3
The TamDera

Recent studies (GARCIA et al, 2009; MACIA et al, 2012) have suggested that architectural erosion and drift processes are often interrelated. Erosion and drift symptoms tend to affect the same or somehow related modules in the source code (MACIA et al, 2012). These studies have also provided initial evidence that one or more drift symptoms tend to provoke the later introduction of erosion symptoms, and vice-versa (PERRY AND WOLF, 1992; GARCIA et al, 2009). However, techniques for preventing architectural degradation tend to focus on supporting the detection of either erosion or drift symptoms only (Section 2.5). Therefore, there is no explicit and unified support for developers to detect both kinds of degradation symptoms.

Moreover, given typical time constraints, developers are only encouraged to specify and maintain anti-degradation rules if they can reuse them in different circumstances. Ideally, architects should be able to reuse anti-drift and anti-erosion rules across projects adhering to similar architecture decompositions. Existing approaches only support the specification and checking of anti-degradation architecture rules for particular systems and do not provide any mechanism to reuse them (Chapter 2). 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 (MOHA et al, 2010).

This chapter systematically presents a domain-specific language, named **TamDera**[3] which enables the detection of both symptoms of architectural degradation in the source code (Section 3.1). We have implemented a prototype for supporting the use of **TamDera** (Section 3.2). The tool checks the architecture conformance of the software implementation with respect to anti-degradation rules. The goal is to provide instrumental support to the automatic detection of

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

both erosion and drift symptoms. Architects can use the tool to prevent architectural degradation and, hence, identify opportunities for architecturally-relevant refactorings.

3.1. The TamDera language

The **TamDera** language allows developers to define and blend anti-erosion and anti-drift rules to produce hybrid strategies for architectural degradation prevention. The hybrid specification of these rules might help to reveal or explain how an anti-erosion rule is leading to drift symptoms, or vice-versa (Chapter 4). In addition, we believe that, in certain circumstances, the description of anti-drift and anti-erosion rules should be naturally blended in architecture decisions and their specifications (Section 3.1.3). **TamDera** also supports the reuse of anti-degradation rules in multiple contexts (Section 3.1.4). Finally, Section 3.1.5 illustrates the use of **TamDera's** abstractions to specify hybrid rules to a design pattern. The following subsections describe the key abstractions of the **TamDera** language, while the appendix A presents the Backus Normal Form (BNF) grammar of the language.

3.1.1. Examples of anti-erosion and anti-drift rules

A set of anti-erosion (AER) and anti-drift (ADR) rules, taken from the motivating example (Section 1.2), is used to illustrate the main abstractions of **TamDera**. Each rule is represented by an acronym, which is used through the text. In particular, a design description of the GUI component (Figure 1) in natural language is provided. This description encompasses an anti-erosion rule (named AER1) and an anti-drift rule (named ADR1) which are strongly inter-related.

"*The GUI component 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 anti-erosion rule AER1 establishes that GUI classes cannot directly access services from modules realizing the Data component. The anti-drift rule ADR1 is intended to capture the GUI component's constraint of not realizing conceptually-different responsibilities. In order to represent this constraint, the rule ADR1 imposes upper boundaries on the size and cyclomatic complexity of GUI classes. The rules AER1 and ADR1 are combined to prevent the GUI component to assume more responsibilities than GUI-related ones.

In addition to AER1 and ADR1, we also use other two anti-erosion rules from the motivating example (Figure 1) in order to illustrate the use of certain **TamDera's** abstractions. They are listed in the following:

- AER2: Only classes realizing the Data component are able to handle Persistence and Transaction exceptions;
- AER3: Command classes realizing the component GUI must extend the `Abstract Command` class;

The anti-erosion rule AER2 is aimed at picking out the unacceptable handling of persistence-related and transaction-related exceptions by classes external to the *Data* component. Furthermore, AER3 is in charge of detecting absence violations (Section 2.5.1) of expected dependencies. In particular, it intends to enforce that all the Command classes extend a default abstract class. The reason is that the latter implements core functionalities to access requests and session objects.

3.1.2. The language overview

TamDera is different from classical architecture description languages (ADLs) (MEDVIDOVIC and RICHARD, 2000; GARLAN et al, 2007) as it is not intended to provide support for specifying component-and-connector decompositions (CLEMENTS et al, 2010). Instead, its goal is to support the specification of: (i) how certain architecturally-relevant concepts (e.g., a component) are realized by modules in the source code, and (ii) the rules governing the modules comprising those architectural concepts. Therefore, **TamDera** should be seen as complementary to ADLs and any other languages or notations for architecture documentation (CLEMENTS et al, 2010).

To achieve the aforementioned goal, **TamDera** provides two abstractions: *concept mapping* and *architectural concept*. They are the basic constructs in **TamDera** to specify rules, which are intended to detect both symptoms of architectural degradation. Figure 4 describes such **TamDera**'s constructs for architectural concept and concept mapping. The BNF description uses the bold font to display terminal symbols. These abstractions are presented and discussed in the following subsections.

```
ConceptDeclaration   ::=   concept ConceptId [ConceptInheritance]
                           { [ConceptMapping]}
ConceptMapping       ::=   name: STRING
                     ::=   parent: STRING
```

Figure 4. **TamDera's** constructions for architectural concept and mapping

3.1.2.1. Architectural concept

A key abstraction of **TamDera** is an *architectural concept*. Each concept represents a relevant concern to the mind-set of software architects. These concerns can be components, interfaces, or any other decision expressed in an architecture document, which is traceable to modules or inner module elements in a program. Concepts associated with design patterns, such as Façade and Chain of Responsibility (GAMMA et al, 1995), are also often critical to architectural decompositions. As a result, **TamDera** can also be used to express architecture rules associated with design pattern concepts.

Each architectural concept is reified by a set of module elements in the architecture's implementation. Module elements realizing a concept in a program can range from classes and interfaces to inner members of modules, such as methods. Architects rely on *architectural concepts* to describe anti-degradation rules that should be respected by aggregate sets of module elements realizing the concepts. The keyword **concept** is used to define an architectural concept, which is given a unique name (ConceptId), as described in Figure 4. Each concept is associated with a concept mapping. For illustration, Listing 4 shows a **TamDera** specification for three architectural concepts relevant to the rules ADR1, AER1, AER2 and AER3.

`GUIHW` (lines 01-03) and `DataHW` (lines 05-07) concepts refer to elements in the program that respectively realize the GUI and Data components, whereas

the `DataHWException` (lines 09-11) denotes specific exceptions pertaining to the Data component (Figure 1). The use of the keywords **name** and **parent** is explained in the next subsection.

Listing 4

```
01:  concept GUIHW
02:  { parent:"Command"|
06:
07:  concept DataHW
08:  { name: "healthwatcher.data.*"}
09:
10:  concept DataHWException
11:  { name: ".*PersistenceException|.*TransactException"}
```

3.1.2.2. Concept mapping specification

Architects define which module elements comprise each architectural concept through the TamDera's notion of *concept mapping specification*. 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.

Name-based Mapping. The definition of common properties governing element names is made using the keyword **name** (Figure 4). The name-based mapping receives a string (i.e., a regular expression) as input and retrieves all source code elements, whose names match it. Regular expressions provide special characters that allow the explicit matching with the names of packages, classes, interfaces and methods. More specifically, the character '|' provides alternative expressions for matching the input string. The character '.' matches any character and the quantifier '*' is used for matching zero or more occurrences of the precedent character.

Listing 4 illustrates the use of the name-based mapping. The concept `DataHWException` is mapped either to classes whose names ends with `PersistenceException` or `TransactException` through name-based concept mapping (Listing 4 - line 11). On the other hand, the concept `DataHW` is

associated with all the code elements included in the package `healthwatcher.data` and in its sub-packages.

Parent-based Mapping. Parent-based mapping is denoted by the keyword **parent** (Figure 4). This pronoun refers to a super class or an interface extended by the code elements realizing the concept. Indeed, Cruz et al motivates the use of pronouns to refer to a set of architecturally-relevant elements or design pattern elements (CRUZ and LUCENA, 2003). However, Cruz et al (CRUZ and LUCENA, 2003) were interested in supporting such pronouns in a programming language rather than a design language to support architecture conformance as in our case.

Listing 4 also depicts an example of how to use the keyword **parent**. The concept `GUIHW` is mapped to all classes whose parent class is named `Command`. Parent-based mapping is particularly interesting for programs with stable interfaces. These stable interfaces in the implementation are typically associated with architecturally-relevant interfaces (e.g., documented in component-and-connector models (CLEMENTS et al, 2010)). For instance, design patterns, such as Chain of Responsibility (GAMMA et al, 1995), often rely on interfaces to structure their solutions; they are also used to realize architecturally-relevant component interfaces, to which certain rules need to be defined.

3.1.3. Blending anti-erosion and anti-drift rules

TamDera allows architects to specify anti-drift rules and anti-erosion rules in terms of architectural concepts and their interactions. Anti-erosion rules refer to concepts by their names in specific declarative statements. They describe unexpected and mandatory interactions between the elements comprising two or more concepts. While anti-erosion rules define concept interaction constraints, anti-drift rules define individual concept constraints. In other words, anti-drift rules are defined to a particular architectural concept. They establish user-defined boundaries (or thresholds) on the structural properties of the implementation elements composing the respective concept. The **TamDera**'s mechanisms for describing both forms of rules are described in the following subsections.

3.1.3.1. Anti-erosion rules

Figure 5 illustrates the TamDera's constructions for anti-erosion rules. Each rule is formed by at least two architectural concepts, a *source* and a *target*, whose interactions between them are constrained. The former encompasses code elements that are the source of an established dependency. The latter are the target concept's elements whose dependencies with the source elements are constrained. The anti-erosion rules refer to definitions of the involved architectural concepts through their unique names (ConceptId).

```
AntiErosionRule   ::=   only ConceptList can-DependType ConceptList
                  ::=   ConceptList cannot- DependType ConceptList
                  ::=   ConceptList must-DependType ConceptList
ConceptList       ::=   ConceptId ( , ConceptId)*
```

Figure 5. TamDera 's constructions for anti-erosion rules

Thus, architects can establish constraints related to expected, unexpected and mandatory dependencies between sets of architectural concepts. Table 1 summarizes the dependency types currently supported by TamDera; *A* indicates a set of source concepts and *B* refers to target ones. If there are multiple concepts in both sets A and B, it means that all the elements of concepts in A must follow the dependency constraint with respect to the elements of concepts in B. Even though the same source code element is mapped to a concept from A and to another concept from B, it must follow the established dependency constraints between these concepts. In other words, a code element mapped to more than one concept must satisfies the rules associated with those concepts.

Some dependency types are not applicable to certain type of module elements. For instance, ***declare*** and ***derive*** are only applicable to classes.

Table 1. Dependency types supported by TamDera

Dependency	Description
A ***invoke*** *B*	A method of *A* calls a method of *B*
A ***create*** *B*	Some method of *A* creates an object instance of a class of *B*
A ***declare*** *B*	The type of a field variable of *A* is a class of *B*

A ***derive*** *B*	A class of *A* extends or implements a class of *B*
A ***handle*** *B*	A code element of *A* has a catch block that handles any class exception of *B*
A ***depend*** *B*	A code element of *A* has some kind of dependency on a code element of *B*

TamDera provides three constructions to establish anti-erosion rules: **cannot**, **only-can** and **must** (Figure 5). They are defined in terms of two architectural concept lists (ConceptList), which denote respectively the set of source architectural concepts (source concepts) and the target architectural concepts (target concepts).

The construction **cannot** establish that source concept elements are prohibited to have a specific dependency type with any target concept element. Listing 5 illustrates the rule AER1 (line 01) from the HealthWatcher architecture (Figure 1). It uses the construct **cannot** and the dependency type ***invoke*** to prohibit the access from `GUHWI` (source concept) code elements to services provided by `DataHW` (target concept) elements.

The construction **only-can** (line 03) establishes that only code elements from the source concepts can have a specific type of dependency with code elements from the target ones. As a consequence, it can be used to restrict the access to target concept services exclusively to the source concepts. In addition, it can be used to ensure that certain architecturally-relevant dependencies governing exception flows, i.e., where certain raised exceptions should be handled. For instance, the second rule from Listing 5 (line 03) verifies whether there is a module which does not realize the Data layer (i.e., `DataHW`), but it is undesirably handling (i.e., catching) an exception comprised by the concept `DataHWException` (AER2).

Listing 5

```
01:     GUIHW cannot-invoke DataHW
02:
03:     only DataHW can-handle DataHWException
04:
05:     GUIHW must-derive AbstractCommand
```

The construction **must** imposes that source concept elements must have a specific dependence type with the target concept elements. Thus, the last rule

from Listing 5 uses the dependency *derives* to enforce GUI command classes to extend a default abstract command class (AER3).

3.1.3.2. Anti-drift rules

TamDera allows architects to define strategies (MARINESCU, 2004) for detecting architectural drift in the form of anti-drift rules. These rules are defined as part of architectural concept bodies (Figure 6) enclosed by curly braces. These rules are composed by a metric, a mathematical operator and a value. More specifically, we use a mathematical operator to bind a threshold value to a quality metric. These metrics are used to capture component's structural constraints, and are intended to capture possible deviations from modularity principles (Section 2.3).

ConceptDeclaration	::=	**concept** ConceptId **{** (AntiDriftRule)* **}**
AntiDriftRule	::=	Metric Operator Value
	::=	ConstraintSetDecl
Metric	::=	**LOC** | **CBO** | **NOP**| **CC** | **DIT** | **...**
Operator	::=	**>** | **<** | **=** | **≤** | **≥**
ConstraintSetDecl	::=	**constraintset** ConstraintSetId **{** (AntiDriftRule) +**}**
Value ∈ NUMBER		

LOC = lines of code, CBO = coupling between object classes; CC = cyclomatic complexity; NOP = number of parameters; DIT = depth of inheritance tree

Figure 6. **TamDera's** constructions for anti-drift rules

These structural metrics are the most popular strategy to detect drift symptoms (MARINESCU, 2004). The association of these metrics-based rules with architectural concepts enables developers to ensure that architecturally-relevant modules in the code – i.e., those comprising an architectural concept – are free from architectural drift symptoms. Existing metrics-based techniques for detecting code anomalies do not allow architects to explicitly segregate anomalous code elements that are directly relevant to the architecture decomposition (Section 2.3). To overcome this limitation, **TamDera** allows

architects to impose boundaries (i.e., thresholds) on structural properties, such as size and coupling, of the source elements mapped to the concept. **TamDera** currently supports classical metrics for size, complexity, cohesion and coupling. The language can be extended to support new metrics (Section 3.2).

For illustration, Listing 6 shows rules 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 user-defined thresholds (i.e., 100 and 5). 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 1.2). The violation of drift rules may imply that developers need to reason about producing and checking anti-erosion rules (Section 4.4).

Listing 6

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

3.1.4. Reusing anti-degradation rules

TamDera offers a compositional reuse mechanism for anti-drift rules (Section 3.1.4.1) as well as a hierarchical reuse mechanism of architectural concepts (Section 3.1.4.2), 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.

3.1.4.1. Compositional reuse

Compositional reuse enables grouping anti-drift rules into a named set (constraint set). Its rationale is that most of drift symptoms are indicated by the neglection of several common and reusable expected property constraints of components. Architects use the keyword **constraintset** (Figure 6) to specify a set

of reusable anti-drift rules in **TamDera**. The definition of an architectural concept can refer to a constraint set to reuse its anti-drift rules. Hence, the set of anti-drift rules of an architectural concept includes: (i) the rules explicitly defined within its body (Section 3.1.2.1), and (ii) those associated with the referenced constraint sets.

Listing 7 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 to avoid that a piece of coherent functionality get artificially decomposed in several hierarchy classes. The constraint set is reused in the `GUI` and `Controller` concept definitions (lines 05, 09 respectively). For conciseness, we omitted the `GUI` and `Controller` anti-drift rules, which are composed with the `InheritanceOveruse` rules.

Listing 7

```
01: constraintset InheritanceOveruse {
02:  DIT < 5
03: }
04:
05: concept GUI{
06:  InheritanceOveruse
07: }
08:
09: concept Controller{
10:  InheritanceOveruse
11: }
```


3.1.4.2. Hierarchical reuse

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.

TamDera supports the reuse of anti-drift rules from a concept (super concept) to another concept (sub-concept). The inheritance of architectural concepts is similar to inheritance mechanisms in object-oriented systems.

Architects use the keyword **extends** in the declaration of a concept to establish an inheritance relationship with a super concept (Figure 7). This construction receives a string as parameter (ConceptId) which identifies the super concept.

ConceptDeclaration ::= **concept** ConceptId [ConceptInheritance]
{ [ConceptMapping] (AntiDriftRule)* **}**

ConceptInheritance ::= **extends** ConceptId

Figure 7. Constructions for concept inheritance

Figure 8 presents the definition of a super concept GUI (on the top of the figure). It defines three anti-drift rules (R1, R2, and R3). They are in charge of realizing the rule ADR. This concept is extended by `ViewMM` (on the left of the figure - lines 02-07), which implicitly inherits all `GUI` rules trough the inheritance mechanism. Therefore, all module elements mapped to `ViewMM` must satisfy these rules.

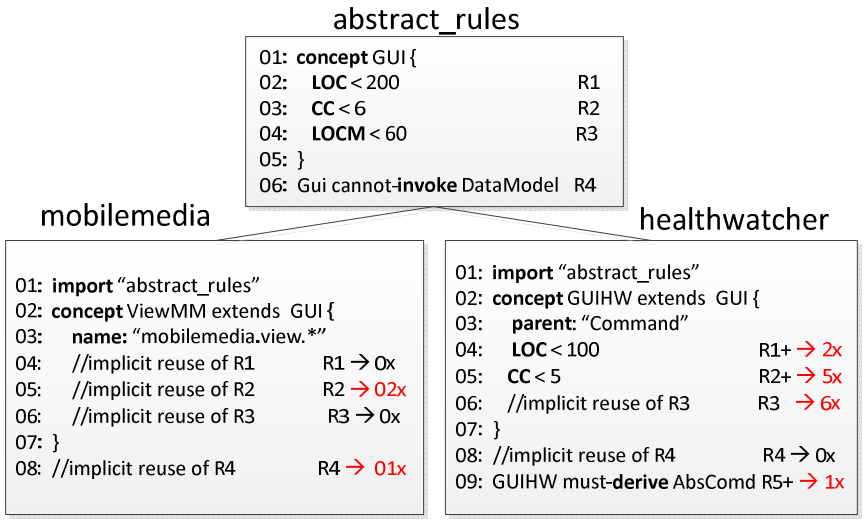

Figure 8. Reuse of anti-degradation rules using TamDera

Abstract concepts. TamDera concepts can be abstract or concrete. Unlike concrete concepts, abstract concepts do not specify a concept mapping (Section 3.1.2.2). For instance, the abstract concept `GUI` (Figure 8) has two concrete sub-concepts (`ViewMM` - left and `GUIHW` - right) that define a concept mapping (line 3). For evident reasons, only concrete concepts are checked during the anti-degradation rule conformance. This will be explained in Section 3.2.2.

Abstract anti-drift rules. **TamDera** supports the definition of abstract anti-drift rules. In particular, users can define these rules using threshold variables

instead of concrete values (Figure 9). In a concept body, architects declare a list of threshold variables (ThresholdVariableList) using the keyword **thresholds**. Also, they can assign numeric values for these variables in sub-concepts of the super concept. More specifically, TamDera has the keyword **assign** which receives an identifier of a variable as input, and assigns a numeric value to the variable. This construct is particularly interesting as architects can define reusable anti-drift rules without necessarily assigning their values. Also, this is often the case when systems are in the first phases of software development and hence, architects do not have enough knowledge to impose implementation constrains of components. In addition, it can also be interesting to the definition of program family architects, where certain threshold values can be specific to certain variants of a family.

```
ConceptDeclaration    ::=   concept ConceptId [ConceptInheritance]
                            {
                              [ThresholdVariableList]
                              (AntiDriftRule)*
                              (AssignmentThreshold)*
                            }
AntiDriftRule         ::=   Metric Operator (Value | VariableId)
                      ::=   ConstraintSetId
ThresholdVariableList ::=   thresholds: VariableId ( , VariableId )*

AssignmentThreshold   ::=   assign VariableId to VALUE

VariableId            ::=   STRING
Value                 ::=   NUMBER
```

Figure 9. TamDera 's constructions for abstract anti-drift rules

Listing 8 illustrates the same `GUI` concept from Listing 6 using abstract anti-drift rules. It defines an abstract concept named GUI (lines 01-06) and declares two threshold variables: `LOW_SIZE` and `LOW_COMPLEXITY` (line 03). They have their values assigned in the sub-concept `GUIHW`. They refer to the rule ADR1 (Section 3.1.1) which assigns 100 for the variable `LOW_SIZE` and 5 to the `LOW_COMPLEXITY`.

Listing 8

```
01:  concept GUI
02:  {
03:    thresholds: LOW_SIZE, LOW_COMPLEXITY
04:    LOC < LOW_SIZE
05:    CC < LOW_COMPLEXITY
06:  }
07:
08:  concept GUIHW extends GUI
09:  { parent:"Command"
10:    assign LOW_SIZE to 100
11:    assign LOW_COMPLEXITY to 5
12:  }
```

Architectural models. TamDera allows users to modularize the specification of concepts and anti-degradation rules in several *architectural models*. Figure 8 presents three such models: *abstract_rules* (top), *mobilemedia* (left) and *healthwatcher* (right). The model *abstract_rules* defines an abstract concept `GUI` (lines 01-05) and an anti-erosion rule that checks the conformance of AER1 (Section 3.1.1). The other models specify rules to respectively constrain the architecture of HealthWatcher and MobileMedia.

The same architectural model can be used in several projects, thus promoting the reuse of both anti-erosion and drift rules across multiple projects (Section 1.2). 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. This construct 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.

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. As an example, Figure 8 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 `AbsComd`.

3.1.5. TamDera specification example

This section illustrates the application of **TamDera** to elaborate hybrid rules related to an architecturally-relevant design pattern (Section 2.3). We selected the Mediator design pattern because its description (GAMMA et al, 1995) encompasses constraints on both components and their interactions. These constraints respectively require the definition of anti-drift and anti-erosion rules for architectures realizing the Mediator pattern. Here, we focus on a subset of constraints related to the Mediator pattern, which are retrieved from the pattern design description (GAMMA et al, 1995):

"The Mediator design pattern intends to promote loose coupling by defining a specific module that encapsulates how a set of other modules interact. More specifically, the term Mediator denotes the module responsible for encapsulating the interactions among a group of particular modules. This group encompasses modules which are named Colleagues. Hence, the colleagues send and receive requests from the mediator while the responsibility of the mediator is the implementation of cooperative behavior by coordinating requests between appropriate colleague(s)".

Anti-drift rules. According to the description, the Mediator pattern intends to promote loose coupling between colleagues. In an architecture description, the colleagues can be realized, for instance, by implementation modules in different components in order to decouple them from each other. According to the pattern description, we should define an anti-drift rule to impose an upper boundary to the coupling of modules realizing colleagues. Otherwise, the pattern applicability would not be worth.

In addition, we should specify an anti-drift rule to establish a lower boundary to the coupling of mediators. This rule aims to avoid the pattern usage to decouple a few colleagues (i.e., assuming the coupling among the mediator and colleagues). In such scenario, the use of the pattern may not entail benefits. For instance, it would not be worthwhile to reduce the coupling between two or three colleagues. Finally, we also define rules to constrain the size and complexity of mediators aiming at avoiding anomalous God classes (FOWLER, 1999), such as the `BaseController` in `MobileMedia` architecture (Section 1.2).

Anti-erosion rules. The main responsibility of the Mediator is to coordinate requests between appropriate colleagues. Hence, we establish two anti-erosion rules according to the interaction constraints between the mediator and colleagues: (i) colleagues must access services from mediator to send their requests to other colleague and, as a consequence, (ii) colleagues cannot directly access services from other colleagues.

TamDera enables the specification of hybrid rules to the Mediator design pattern. Listing 9 presents each concept definition and the two anti-erosion rules. We use abstract concepts (Section 3.1.4.2) to denote the mediator and colleagues. Hence, the concepts `Mediator` and `Colleague` do not define a concept mapping (Section 3.1.2.2). The idea is to provide a reusable specification of hybrid rules for the Mediator pattern.

The concept `Mediator` (lines 01-07) establishes three anti-drift rules. First, it establishes a minimum value of coupling to a mediator module (line 04) to avoid the cases of mediators coupled with a few colleagues. Second, the concept constrains the size of the enclosed elements (line 5). Third, it also constrains their complexity (line 6). These two rules are tailored for detecting mediators which may implement several responsibilities. On the other hand, the concept `Colleague` constrains only the coupling of the colleague modules (line 12). Regarding the pattern intention, they must be decoupled through the mediator. As it can be noticed, each anti-drift rule in the concept definitions is defined using thresholds variables (Section 3.1.4.2).

Listing 9

```
01:  concept Mediator
02:  {
03:       thresholds: LOW_COUPLING, HIGH_SIZE, HIGH_CC
04:       CBC > LOW_COUPLING
05:       LOC < HIGH_SIZE
06:       CC < HIGH_COMPLEXITY
07   }
08:
09:  concept Colleague
10:  {
11:       thresholds: LOW_COUPLING
12:       CBC < LOW_COUPLING
13:  }
14:
15:  Colleague must-invoke Mediator
16:  Colleague cannot-invoke Colleague
```

Listing 9 also depicts two anti-erosion rules. First, it establishes that colleagues must have a invoke dependency with the mediator (line 15). This situation occurs as colleagues must invoke services provided by the mediator to send request to another colleague. The second rule is particularly interesting as it involves the same concept playing both roles of source and target of a given established dependency constraint. It prohibits colleagues to directly invoke services from each other (line 16). It is important to highlight that dependencies are defined in term of different modules (Section 2.1.1). Otherwise, this rule would incorrectly detect colleagues who invoke any of their own methods.

3.2. The TamDera tool

This section outlines key issues on the implementation of the **TamDera** tool. The tool design was driven by the goal of maximizing the reuse of anti-erosion and anti-drift tools whenever it was possible (Section 3.2.1). Section 3.2.2 presents how the **TamDera** tool detects architectural degradation symptoms. Section 3.2.3 depicts how the **TamDera** tool identifies inconsistencies among rules (e.g., contradictory dependency constraint) in architecture models (Section 3.1.4.2).

3.2.1. Tool design

The **TamDera** language (Section 3.1) allows the specification of anti-degradation rules in software systems. In the first version of the **TamDera** tool, we decided to support architecture enforcement of Java programs. The goal was to evaluate the feasibility of our approach in the context of programs implemented with a popular programming language. This strategy also allowed us to reuse static analysis platforms for Java programs (EICHBERG et al, 2008; TERRA and VALENTE, 2009). In addition, we implemented the tool upon the Eclipse platform using XText (XTEXT, 2012) which is a framework for developing the parser and editor. XText provides features, such as syntax coloring and code completion which are interesting to reduce the effort to specify concepts and anti-degradation rules.

Design overview. A simplified view of the tool design is composed of four components: Controller, Concept Mapper, Consistency Checker and Rule Translator (Figure 10). Each of them has a particular responsibility with respect to the detection of erosion and drift symptoms. The Controller component coordinates the detection of degradation symptoms in a system implementation. It receives some inputs (e.g., architecture models) and delegate activities to other internal or external components to perform the architectural conformance (Section 3.2.2). Concept Mapper is responsible for evaluating the expressions related to concept mappings (Section 3.1.2.2). The component Consistency Checker is tailored for verifying certain inconsistencies in the specification of anti-erosion rules (Section 3.2.3). Finally, the Rule Translator takes syntactic nodes representing anti-erosion or anti-drift rules and translates them to Prolog queries.

The tool uses the Prolog engine (SWI-PROLOG, 2012) to statically check the conformance of anti-degradation rules. Structural properties of module elements are stored as logic statements (i.e., knowledge base) (CERI, 1989). The tool also stores dependencies between code elements and several metric values. The set of code dependencies encompasses all those required to describe anti-erosion rules (Section 3.1.3.1), such as method invocation, class inheritance, exception handling block (i.e., catch), and field declaration. Thus, a unique Prolog-based representation is used for detecting erosion and drift symptoms.

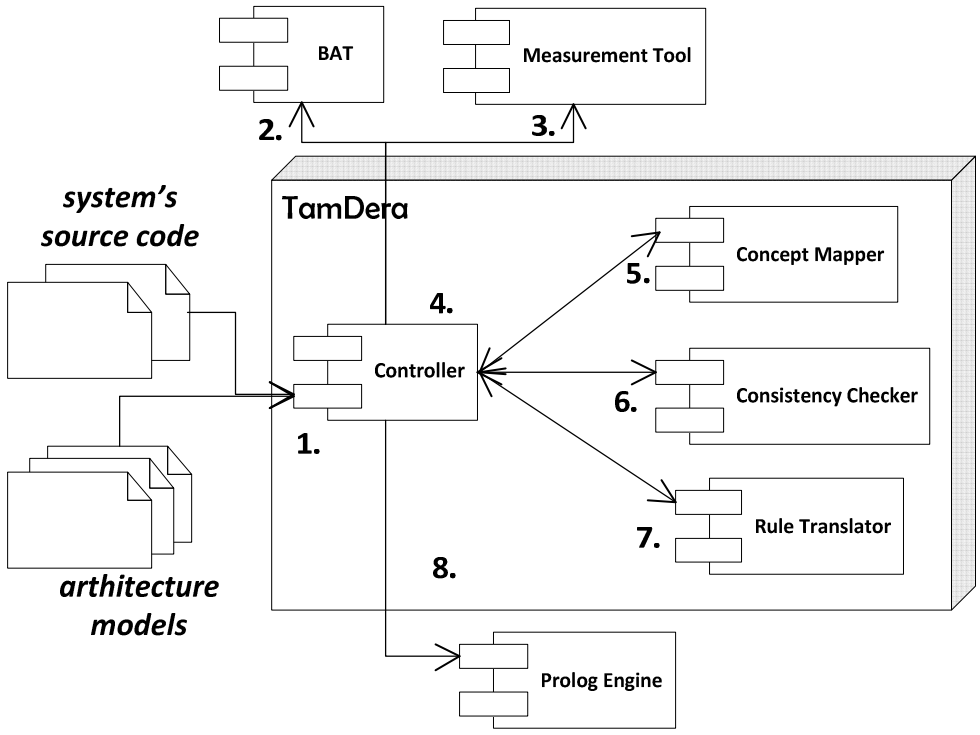

Figure 10. Simplified design of TamDera's tool

This single representation can also facilitate the tool extension to other programming languages as we can develop translators of programs in these languages to Prolog statements. This characteristic is particularly interesting as recent studies have shown that most software projects nowadays are implemented in four different programming languages (UBAYASHI et al, 2010). Finally, a recent research (EICHBERG et al, 2008) has provided evidence suggesting that it is reasonably efficient to use Prolog as an engine to detect erosion symptoms into the incremental build process of large systems. This means that the Prolog engine is fast enough to perform architectural conformance after the build process. This strategy allows the system to be continuously checked when the source code is modified (EICHBERG et al, 2008).

Figure 10 illustrates the design of the **TamDera** tool. The inputs to **TamDera** consist of two basic artifacts (*step 1)*: the system's source code and **TamDera** architecture models (Section 3.1.4.2). The former is used by **TamDera** for obtaining structural information about the system implementation, i.e., the program modules, their inner members, and their dependencies. More specifically, the tool reuses the Bytecode Analysis Toolkit (BAT) (EICHBERG et al, 2008),

which receives the system binaries (obtained from its source code) as parameter and retrieves a Prolog-based representation of the system (*step 2*). Then, this representation stores the information about modules, their members, and their dependencies. The representation enables to check all dependency types supported by **TamDera** (Section 3.1.3.1).

The tool also uses the system's source code to gather several measurements for module properties, such as size and coupling used for describing anti-drift rules (Section 3.1.3.2). In particular, we use the output file generated by Together (TOGETHER, 2012) to obtain and store these measures (*step 3*). However, any other measurement tool could be used by **TamDera**. For instance, it is straightforward to integrate our tool with all the anti-drift tools mentioned in Section 2.5.2.

The tool receives the system architecture model as input. It parses the main architectural model and also the ones which are referred by the former through the use of the **import** keyword (Section 3.1.4.2). Hence, the component `Controller` (Figure 10 - *step 3*) evaluates all anti-degradation rules taking into consideration the use of reuse mechanisms (Section 3.1.4). Also, it evaluates the concept mappings (*step 5*) and stores them in the knowledge base through the component `Concept Mapper` (Figure 10 - *step 4*). Our engine uses this information to identify concept's code elements that violate one or more anti-degradation rules. However, before checking the rule conformance, the tool verifies whether the set of anti-degradation rules is consistent (*step 6*). Section 3.2.2 describes this procedure in more detail. The component `Rule Translator` takes the anti-degradation rules and translate them to Prolog queries (*step 7*). Then, the `Controller`: (i) takes these queries and the knowledge base (i.e., dependencies and metric values for each code module), and (ii) executes the Prolog Engine to check the conformance of anti-degradation rules (*step 8*).

3.2.2. Detecting degradation symptoms

TamDera tool is used for preventing architectural degradation through the detection of erosion and drift symptoms. In particular, it supports the detection of rule violations by transforming them into Prolog queries (Section 3.2.1). This

section details this process by exemplifying the transformation of the rules AER1 and ADR1, presented in Section 3.2.1.

Concept mapping. The tool parses all concept definitions and evaluates their mappings. Listing 10 presents the mappings of the concepts `GUIHW` and `Data`. As it can be noticed, we define Prolog facts (CERI, 1989), named `conceptMapping`, that are in charge of representing the connection between the source code elements and each concept. For instance, the class `Login` from the package `hw/view` is mapped to the concept `GUIHW`. Then, we translate anti-erosion and anti-drift rules to Prolog queries, the so-called *anti-erosion* and *anti-drift queries*.

Listing 10

```
01:   conceptMapping('GUIHW',class('hw/view, Command')).
02:   conceptMapping('GUIHW',class('class(hw/view, Login')).
03:   conceptMapping('GUIHW',class('class(hw/view, InsertComplaint')).
04:   ...
05:   conceptMapping('Data',class('hw/data','IRepository')).
06:   conceptMapping('Data',class('hw/data','ISymptomRepository')).
07:   conceptMapping('Data',class('hw/data','ComplaintRepository')).
08:   conceptMapping('Data',class('hw/data','DiseaseRepository')).
09:   ...
```

Checking anti-erosion rules. The generated Prolog queries use functions that were defined in particular Prolog files in the TamDera tool. In a nutshell, anti-erosion queries search for source code elements from the source and target concepts (Section 3.1.3.1) through the `conceptMapping` statement. Also, they verify the presence or absence of a specific dependency between the elements from the source and target concepts. For instance, Listing 11 depicts the anti-erosion query for rule AER1 (line 01). As we can notice, the string 'invoke' in the function `cannot_invoke` refers to the corresponding dependency type used in the anti-erosion rule.

Listing 11 also illustrates the definition of a function (lines 05-08). It traverses the knowledge base looking for source code elements from the source and target concepts (Section 3.1.3.1). These code elements are respectively named `SOURCE` and `TARGET`. Then, the function checks the existence of accidental method invocations between these code elements. For illustrative purpose, we removed other parameters from these functions which are used to represent

detailed information (e.g., method signatures). Finally, the tool dynamically maps the dependency types to previously-defined functions (e.g., `cannot-invoke` - line 03). As a consequence, we can extend the set of dependency types through the addition of the respective Prolog functions.

Listing 11

```
01:  GUIHW cannot-invoke Data.
02:
03:  cannot_invoke('GUIHW','Data', SOURCE, TARGET).
04:
05:  cannot_invoke( C1, C2, SOURCE, TARGET) :-
06:       conceptMapping(C1, SOURCE),
07:       conceptMapping(C2,TARGET),
08:       method_invocation(SOURCE,TARGET).
```

Checking anti-drift rules. Anti-drift queries rely on the verification of measurement results against the established boundary constraints (i.e., the metric thresholds). They use the same `conceptMapping` for identifying code elements from the component which is being constrained. Listing 12 illustrates the evaluation of an anti-drift query to check the rule ADR1 (line 01), presented in Section 3.1.1. The **TamDera** tool stores the result of a particular metric for each source code element in a prolog file. The statement property stores the measurement results and receives the name of a metric, the enclosed concept and the measure value. For instance, Listing 12 presents stored information about `SearchComplData's` (Section 1.2) size and complexity (lines 03-05). These queries also use defined functions that compare stored properties with desirable thresholds. For instance, the function `less_than` checks if all elements mapped to `GUI` have less than 100 lines of code (ADR1). This function is instantiated in order to perform the anti-drift queries. As we can notice, we verify for the GUI concept, if there is any element violating the thresholds establishes for lines of code or complexity (lines 09-10). If so, they are retrieved by the engine. Queries for anti-drift rules refer to measures for each code element, and these measures are imported by **TamDera**. As a result, the set of metrics available for anti-drift rules can be extended considering the new imported metrics.

Listing 12

```
01:   concept GUIHW { LOC < 100; CC < 5 }
02:
03:   property(
04:         'LOC',class('hw/view','SearchComplData'), 200.0).
05:   property('CC',class('hw/view','SearchComplData'),12.0).
06:
07:   less_than('LOC','GUIHW',100,CLASSNAME,MEASURE).
08:   less_than('CC','GUIHW',5,CLASSNAME,MEASURE).
10:   less_than(PROPERTY, CONCEPT, THRESHOLDE):-
11:         conceptMapping(CONCEPT, ELEMENT),
12:         property(PROPERTY, ELEMENT,MEASURE),
13:         MEASURE >= THRESHOLD.
```

3.2.3. Checking inconsistencies among rules

The tool also checks the consistency of anti-degradation rules defined in architectural models (Section 3.1.4.2). In fact, users can unconsciously define an inconsistent set of rules which hinder the architectural conformance checking (EICHBERG et al, 2008; TERRA and VALENTE, 2009). For instance, Listing 13 presents two cases of inconsistent rules. They impose contradictory constraints to the implemented architecture. In particular, the first rule (line 01) establishes that elements from A must invoke services from B. The second rule is inconsistent with the first one since it prohibits the access of B services by modules of A. In order to address this inconsistency case, we use a Prolog query to check if there are two anti-erosion rules that refer to the same concepts (A and B) and interaction types (invoke) but one uses a **must** relationship while the other uses a **cannot** one.

As a first step, the tool takes each anti-erosion rule and stores its source concepts, target concepts, the dependency types (i.e., must, cannot and only-can) and the kind of the rule in the knowledge base. Listing 13 also presents other rule inconsistencies (lines 04-05). It establishes that only elements from concept A are able to declare variables of type C. The last rule imposes that only elements of B can declare variables of type C. Similarly to the first case, **TamDera** also detects this case by elaborating queries that check **only-can** definitions, which denote the same target concepts (e.g., C) and dependency types (declare), but differ from the source concept (e.g., A and B).

Listing 13

```
01:  A must-invoke B
02:  A cannot-invoke B
03:
04:  only A can-declare  C
05:  only B can-declare C
```