2 Background and related work

As systems evolve, their size and complexity increasingly grow. As a result, the preservation of their software architectures plays a crucial role in the longevity of software systems (PERRY and WOLF, 1992; HOCHSTEIN and LINDVALL, 2005). Central to the architecture preservation is the use of effective techniques that verify the conformance of the system implementation with respect to the intended architecture. However, the verification of the intended architecture design in the source code is widely recognized as a challenging task (ALDRICH, 2002; UBAYASHI et al, 2010).

This chapter presents terminologies and definitions associated with intended architecture (Section 2.1). Section 2.2 describes concepts related to architectural erosion and drift. The definition of intended architecture also includes the selection of architectural (Section 2.3) and design patterns and modularity principles (Section 2.4). This chapter also outlines existing work on supporting the detection of architectural degradation symptoms (Section 2.5). Section 2.5.1 presents techniques that solely support the detection of erosion symptoms. Then, Section 2.5.2 introduces techniques aimed at supporting the detection of architectural drift symptoms. We refer to the motivating example (Section 1.2) in order to illustrate the anti-erosion and drift techniques. Finally, we overview the limitations of current techniques to holistically support the detection of both forms of architectural degradation symptoms (Section 2.5.3).

2.1. Basic terminology

Software architecture is concerned with the definition of architecture components and their interactions as well as with the definition of constraints on both of them (PERRY AND WOLF, 1992). *Components* are architectural entities which encapsulate a subset of the system's functionalities (TAYLOR et al, 2009). Each component *interacts* with other components in the system in order to access their exposed functionalities. They restrict access to that subset via interfaces that *constrain* which functionalities are exposed to the components (TAYLOR et al, 2009).

A component of the architecture description is realized by one or more modules in the implementation. The term *module* is used to represent source code elements, such as a package, an (implementation-level) interface or a class, which contribute to the implementation of a coherent unit of functionality (CLEMENTS et al., 2002). In certain cases, a module in the code can also partially contribute to the implementation of a component (EICHBERG et al, 2008; TAYLOR et al, 2009). This means that inner elements of a module can eventually contribute to the implementation of different architectural components. *Inner module elements* refer to specific methods of a class and fields.

The *intended architecture* (or prescriptive architecture) is formed by the explicit decisions made by the designers on the selection of components, their interactions, and constraints related to them (TAYLOR et al, 2009). The intended architecture decisions are mostly made prior to the system's construction (GARLAN and SHAW, 1993). However, these decisions can be revisited and consciously changed as the system evolves (TAYLOR et al, 2009).

On the other hand, the *implemented architecture* describes how the system has been actually built (TAYLOR et al, 2009). In software projects, the implemented architecture often does not match the intended architecture (TAYLOR et al, 2009). Many prescribed architecture decisions can be undesirably violated by the actual implementation of a system. These violations might be introduced not only in the first version of the system implementation, but also later through code changes during system evolution.

These violations represent architectural degradation symptoms (Section 2.2). In particular, the continuous adherence of constraints governing the components and their interactions in the source code is hard to be verified. The main reason is the frequent lack of an one-to-one relationship between architectural components and implementation modules, as mentioned above. In other words, in many cases a same module realizes more than one component. In this dissertation, we are particularly concerned with these constraint violations.

2.2. Architectural drift and architectural erosion

The preservation of the intended architecture in the implementation is also related to ensuring that constraints governing components and their interactions are not violated (Section 2.1). The violations of such constraints on components and their interactions respectively characterize the process of erosion and drift (Section 1.1). It is important to highlight that we adapted the definitions of these processes presented in (MACIA et al, 2012; PERRY AND WOLF, 1992) to establish a direct relationship between those definitions and component and their interactions violations.

In the implementation, interactions between components are realized through code dependencies between their modules, such as method invocations, class inheritance, and object creation (EICHBERG et al, 2008). In this dissertation, the term *dependency* is used to refer to each of those structural relationships between modules in the code (TERRA et al, 2010). Each dependency has an explicit direction, i.e., a source and a target.

Interaction constraints impose how the different components interact and how they are organized with respect to each other implementation (PERRY AND WOLF, 1992). As an example, the GUI component from the HealthWatcher architecture (Figure 1) is not allowed to interact with the Data component. On the other hand, *component constraints* impose expected structural properties on modules realizing a component to the degree desired by the architect (PERRY AND WOLF, 1992). For instance, the GUI component presented in the motivating example has constraints establishing size boundaries to GUI modules.

The verification of interaction and component constraints is the source code are challenging tasks (SANGAL et al, 2005; UBAYASHI et al, 2010). The reason for this difficulty is manifold. First, the architecture design is comprised of a wide range of design decisions, such as the adopted set of architectural patterns and design patterns. Each single pattern establishes several constraints that are relevant to the intended architecture (Section 2.3). Second, the architecture design also involves the selection of modularity principles (Section 2.4) to be realized by specific (if not all) components of a software architecture. Each principle can also entail more than one constraint. Third, the violation of each single constraint represents a symptom of the architectural degradation process. Even the identification of a single component constraint violation is not trivial as the implementation of a single component is often scattered through multiple modules (Section 2.1). Fourth and more importantly, these several constraints are likely to be repetitive and inter-related (Section 1.2.2). Then, the approach to specify these constraints should be targeted at reducing architects' effort.

There are many complementary factors in mainstream software projects that contribute to the implemented architecture to depart from the intended architecture, thereby leading to drift and erosion symptoms. They range from: (i) deadline pressure and programmers that are unaware about intended architecture decisions (TERRA and VALENTE, 2009), to (ii) out-of-date architectural documentation (HOCHSTEIN and LINDVALL, 2005; MOHA et al, 2010) and lack of proper tool support for verifying the adherence of the source code to the intended architecture (MACIA et al, 2012).

2.3.
Architectural and design patterns

When defining the prescribed software architecture, many decisions need to be made. The selection of architectural patterns (BUSCHMANN et al, 2007) and design patterns (GAMMA et al, 1995) are basic steps in this process as they provide complementary reusable solutions to architecturally relevant problems. A pattern is a general reusable solution to a recurring problem when structuring a software system (GAMMA et al, 1995).

Architectural patterns. Software architects often use architectural patterns to guide the architecture building and understanding. *Architectural patterns*[2] (BUSCHMANN et al, 2007) are specifically targeted at addressing architectural level problems. Each architectural pattern solves a problem by prescribing a specific architecture solution in terms of a set of components and interactions as well as a set of constraints on how they can be used (CLEMENTS et al., 2010). Therefore, each pattern entails various architectural constraints to be enforced in

[2] In this dissertation, for simplification purpose, we use the terms architectural patterns (BUSCHMANN et al, 2007) and architectural styles (CLEMENTS et al., 2010) interchangeably.

the source code. More than one architectural pattern can be used in the prescribed architecture of a software system.

Model-View-Controller and Layers are well-known examples of architectural patterns (BUSCHMANN et al, 2007). The *Layer pattern* structures applications into a group of components, where each of them is realized as a *layer* that provides a cohesive set of services (BUSCHMANN et al, 2007). Layers interact with each other according to a strict ordering relation (CLEMENTS et al., 2010). More specifically, the modules realizing a layer at a specific level J, are allowed to interact with the layer at the level J - 1 by the use its public services. Hence, interfaces from the layer at the level J - 1 provide services used by the layer at the level J (BUSCHMANN et al, 2007). On the other hand, the *Model-View-Controller* pattern decomposes an interactive application into three types of components. The *Model* component contains the core functionality and application data. The *View* component displays information and realizes the graphical user interface, while the *Controller* component is in charge of handling user inputs.Data changes are consistently propagated from model to user interface via the controller. As a consequence, implementation modules realizing the view component cannot directly access services provided by the modules implementing the model component.

Design patterns. Design patterns refer to problems at the detailed design level and provide reusable common design structure which involves participating modules and their interactions. Each design pattern is also formed by a set of constraints governing these modules and interactions. Even though design patterns are intended to address problems emerging at the detailed design stage, software architects might want to explicitly select them. Their goal is to structure certain architectural components in their intended architecture description. In this dissertation, we refer to these cases as *architecturally-relevant design patterns*. In fact, it is often the case that architects realize upfront that they should enforce, for instance, that modules of certain components realize certain design pattern constraints.

Façade and Chain of Responsibility are popular examples of design patterns (GAMMA et al., 1995) that are employed to structure the intended software architecture (EICHBERG et al, 2008; UBAYASHI et al, 2010). The pattern *Façade* is often used in system architectures that are decomposed in subsystems

(GAMMA et al., 1995), where all the interfaces of a subsystem should be unified in a singular interface, the so-called façade. The goal is to provide a higher-level interface for the subsystem to make it easier to use. It also intends to decouple the subsystem from clients and other subsystems. This pattern is also usually referred in intended architecture descriptions following the layers pattern in order to define that each layer should have only a single entry point. As a consequence, the access to any service provided by the layer is made through the façade. In addition, only the façade is allowed to access the internal services realized by the implementation modules of a layer.

On the other hand, the pattern *Chain of Responsibility* (GAMMA et al, 1995) aims to avoid coupling between a sender of a request (i.e., client) and its receivers by allowing more than one module to handle the request. Thus, the receivers compose a chain where the request passes until an appropriate receiver handles it. Thus, the pattern constrains how the implementation of services is distributed in a chain of modules named handlers. The pattern also establishes an interface which must be implemented by all handlers (i.e., concrete handlers). This interface provides services which are exposed to component clients. Hence, clients must access the interface to send their requests. In other words, they are not allowed to directly access services provided by concrete handlers.

2.4. Modularity principles

Modularity is concerned with the logical decomposition of a software system into components (BUSCHMANN et al, 2007). In other words, modularity in architecture design refers to a logical partitioning of the software architecture that allows it to become manageable for the purpose of implementation, maintenance and evolution (BUSCHMANN et al, 2007). Central to modularization is deciding how to decompose the components that form the logical structure of an application.

In order to achieve a modular architecture, a number of modularity principles are selected and realized by the prescribed architecture. For instance, a basic modularity principle is to minimize the strength of the coupling of each system component (GARCIA et al, 2009; MARTIN, 2002). In addition, different or unrelated responsibilities should be separated from each other in a software architecture decomposition. These responsibilities should be attached to different components (PERRY AND WOLF, 1992; GARLAN AND SHAW, 1993).

A complementary principle is the single responsibility principle (MARTIN, 2002), which determines that each component should be cohesive, i.e., being in charge of addressing a single responsibility. The term cohesive refers to components that have a well-defined purpose. The notion of component cohesion is closely related to the narrow interface principle. The latter states that architects should hide the complexity of each component behind an abstraction that has a simple interface; the interface indicates how the elements interact with the entire system (PERRY AND WOLF, 1992; BUSCHMANN et al, 2007).

The solutions documented by many architectural and design patterns are also driven by the application of one or more modularity principles. For instance, the layers pattern is intended to reduce the coupling of the layer components by ensuring that each of them only interacts with the adjacent layers. Therefore, by reusing existing patterns in the appropriate contexts, the architect is likely to promote the application of the modularity principles. However, the mere choice of architectural pattern does not prevent possible symptoms of architectural drifts (Section 3.1.5). For example, two specific layers might become strongly coupled. The strong coupling of modules realizing two layers might represent, for instance, that either: (i) one of the layers is realizing a responsibility that should be implemented by the other layer, or (ii) these two layers should be merged in a single component.

2.5. Related work

The prescribed architecture consists of a set of component and interaction constraints (Section 2.1). Some of these constraints are defined in the architectural and design patterns (Section 2.3) being adopted. Other constraints in the prescribed architecture are also derived from the conscious selection and application of modularity principles by software architects (Section 2.4). The adherence of constraints governing the components and their interactions in the source code are hard to be verified and enforced (Section 2.2). The violation of component and interaction constraints represents symptoms of architectural drift and erosion, respectively (Section 2.2).

Over the last decades, several anti-degradation techniques have been proposed to support the detection of architectural degradation symptoms. Each of these techniques usually relies on the specification of anti-degradation rules to constraining either properties of individual components or constraining their interactions. However, to the best of our knowledge, there is no technique whose main purpose is to simultaneously support the detection of both erosion and drift symptoms. Each technique is limited to only directly support the detection of either erosion symptoms (Section 2.5.1) or drift symptoms (Section 2.5.2). However, some techniques can be adapted in order to partially support the detection of both symptoms (Section 2.5.3).

2.5.1. Anti-erosion techniques

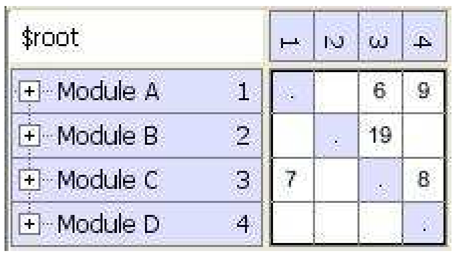
Techniques for detecting erosion symptoms provide mechanisms to explicitly define the intended architecture of a system through the description of adopted rules. These rules are limited to specifying the interaction constraints (Section 2.1) and are called *anti-erosion rules* in this dissertation. These techniques often provide automated support to check if the system's implementation is in conformance to the intended architecture.

They often rely on static analysis of the implemented architecture to detect erosion symptoms (TERRA and VALENTE, 2009; UBAYASHI et al, 2010). These symptoms are distinguished in *divergence* and *absence* violations (KNODEL and POPESCU, 2007). A *divergence* violation takes place when the dependency constraints specified in the intended architecture are not respected in the system implementation. On the other hand, an *absence* occurs when the implemented architecture does not establish an expected dependency prescribed in the intended architecture. In the following, we present the representative anti-erosion techniques: Lattix's Dependency Manager Tool (SANGAL et al, 2005), the Vespucci (EICHBERG et al, 2008) and Dependency Constraint Language (TERRA and VALENTE, 2009). Our criterion was to select techniques which are instrumentally supported by industry tools and techniques that encompass a wide scope of different abstractions. In particular, the techniques involve abstractions range from pseudo-natural languages to architecture models. Section 2.5.1.4 also describes other anti-erosion techniques.

2.5.1.1. Lattix's Dependency Manager tool

Lattix’s dependency manager tool (LDM) automatically extracts a dependency structure matrix (DSM) (SULLIVAN et al, 2001) from source code to represent module dependencies for a single project (SANGAL et al, 2005). The goal is to enable architects to detect unexpected dependencies between modules that violate constraints on component interactions.

A DSM is a square matrix where rows and columns represent modules. A cell is marked when there is a dependency between the respective modules of the selected row and column (SULLIVAN et al, 2001). For instance, module A (#1) has dependencies with module B (third column) and module D (forth column). The number in the cell indicates the strength (e.g., number of method invocations) of the dependency.



$root		1	2	3	4
+ Module A	1	.		6	9
+ Module B	2		.	19	
+ Module C	3	7		.	8
+ Module D	4				.

Figure 2. Dependency structure matrix

LDM also provides graphic diagrams to exhibit components and their modules (SANGAL et al, 2005). Users are able to establish architectural components by grouping modules presented in the DSM. LDM allows architects to visualize the DSM and establish rules for detecting only divergence violations (SANGAL et al, 2005). In particular, it provides only two different dependency relationships: *can-use* and *cannot-use* and, hence, architects can indicate whether a component can or cannot interact on another one. Therefore, architects can use LDM for checking whether GUI components accidently access Data components in HealthWatcher architecture (Section 1.2). However, LDM does not enable architects to detect absence violations.

2.5.1.2. Vespucci

Vespucci enables reasoning about anti-erosion rules at the levels of architecture design and implementation (EICHBERG et al, 2008). Vespucci uses declarative queries to group source code elements named *ensembles*. Architects can define an ensemble to represent each architectural component. An ensemble groups modules and inner module elements that structurally belong together (e.g., same role defined by an architecturally-relevant design pattern) and share similar anti-erosion rules. For instance, architects may define an ensemble to encompass façades that provide model services to controller elements.

In addition, Vespucci also introduces a graphical notation based on arrows and boxes for specifying expected and non-expected dependencies between ensembles (EICHBERG et al, 2008). As an example, we create three ensembles `GUI`, `Business` and `Data` to group code elements from each component of the HealthWatcher architecture (Figure 3). Figure 3 depicts anti-erosion rules governing the three ensembles. The architecture specification establishes that all direct access from `GUI` to `Data` components and vice-versa are interaction violations. In Vespucci, interaction violations are represented by the notation "!" (Figure 3).

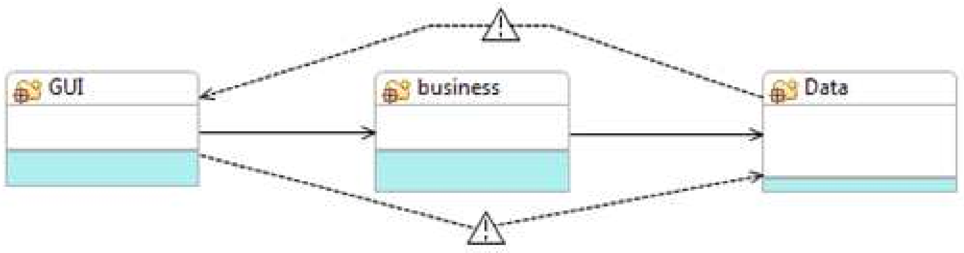


Figure 3. Modeling anti-erosion rules in Vespucci

Vespucci uses the Bytecode Analysis Toolkit (BAT) (EICHBERG et al, 2008) and Prolog (CERI et al, 1989) for supporting the static conformance checking of anti-erosion rules in the system implementation. It uses BAT to generate the system prolog-based representation from its Java bytecode. The anti-erosion rules defined in the graphical specification are translated to prolog queries to support the checking of architectural violations.

2.5.1.3. Dependence Constraint Language

Dependence constraint language (DCL) is a domain-specific language that allows architects to constrain component dependencies in object-oriented systems (TERRA AND VALENTE, 2009). It supports the detection of divergence and absence violations. DCL uses imperative pseudo-natural statements to establish anti-erosion rules. It composes components by the grouping modules. In particular, architects are able to select a set of packages or classes filtered by common name expressions such as default suffixes and prefixes.

Listing 1 depicts the specification of three anti-erosion rules in DCL. At first, we define components that enclose the respective *GUI* and *Data* code elements. As we can notice, DCL follows a different terminology from ours (Section 2.1). DCL uses the keyword **module** to refer to architectural components. As an example of anti-erosion rule, the statement `'GUI cannot-invoke Data'` establishes that any `GUI` element is not allowed to invoke any service provided by any `Data` element.

Listing 1

```
module GUI: healthwatcher.view.*
module Data: healthwatcher.data.*
GUI cannot-invoke Data
```

2.5.1.4. Other anti-erosion techniques

Marwan and Aldrich developed SCHOLIA, an embedded language for documenting the system's architecture in the source code and checking its conformance with a prescribed architecture (MARWAN and ALDRICH, 2009). Morgan defined a domain-specific language to specify and check anti-erosion rules in the system implementation (MORGAN, 2007). Ubayashi et al. presented Archface, a programming-level interface to represent the intended architectural design and detect erosion symptoms in the system's implementation (UBAYASHI et al, 2010). Oliveira presented the PREViA approach which provides features for defining components and expected interactions in the intended architecture using UML class and component diagrams (OLIVEIRA, 2011). It prevents architectural erosion by evaluating the conformance between the system implementation with respect to the intended architecture.

2.5.2. Anti-drift techniques

Anti-drift techniques are aimed at supporting the detection of component constraint violations in order to reveal symptoms of architectural drift (MARA et al, 2011; MOHA et al, 2010; MARINESCU et al, 2005). In this case, developers specify the intended architecture of a system through *anti-drift rules*. These rules are often metrics-based strategies that quantify structural properties of modules realizing software components (MARINESCU et al, 2005; LANZA and MARINESCU, 2006). To be more exact, anti-drift techniques usually rely on detection strategies (MARINESCU, 2004) which are based on expressions that combine logic operators and static code metrics. The goal is detecting architecturally-relevant anomalous modules in the implementation (MACIA et al, 2012). In fact, anomalous modules in the implementation are often the source of relevant architectural drift symptoms (GARCIA et al, 2009; MACIA et al, 2012). Hence, the identification of such anomalous modules may reveal violations of certain modularity principles (Section 2.4) in the source code (MACIA et al, 2012). They represent the violation of intended constraints for one or more individual components.

Marinescu et al. presented iPlasma, a tool that relies on strategies (MARINESCU, 2004) to detect anomalous code elements (MARINESCU et al, 2005). Moha et al. presented a methodology to detect anomalous code structures by combining metric-based evaluations to structural properties of modules (MOHA et al, 2010). Mara et al. proposed Hist-Inspect which enables the definition and application of conventional detection strategies (MARA et al, 2011). In particular, we selected Hist-Inspect as representative tool to illustrate the anti-drift techniques.

Hist-Inspect. It declares each anti-drift rule as a detection strategy to identify occurrences of an implementation module anomaly. The tool supports conventional metrics, such as coupling (CBO) and lines of code (LOC). The anti-drift rules are defined in XML format as illustrated in Listing 2. This listing contains a strategy for detecting the GUI classes that may implement more responsibilities than desired (Section 1.2). It defines an anomaly named `unexpectedComplexGUI` that may be manifesting in classes (lines 07-09). It also instantiates a strategy to detect this anomaly (lines 03-05). This strategy (line 05) checks all system classes (i.e., including those classes that do not realize the GUI component) and retrieves the ones who have 100 or more lines of code or whose cyclomatic complexity is greater or equals to 5. These metrics were selected for illustrative purpose, and other metrics could be used for detecting similar or different drift symptoms.

The strategy may retrieve classes neglecting this rule but that are not part of the *GUI* component. This negative aspect of Hist-Inspect is also applicable to other existing anti-drift techniques. As a consequence, architects have to spend resources in the manual identification of anomalous GUI classes from those which are retrieved by the tool. This situation occurs given the inability of many anti-drift techniques for exploiting component properties in the source code (MACIA et al, 2012).

Listing 2

```
01:   <?xml version="1.0" enconding="UTF-8"?>
02:   <rule-catalog>
03:         <rule id="sampleRule"
04:               anomaly="unexpectedComplexGUI"
05:               expression="LOC >= 100 || CC >= 5"/>
06:   </rule-catalog>
07:   <anonaly-catalog>
08:         <anomaly id="unexpectedComplexGUI">
09:               applyTo="class"/>
10:   </anomaly-catalog>
```

To the best of our knowledge, state-of-art techniques for preventing architectural drift are limited to only identifying symptoms of architectural drift in system's implemented architecture. Consequently, developers can introduce unacceptable interactions (i.e., erosion symptoms) between components (Section 1.2). Moreover, these techniques usually only support the definition of anti-drift rules to all modules of a program as a whole. In other words, they often do not support the specification of rules to particular components, taking into consideration their properties and responsibilities. According to (MACIA et al, 2012), this inability impact the use of such techniques to detect relevant anomalous modules which are impairing the architecture modularity. Finally, these techniques support anti-drift rules 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 (MOHA et al, 2010) (Section 1.2).

2.5.3. Techniques for detecting both degradation symptoms

As far as we know, a few techniques can be adapted in order to partially support the simultaneous detection of erosion and drift symptoms (PMD, 2012; SEMMLE CODE, 2012). One of them is a recently-developed tool, called Semmle Code (SEMMLE CODE, 2012), which is based on a source code query language (VERBAERE et al, 2008). This tool allows architects to elaborate code queries taking into consideration several properties of source code elements such as method invocations, lines of codes and depth of inheritance tree. Hence, architects can define queries to check whether modules that are relevant to architecture (i.e., those which realize a component) violate any component or interaction constraints.

Listing 3 illustrates code queries that check the conformance of an anti-erosion rule (lines 01-11) and anti-drift one (lines 13-20) in the HealthWatcher architecture (Section 1.2). The former checks if there is a class that realizes the *Data* component and has any method invoked by a GUI module. The latter aims to detect anomalous GUI modules. Such modules may manifest drift symptoms. More specifically, it checks if a class from the package `GUI` has 100 or more lines of code and whose cyclomatic complexity value is greater or equals to 5. In particular, GUI modules should just delegate request to the Controller and, thereby, they use to have few lines of code.

Listing 3

```
01:  //R1: gui classes cannot directly access data classes
02:  from Class guiClass , Class dataClass, Method classMethod
03:  where (
04:          guiClass.getPackage ().getName ().
05:          matches( "%healthwatcher.gui%" ) and
06:          dataClass.getPackage ().getName ().
07:          matches( "%healthwatcher.data%" )
08:       )
09:  and guiClass.getACallable().calls( classMethod )
10:  and dataClass = classMethod. getDeclaringType ()
11:  select guiClass, dataClass, classMethod
12:
13:  //R2: GUI classes are expected to have low complexity
14:  from Package p, Class c , MetricCallable m
15:  where p.getName().matches("%healthwatcher.gui%")
16:  and c.getPackage() = p
17:  and m. getDeclaringType () = c
18:  and c.getLocation().getNumberOfLinesOfCode() >= 100
19:  and c.getCyclomaticComplexity()>= 5
20:  select c, c.getLocation().getNumberOfLinesOfCode()
```

We highlight two issues in Listing 3 regarding the detection of architectural erosion and drift symptoms. First, each query explicitly specifies the source code elements whose properties are being checked instead of specifying architectural components. For instance, the second rule (lines 13-20) checks the size and complexity of classes from the package `healthwatcher.gui` (line 15). All queries that refer to the GUI component (e.g., R1 - lines 01-11) replicate the same expression (line 15) to select the modules realizing the component. Hence, whenever the architectural component changes, all related queries require modifications. As an example, rename operations into the package `healthwatcher.gui` require modifications on all rules referring to the GUI component (e.g., R1 and R2 from Listing 3). The second issue is that code query languages often relies on syntaxes similar to SQL (TROPASHKO and BURLESON, 2007). This situation can restrict the use of these techniques to architects who have limited familiarity with query languages.