing the conformance of its implementation to the intended architecture (MALEK, 2007). It means that no interaction violation (i.e., erosion symptom) would remain in the code and, therefore, this system's versions did not exhibit any erosion symptom (GARCIA et al, 2009).

However, the quality of MIDAS architecture had progressively declined until the point where a major restructuring was required (GARCIA et al, 2009). 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 non-related services. In other words, they were increasingly manifesting anomalies related to the "single responsibility" principle (MARTIN, 2002). The MIDAS architecture significantly decayed due to the continued incidence of architectural drifts (GARCIA et al, 2009; MACIA et al, 2012). 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 **TamDera** (Section 3.1.3). More importantly, we observed in the MIDAS case that the hybrid 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 **TamDera** 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 can favor drift problems in later versions and vice-versa (Figure 12). For instance, in HealthWatcher, several method declarations were signalizing exceptions to other components, but those exceptions were supposed to be internally handled. For instance, Figure 12 illustrates the `AbsFacade` whose method `updateEmployee` has cases of erosion symptoms related to the throwing of exceptions. Those exception declarations were placed in methods in parent classes (e.g., `AbsFacade`), and those erosion problems in turn caused drift symptoms in children classes (e.g., `HealthWatcherFacade`). 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 neighboring components.



```
public void updateEmployee() throws
        TransactionException, RepositoryException,
        ObjectNotValidException, ...
        // 4 exception were remoded;

public void update() {
    try {
        getPm().beginTransaction();
        employeeRecord.update();
        getPm().commitTransaction();
    } catch (TransactionException e) {
        log(e);
        getPm().rollbackTransaction();
        throw e;
    } catch (ObjectNotValidException e) {
        log(e);
        getPm().rollbackTransaction();
        throw e;
    }
  //5 similar catch blocks were removed
}
```

Figure 12. Erosion symptoms caused drift problems in HealthWatcher

There were also 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 forced the Controller to access information made available from different components, thereby contributing to the establishment of unintended dependencies between the former and the latter ones. The analysis revealed that 66% of drift symptoms in Controller classes were sources of interaction violations emerging in later versions.

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 could be 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 hybrid 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 on the detection and removal 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 1.2). These scenarios might suggest 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 2.5) are not enough to enable developers to prevent architectural degradation.

4.4.2.
Reuse analysis

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 4.2). 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 3.1.4) to multiple projects.

Table 3. Reuse of anti-degradation rules

	MM			MA		HW			HA		MIDAS	
	1	**4**	**7**	**1**	**4**	**1**	**4**	**8**	**1**	**4**	**AFT**	**BEF**
# of reused AE rules	19	19	23	20	22	16	32	48	18	50	15	15
# of reused AD rules	20	21	26	23	25	27	36	42	31	54	13	13
# of reused ADG rules	39	40	49	43	47	43	68	90	49	73	28	28
# of ADG rules	47	48	59	54	68	67	94	117	73	104	42	46
% of RADG rules	82	83	83	79	70	64	72	76	67	70	66	60

AE = anti-erosion; AD = anti-drift; ADG = anti-degradation;

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 4.2). The definition of each single style or pattern is often formed by a cohesion set of component (anti-drift) and interaction (anti-erosion) constraints (Section 3.1.5). For instance, the Controller classes of MobileMedia realize the design pattern Chain of Responsibility (CoR) (GAMMA et al, 1995). This pattern reduces the coupling between the sender of a request to its receiver by delegating the request handling to multiple objects.

Listing 14 presents part of a reusable hybrid 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. The interface to clients as well other concepts and rules of the CoR were removed from Listing 14 for illustrative purpose. First, it defines an anti-drift rule constraining the coupling strength of each concrete class handling the request (`ConcreteHandler`, lines 01-04). 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 1.2): while the former ones enforce structural properties of modules realizing pattern concepts, the latter ones constrain their interaction with other architecturally-relevant concepts of the system.

Listing 14

```
01: constraintset ConcreteHandler {
02:  thresholds: LOW_COUPLING
03:  CBO < LOW_COUPLING
04: }
05: Client cannot-invoke ConcreteHandler
```

Significant detection of degradation symptoms by reused rules. As previously mentioned, 72% of the rules were reused. The remaining 28% were particularly defined for specific project concepts. More importantly, we observed that a significant number of erosion and drift symptoms were detected by reused rules. They were responsible for detecting on average 75% of the existing symptoms. Table 4 illustrates the effective reuse (Section 4.3) for each system version. These measures represented a balance between the reused rule percentage and the symptoms detected by them. In other words, 72% of the rules were reused, and they were responsible for detecting, approximately, 75% of all degradation symptoms. The 28% of the remaining (non-reused) rules detected 25% of the degradation symptoms. Therefore, the reused rules had similar efficiency to detect architectural deviations in comparison to the non-reused rules unique to each project.

Overriding and anti-drift rules. There was a need to subtly override reused rules in 11% of the cases (TAMDERA, 2012). For instance, as we mentioned, the concept `GUIHW` overrides anti-drift rules from `GUI` to impose more restrictive constraint boundaries (Section 3.1.4.2). These boundaries are used to capture particular symptoms in HealthWatcher 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 MobileMedia analysis as `ViewMM` is also a sub-concept of `GUI`. In fact, these restrictive boundaries are not applicable for the `ViewMM` code elements (Figure 8). Rule overriding (Section 3.1.4.2) 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 4.1). For instance, concept mappings need to be often overridden to consider: (i) the inclusion of new code elements in the AspectJ implementation, and (ii) the rename 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

	MM			MA		HW			HA		MIDAS	
	1	**4**	**7**	**1**	**4**	**1**	**4**	**8**	**1**	**4**	**BEF**	**AFT**
# of DG	24	41	55	20	33	116	140	159	85	105	50	46
# of DGRR	21	37	46	16	27	85	108	113	58	81	28	25
ERR	87	90	83	80	81	73	77	71	68	77	56	54

DG = degradation symptoms; DGRR = degradation symptoms detected by reused rules; ERR = effective reuse of rules

Detection of the same degradation symptom in multiple projects. The reused rules were also effective in the detection of the same degradation symptom manifesting across different projects. In order to perform this analysis, we selected a representative set of pairs of system versions, which were sharing reusable rules. Figure 13 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 systems. The rules that detect degradation symptoms in both systems are called *common reused rules* in Figure 13. Their percentage (dark grey bar) is computed from the total number of rules defined for the pair of versions (Section 4.3). 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 HealthWatcher and MobileMedia applications. Examples of these rules are: (i) anti-erosion rules constraining component interactions imposed by architectural styles (e.g., R4 - Figure 8), (ii) anti-erosion rules that restrict the access of specific-component exceptions, (iii) anti-drift rules to constrain size, complexity and coupling of particular components, such as View, Model, GUI and Data (e.g., R3 - Figure 8), and (iv) both erosion and drift rules associated with architecturally-relevant design patterns used in both systems, such as the Command (GAMMA et al, 1995).

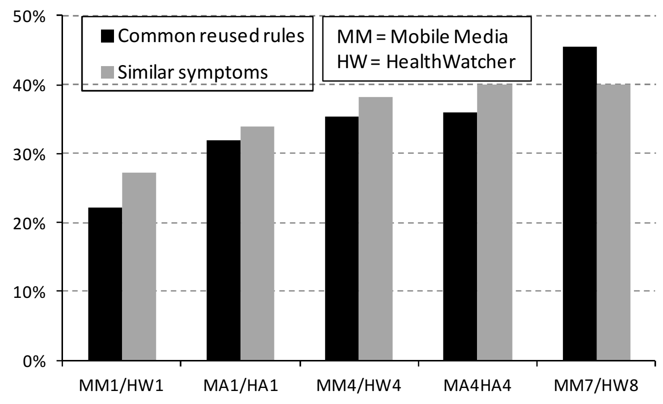


Figure 13: Effective reuse of common rules in multiple projects

4.4.3. Discussion

Our results (Section 4.4) confirmed the expectation about the frequency of co-occurring erosion and drift symptoms in the same (or inter-related) modules. Regarding the research question Q1, the results suggest that the number of modules containing both forms of degradation is significant. On average 45% of the HealthWatcher and MobileMedia classes which suffered from at least one symptom, have occurrences of both erosion and drift symptoms (Section 4.4.1).

These results are particularly interesting as the same concept definition can group hybrid rules for detecting erosion and drift symptoms and, thereby, promoting a full prevention of architectural degradation. Then, if an erosion symptom is detected, for example, the developer can reactively check whether drift symptoms (defined together with the anti-erosion rule) are possibly affecting the same module. It might be that the anti-drift rules are not accurate enough to detect the drift symptom given imperfections the metrics and thresholds adopted in such rules (MACIA et al, 2012).

In addition, many inter-related drift and erosion symptoms do not necessarily affect the same modules in the code (e.g., Chain of responsibility case – Section 4.4.2), which make them difficult to detect together using existing anti-drift and anti-erosion techniques (Section 2.5). The analysis of the MIDAS project also confirmed that automated detection of erosion rules was not enough to prevent architectural degradation (Section 4.4.1). Developers did not observe the progressive manifestation of drift symptoms in modules where erosion-related constraints were being enforced.

The study also evaluated the amount of anti-erosion and anti-drift rules that were reused in each project version. Taking into consideration the research question Q2, our analysis revealed that on average 75% of the rules to a particular project were reused from a source of general rules. These reused rules are often the cases of constraints to architectural styles, design pattern as well as strategies for detecting recurring drift-related anomalies (Section 4.4.2). Therefore, this may better foster the specification of rules governing architectural concepts as architects do not need to define them from scratch.

4.5.
Threats to validity

This section presents threats that might hinder the validity of the conclusions made in our study (Section 4.4). They are presented below.

Choice of the target applications. In empirical studies, the results are always limited to the scope of the selected applications. We tried to mitigate this threat by selecting applications from different domains and developed by different programmers. We and other researchers should replicate the study presented here to embrace the analysis of other target applications, following different architecture decompositions. Then, we could reach a better generalization of the results. However, we should highlight that these applications should be in conformance to the requirements described in Section 4.1. For instance, it would not be worth to evaluate the reuse of anti-degradation rules in systems that follow completely different design decisions.

Validity of the architecture models. Another issue possibility threatening the conclusion validity is how the architectural concepts and their anti-degradation rules were defined. They directly impact the measurement of reused and non-reused rules as well as the degradation symptoms detected by them. Some could claim that our previous knowledge of all the concepts and rules for all the systems artificially facilitated the reuse achievement with TamDera rules. To reduce the influence of this treat, we specified the concepts referring to architectural components presented in high-level component diagrams for all projects (Section 4.3). We also performed a detailed revision with the architects of each system to guarantee that the defined concepts capture the constraints associated with the intended architecture. However, we should recall that reuse obviously does not occur for free anyway; no reuse can be promoted if there is no effort upfront to anticipate general rules applicable to an organization or to a particular domain.

Validity of architectural degradation symptoms. We evaluated the efficiency of reused rules in terms of the amount of degradation symptoms they are able to detect. The study used previous reports about architectural degradation and refactorings in each application as an “oracle” (i.e., a reference model) to retrieve the degradation symptoms. This may externally impact the conclusion results. However, we also consulted the architects to confirm several reported symptoms of architectural degradation.

Validity of the detection strategies. A threat to construct validity includes the suite of metrics (and thresholds) used for detecting drift symptoms in each system. They are directly related to the amount of drift symptoms that are detected in the evaluation of effective reused rules. We focus on a classical suite of metrics for quantifying size, complexity, cohesion and coupling to evaluate modularity properties of system modules. We adopted these metrics because they are often used and supported by commercial tools (TOGETHER, 2012). They have also been used in previous studies to detect architectural degradation symptoms (MARA et al, 2011; MACIA et al, 2012).