1
Introduction

As systems evolve, the preservation of their software architectures (PERRY and WOLF, 1992) plays a crucial role on the longevity of software systems (EICHBERG et al, 2008; HOCHSTEIN and LINDVALL, 2005; UBAYASHI, 2010). However, architectural degradation (HOCHSTEIN and LINDVALL, 2005) is a long-standing problem in software engineering. The architecture of software systems is well-known for increasingly degrading through the maintenance and evolution stages (HOCHSTEIN and LINDVALL, 2005; PERRY and WOLF, 1992). Hochstein and Lindvall introduced the term architectural degradation to refer to the continuous quality decline of architecture designs in evolving software systems (HOCHSTEIN and LINDVALL, 2005). More specifically, the actual architecture in the system implementation starts to depart from the intended architecture as the system evolves (GARCIA et al, 2009; WERNER et at, 2011). Every code change may contribute to increasing the mismatch between the implemented and the architecture (Aldrich, 2010; TERRA and VALENTE, 2009).

The architectural degradation also manifests in architecture-driven projects, i.e., in projects where there is a concern in explicitly prescribing upfront the intended software architecture. In fact, even architecture-driven projects of widely-used software systems, such as Mozilla, FindBugs, ArgoUML, Enterprise Java Beans and Jakarta have shown to be susceptible to architectural degradation (EICHBERG et al, 2008; LI, 2010; MOHA et al, 2010; MERKLE, 2010). In particular, MacCormarck et al reported in 2006 that the exceptionally–tight coupling of Mozilla's components was the main cause for its complete re-engineering in 1998 (MACCORMACK et al, 2006). The process consumed five years of rewriting two million lines of source code thanks to its full architectural degradation.

Software architecture is concerned with the definition of architecture components and their interactions as well as with constraints on both of them (GARLAN and SHAW, 1993). Symptoms of architectural degradation arise

through the processes of *architectural erosion and drift* (HOCHSTEIN and LINDVALL, 2005; PERRY and WOLF, 1992). Erosion occurs when constraints governing the interaction of architecture components are violated (PERRY and WOLF, 1992). Therefore, we also refer to *interaction violations* as erosion symptoms in this dissertation. A typical example of erosion symptom is an unintended dependency established between two components. In contrast to erosion, drift symptoms imply the violation of individual component's constraints (PERRY and WOLF, 1992). Typical examples of drift symptoms are components that violate modularity principles, such as the "narrow component interface principle" or "single responsibility principle" (GARCIA et al, 2009; PERRY and WOLF, 1992). The violation of such modularity principles is perceived when components of the actual implemented architecture starts to exhibit structural anomalies, such as interface bloat and components realizing multiple responsibilities (GARCIA et al, 2009; MACIA et al, 2012).

1.1. Problem statement

In order to fully prevent architectural degradation, both erosion and drift symptoms need to be detected and removed from the actual implementation of a software system. In fact, the erosion and drift processes are often intertwined in spite of their conceptual differences (MACIA et al, 2012; PERRY and WOLF, 1992). Violations of component constraints (i.e., drift symptoms) may foster the later introduction of interaction violations (i.e., erosion symptoms) or vice-versa, thus, the same modules in a program become the locus of both drift and erosion symptoms.

For instance, drift symptoms are related to complex components or interfaces as they are sources of violation of modularity principles (Section 1.2). Hence, if any of these drift symptoms remains undetected, it may provoke the emergence of the erosion symptoms along the system's change history (GARCIA et al, 2009; MACIA et al, 2012). In fact, drift symptoms impair design or implementation comprehension, thereby contributing to the unconscious introduction of interaction violations in their programs (PERRY and WOLF, 1992). This possibly means that if a drift symptom is detected early in the project

history, it is likely that many other inter-related drift and erosion symptoms will be prevented to occur later. Hence, the longevity of software projects largely depends on the early detection and repair of both symptoms of architectural degradation.

Therefore, architects should elaborate strategies for detecting simultaneous occurrences of both degradation symptoms. These strategies are often based on the specification of architectural rules based on the intended architecture (EICHBERG et al, 2008; MOHA et al 2010; TERRA and VALENTE, 2009; MARA et al, 2011). However, the specification of such architectural rules is repetitive as they are often similar across different software projects. If fact, it has been noticed that similar degradation symptoms can infect several projects. This phenomenon occurs, for instance, when multiple projects in the same company share similar architectural constraints either to particular components or to their interactions. For example, many architectures of system from a same company share the adoption of the same architectural and design patterns (GAMMA et al, 1995; BUSCHMANN et al, 2007). This may also be the case when projects from different companies are from the same domain, thereby possibly sharing similar architecture decisions.

The observations above call for an approach that enables software developers to: (i) detect both forms of architectural degradation in the evolving implementation of their systems, and (ii) encourage developers to reuse specification of architectural rules in order to avoid recurring symptoms of architectural degradation across different projects. The next section presents a motivation example in order to better illustrate the aforementioned problems.

1.2. Motivating example

The processes of architectural erosion and drift are further illustrated using a motivating example. This section also discusses their relationship (Section 1.2.1) as well as the main characteristics of these processes in projects that follow similar design decisions (Section 1.2.2). From herein, architectural degradation, architectural drift and architectural erosion are also referred to simply as degradation, drift and erosion.

Example of architectural erosion. Figure 1 shows an example of an architectural erosion symptom in the MobileMedia system (FIGUEIREDO et al, 2008). This system realizes the architecture pattern Model-View-Controller (MVC) (BUSCHMANN et al, 2007). Each component is realized as a set of modules (classes) in the source code. For conciseness, only a few classes are shown in the figure. The erosion problem concerns an architecture constraint governing the intended exception handling policy: exceptions are incorrectly propagated through module interfaces across system components. As an example, the class `BaseController` defined in the Controller component invokes the Data services provided by `AlbumData`. `BaseController` ends up handling exceptions (e.g., `PersistenceException`) thrown by `AlbumData`, which should have been handled within the Model component. As a consequence, dependencies between code modules realizing the Controller and Model components are undesirably introduced. They lead to interaction violations, thereby contributing to the degradation of the intended software architecture of MobileMedia.

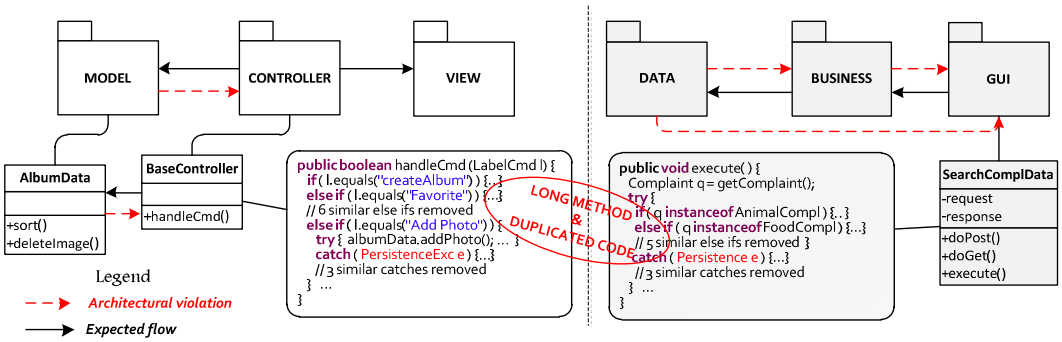

Figure 1. MobileMedia (left) and HealthWatcher (right) architectures

Example of architectural drift. Let us consider the aforementioned case of the class `BaseController` in Figure 1. This class plays a crucial role in the implemented architecture as it is in charge of realizing the *Controller* component in the architecture specification. However, this class is the source of a code anomaly, called *Large Class* (FOWLER, 1999). The class defines many methods, each of them realizing various non-cohesive functionalities of the system (e.g., video deletion and photo sorting). A side effect negative consequence is that it ends up contributing to the manifestation of an architectural drift symptom, called *Ambiguous Interface* (GARCIA, 2009). The implementation of this class provides an over-generalized interface - `handleCmd` - for handling all commands. This

means that the actual interface and the implementation of the `BaseController` module in the source code (and, therefore, the Controller component) are aggregating several responsibilities from different service requests that should be implemented independently. This drift symptom was addressed in a later version by MobileMedia developers. The `BaseController` class was further decomposed into smaller classes; each of them implementing a specific controller responsibility. The goal was to explicitly decompose the interface of the Controller component into simpler ones.

1.2.1. Erosion and drift symptoms tend to be intertwined

Previous works have shown how, in many cases, erosion and drift symptoms are somehow related (PERRY AND WOLF, 1992; MACIA et al, 2012). Architectural drift symptoms often foster the later introduction of erosion symptoms and vice-versa. The left-hand side of Figure 1 illustrates an example of the relationship between erosion and drift symptoms in the MobileMedia. The class `BaseController` implements different services through the interface `handleCmd`. The amount of services exposed by `handleCmd` 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 original MVC decomposition, thereby characterizing the occurrence of several interaction violations.

This example illustrates a direct relation between erosion and drift phenomena. Hence, in order to detect the architectural 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. In addition, intertwined occurrences of these symptoms are also a sign of severe stages of architectural degradation (MACIA et al, 2012). More specifically, recent studies have revealed that modules exhibiting both forms of degradation tend to manifest

more severe architecture instabilities in a project history than modules containing just one singular form of degradation (MACIA et al, 2012, MARA et al, 2011).

1.2.2. Similar degradation symptoms can infect several projects

Similar degradation symptoms can manifest in several projects (Section 1.2.2). To illustrate this phenomenon, the right part of Figure 1 depicts the architecture of the HealthWatcher system (HEALTHWATCHER, 2012), which follows the decomposition prescribed by the Layer architectural pattern (BUSCHMANN et al, 2007). Certain modules of HealthWatcher suffer from similar erosion and drift symptoms as those occurring in MobileMedia modules. For instance, `SearchComplData` introduces interaction violations through exception propagation behavior, similarly to `BaseController` in MobileMedia. Also, it reifies drift symptoms related to the implementation of independent responsibilities, such as GUI and Persistence. In this way, `SearchComplData` has a similar code structure to `BaseController (code containing several catch blocks)`. Unlike `BaseController`, it does not suffer from an interface bloat. Furthermore, these classes implement components with different responsibilities (i.e., Controller and GUI).

The re-occurrence of similar degradation symptoms in both classes (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 rules for detecting recurring symptoms of architectural erosion (via anti-erosion rules) and drift (via anti-drift rules). For instance, an architectural specification can group correlated anti-erosion and anti-drift rules that constrain MobileMedia component elements. More specifically, 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 symptoms related to interface bloat, such as large methods in `BaseController`. Therefore, this abstraction can be reused in HealthWatcher to constrain the dependencies between non-adjacent layers, the exception handling behavior and the size boundaries for GUI components such as `SearchComplData`. As we can perceive, the reuse of hybrid specification of

anti-erosion and anti-drift rules can help developers to reduce the effort of specifying such rules repeatedly in each project. These specifications are also referred in this dissertation as *hybrid rules*. Finally, companies would maintain a uniform and reusable base of anti-erosion and anti-drift rules in several projects to save resources and time in the specification of similar rules.

In addition to the reuse "as is", 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 (i.e., thresholds), such as those applicable to `BaseController`, should be adjusted to other systems (e.g., `SearchComplData` in HealthWatcher).

In summary, the motivating example illustrated the need for supporting: (i) the hybrid specification of anti-erosion and anti-drift rules; and (ii) reuse of hybrid rules from a base of uniform anti-degradation rules (i.e., the set of anti-erosion and anti-drift rules) in multiple projects.

1.3. Limitations of related work

Several techniques have been devoted to support architectural degradation detection (EICHBERG et al, 2008; MOHA et al 2010; MARA et al, 2011; MARWAN and ALDRICH, 2009; MERKLE, 2010; OLIVEIRA, 2011; SANGAL et al, 2005; TERRA and VALENTE, 2009; UBAYASHI et al, 2010). However, there are two main problems with the current state of the art. First, existing approaches usually promote the exclusive detection of either erosion (EICHBERG et al, 2008; MARWAN and ALDRICH, 2009; TERRA and VALENTE, 2009; UBAYASHI et al, 2010; OLIVEIRA, 2011) or drift symptoms (MARINESCU, 2004; MOHA et al 2010; MARA et al, 2011). Even though these symptoms are often inter-related, these approaches are limited to solely focus on just one particular degradation symptom. The detection of either erosion or drift symptoms may not prevent the increasing decay of the software architecture (PERRY and WOLF, 1992; HOCHSTEIN and LINDVALL, 2005). For instance, the removal of the former symptoms may not imply the amelioration of the latter symptoms.

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 inverse is also true. Given the simultaneous occurrence of erosion and drift symptoms (HOCHSTEIN and LINDVALL, 2005; MACIA et al, 2012), architects should be able to elaborate hybrid strategies for detecting both forms of degradation symptoms. These strategies ought to rely on the specification of architectural rules for drift and erosion prevention.

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 (GAMMA et al, 1995). Ideally, architects should be able to reuse anti-drift and anti-erosion rules across projects adhering to similar architecture decompositions. Nowadays, languages for describing architectural rules (EICHBERG et al, 2008; TERRA and VALENTE, 2009; UBAYASHI et al, 2010) do not provide this support. All these limitations of related work are discussed in more detail in Chapter 2.

1.4. Proposed solution and contributions

This dissertation addresses the limitations of related work by proposing a new Domain-Specific Language (DSL) for specifying rules to detect architectural degradation (Chapter 3). The proposed language, called **TamDera**[1] is the main contribution of this work and has two distinguishing features: (i) support for specifying and blending rules for erosion and drift detection in a unified way, and (ii) support for hierarchical and compositional reuse of anti-drift and anti-erosion rules.

More specifically, the language provides a single abstraction, called *architectural concept*, which allows architects to impose anti-drift rules on components and map them to implementation elements. Similarly, architects can also impose anti-erosion rules on component interactions. These rules can be

[1] **TamDera** stands for “Taming Drift and Erosion in Architecture”.

reused through compositional and hierarchical reuse mechanisms. For instance, anti-erosion and anti-drift rules can be reused through the specialization of rules associated with abstract architectural concepts.

The second contribution is a tool to supports the language usage and rule enforcement in the source code (Chapter 3). The tool is unique as it integrates facilities for detecting both erosion and drift symptoms. Its implementation provides features to: (i) easily extend the set of strategies available for specifying anti-degradation rules, and (ii) check if the set of specified anti-degradation rules is inconsistent.

As a final contribution, we evaluate the usefulness of supporting blend and reuse of architectural rules. The study encompassed 21 releases of 5 projects, and more than 600 anti-degradation rules (Chapter 4). The findings provide evidence for the usefulness of detecting the co-occurrence of erosion and drift symptoms in multiple projects and the value of supporting reuse of single and hybrid rules. Our analysis pointed out several cases where the exclusive detection and removal of a particular symptom was not sufficient to prevent architectural degradation. Even worse, by exclusively detecting erosion symptoms, developers neglected severe drift symptoms in later versions, or vice-versa. In addition, we observed that most rules defined in a particular project could be reused from rules previously defined for architectural and design patterns . They were responsible for detecting the majority of the degradation symptoms in each project.

The contributions of this work were reported in papers, which have been published or are under submission. Some of the papers (in particular, papers #3 and #4) represent preliminary results of our research work and only have marginal relation to this dissertation. However, those studies were instrumental to reveal the research problem being addressed in this dissertation. These papers are listed as follows:

1. GURGEL, A., MACIA, I., GARCIA, A., MEZINI, M., EICHBERG, M., VON STAA, A., MITSCHKE, R. **TamDera**: Blending and Reusing Rules for Architectural Degradation Prevention; In Proceedings of 20th Symposium on the Foundations of Software Engineering (FSE'12). 2012; (in submission).

2. GURGEL, A., DANTAS, F., GARCIA, A. Um Estudo de Composições de Padrões de Projeto em CaesarJ. In Proceedings of the IV Latin American Workshop on Aspect-Oriented Software Development - LA-WASP 2010. pags. 30-36. 2010.
3. GURGEL, A., DANTAS, F., GARCIA, A. On-Demand Integration of Product Lines: A Study of Reuse and Stability. In Proceedings of the 2nd International Workshop on Product Line Approaches in Software Engineering - PLEASE '11 at ICSE'11, 2011; pags 35-39. 2011.
4. DANTAS, F., GURGEL, A., GARCIA, A. Towards a Suite of Metrics for Advanced Composition Mechanisms. In Proceedings of the 2nd International Workshop on Empirical Evaluation of Software Composition Techniques -ESCOT 2011 at ECOOP'11. Lancaster, United Kingdom. 2011.

The first paper reports the key results of this dissertation encompassing the design and evaluation of the **TamDera** language. The paper also provides a brief description of the tool. Many of our insights on the language design were gathered during the study reported in the second paper. The study involved the implementation of a code library of several design patterns. This experience enabled us to understand the importance of enforcing both anti-erosion and anti-drift rules in implementation of these patterns. The third and the fourth papers focused on studies related to the integration and evolution of multiple software product lines. They also enabled us to grasp the importance of a unified approach for describing and enforcing anti-erosion and anti-drift rules.

1.5. Dissertation structure

The next chapters have the following purposes:

Chapter 2- *Background and Related Work*: presents general background, basic terminology, and outlines related work on architectural degradation prevention.

Chapter 3- *The* **TamDera** *Language*: presents the **TamDera** language which allows the blend and reuse of anti-erosion and anti-drift rules. The chapter also depicts a tool implementation and design that supports **TamDera.**

Chapter 4- *Evaluation*: presents and discusses results of a study tailored for evaluating the co-occurrence of erosion and drift symptoms and reuse of hybrid rules in multiple contexts.

Chapter 5- *Conclusion*: discusses the conclusions and the contributions for this dissertation, and describes planned future work.