4
Concern-Driven Metrics

In the first chapter we identified a set of limitations that hinders the evaluation of modularity by means of conventional metrics. These limitations are caused by the fact that the metrics currently used for evaluating design modularity mainly rely on abstractions, such as components, classes, and aspects, derived from the syntax of either programming or design specification languages. As a consequence, existing measurement approaches disregard the concerns that drive the system design.

This chapter and Chapter 5 present our proposed measurement approach for closing this gap between quantitative modularity assessment and the concerns that drive the design. The approach is based on concern-driven metrics for assessing architecture and detailed design. This chapter defines our suite of concern-driven architecture metrics. It starts introducing the notion of concerns and the categories of concerns we consider in our approach (Section 4.1). Section 4.2 depicts the model of concern representation upon which the metrics are defined. Section 4.3 defines our suite of concern-driven architecture metrics. Section 4.4 complements the definition of the metrics in the light of a theoretical measurement framework tailored for concern-oriented metrics. Finally, Section 4.5 discusses the interplay of our proposal and related work.

4.1.
Classification of Software Concerns

The term concern has been loosely defined as any property or part of the problem that stakeholders of a software system want to consider as a conceptual unit and treat in a modular way (Elrad et al., 2001; Tarr et al., 1999; Robillard & Murphy, 2007). Concerns can range from high-level notions like security and quality of service to low-level notions such as caching and buffering (Elrad et al., 2001). They can be functional, like business rules or features, or non-functional, such as synchronization and transaction management (Elrad et al., 2001). Typical

concerns in software systems capture (Elrad et al., 2001; Robillard & Murphy 2007; Hannemann & Kickzales, 2002):

- features from a feature list of a product line;
- functional requirements from a requirements specification document;
- non-functional requirements from a requirements specification document;
- roles from architectural patterns (e.g. the view role from the model-view-controller pattern (Buschmann et al. 1996));
- roles from design patterns (e.g. the subject role from the Observer design pattern (Gama et al., 1995)); and
- implementation mechanisms (e.g. caching).

In fact, this list is not exhaustive. The design of a system encompasses several concerns which come directly from the requirements specification or emerge during the architecture or detailed design conception. In the example of the Health Watcher architecture (Figure 8)[2], there are a number of architecturally-relevant concerns such as GUI, Business, Distribution, Persistence and Exception Handling. As mentioned before, in a software design, a concern is realized by a set of design elements. In an architectural design a concern can be addressed, for instance, by components, interfaces and operations. In an aspect-oriented detailed design, a concern can be realized by classes, aspects, methods, attributes, pieces of advice and intertype declarations.

Our measurement approach focuses on concerns that eventually evolve into concrete pieces of code and contribute directly to the functionality of the system. Persistence and exception handling are examples of concerns that evolve into pieces of code. Our metrics do not rely on concerns that influence how the system is built but do not trace to any specific piece of code. In particular, we do not focus on concerns that are not observable when the system executes, such as maintainability.

[2] Figure 8 is the same as Figure 1 (Section 1.2). We repeat it here to facilitate the reader to refer to it.

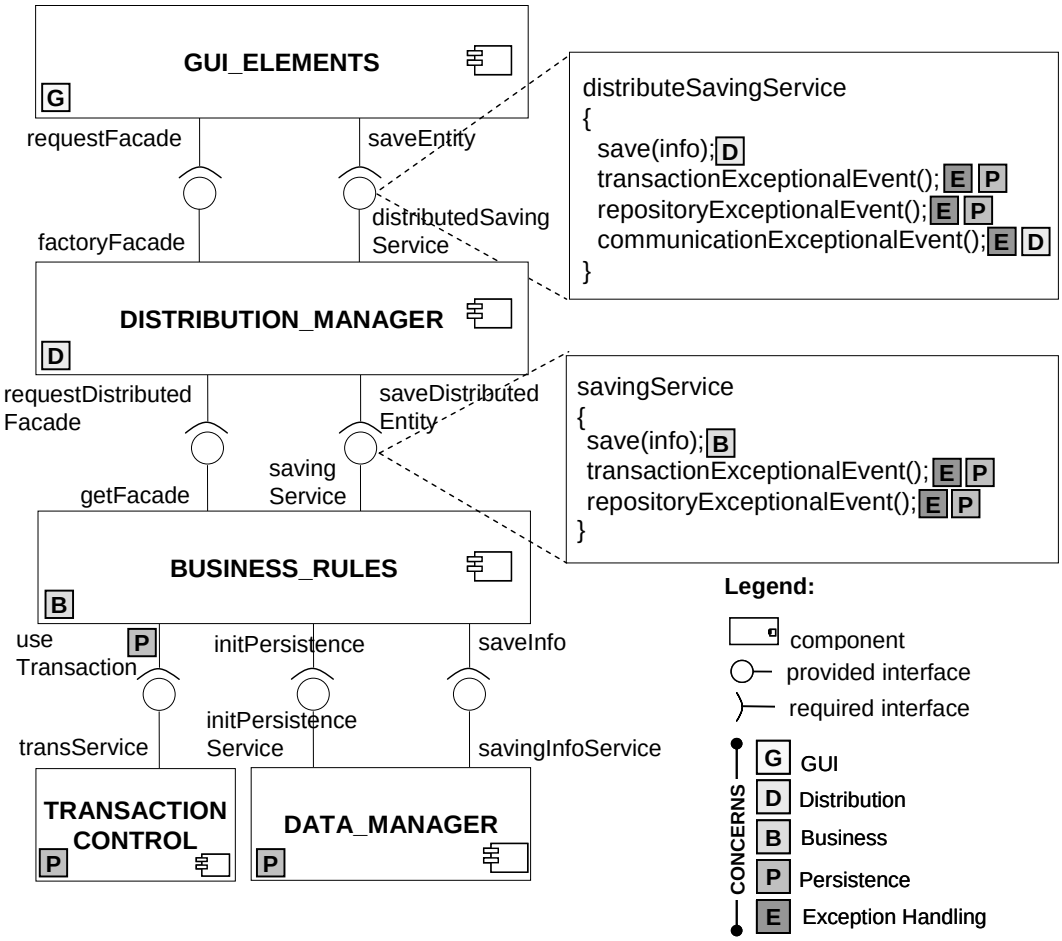

Figure 8: Simplified representation of the Health Watcher software architecture

4.2. Concern Representation

Our measurement approach is based on a *concern-to-design mapping*. This means that we have two domains related to each other through a mapping relationship. The source domain is a set of concerns and the target domain is a set of design elements, as illustrated in Figure 9. The mapping consists of assigning a concern to the corresponding design elements that realize it. For instance, in the architecture of Figure 8, the persistence concern is realized by the following elements: the Data_Manager and Transaction_Control components and their interfaces; the useTransaction required interface; and the transactionExceptionalEvent and repositoryExceptionalEvent operations. The last two elements represent persistence-specific exceptional events.

A systematic concern mapping process for consistently identifying the design elements realizing each concern is essential for the success of the proposed

measurement approach. In our empirical studies (Chapters 7 and 8), we followed a specific guideline in order to systematically map concerns to design elements. This guideline, which is inspired on the guidelines proposed by Eaddy et al (2007), states that a concern should be assigned to a design element if the complete removal of the concern requires with certainty the removal or modification of the element. In Figure 8, for instance, the useTransaction required interface in the Business_Rules component only encompasses three operations, namely beginTransaction(), commitTransaction() and rollbackTransaction() (not shown in the figure). These operations are totally dedicated to invoke transaction control services for persisting information. Therefore, removing the persistence concern would lead to the removal of the useTransaction interface. Thus, according to the aforementioned guideline, the persistence concern should be assigned to this interface. Section 7.1 describes other measures we took during our empirical studies in order to make the concern mapping sufficiently systematic for the goals of our studies.

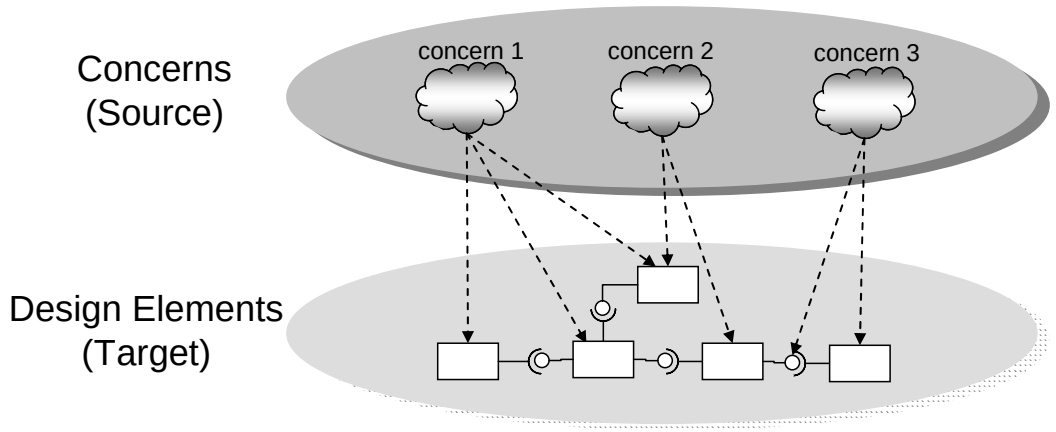

Figure 9: Mapping between concerns and design elements

The notion of *concern representation* (or simply, concern) is, therefore, linked to an underlying design model. The design model specifies which information about a system design can be captured by a concern representation. Our measurement approach aims at quantifying the modularization of concerns in both architectural and detailed designs. Therefore, before defining how a concern is represented, the next subsection defines the representation of architectural design (Section 4.2.1) adopted by our approach. We left to present the

representation of detailed design in Chapter 5, because it is used in the definition of detailed design metrics, which are presented in the same chapter.

4.2.1. Architectural Design

Our measurement approach is rooted at component-and-connector views (Bass et al., 2003) of the system architecture. Component-and-connector views are the models adopted by a plethora of ADLs and, more notably, by UML 2.0 (OMG, 2005). Moreover, a number of aspect-oriented ADLs have been developed recently (Section 3.2). The examples based on the component-and-connector views are described in this thesis using UML 2.0 (OMG, 2005). This section defines the terminology and formalism regarding component-and-connector views used for expressing our architectural metrics. In order to facilitate the understanding of the definitions, an informal example of each definition is presented based on Figure 8 and Figure 10.

Component-and-Connector View

The terminology presented here is based on the definition of component-and-connector views presented by Bass et al (2003). However, for the sake of simplicity, we focus only on the architectural elements needed for defining our metrics, namely components, interfaces and operations. Our metrics suite does not distinguish between conventional components and aspectual components, thus the terminology does not need this distinction either.

Definition 1: Components and Interfaces. Let S be the architecture of a system. The component-and-connector view of S consists of a set of components $C(S)$. $C(S)$ includes both conventional and aspectual components. Each component $c \in C(S)$ contains a set of interfaces $I(c)$. Each interface can be a provided interface or a required interface. Therefore, each component $c \in C(S)$ contains a set of provided interfaces $PI(c) \subseteq I(c)$, and a set of required interfaces $RI(c) \subseteq I(c)$, so that $I(c) = PI(c) \cup RI(c)$. Each interface $i \in I(c)$ encompasses one or more operations denoted as $O(i)$. These operations also include events. We also define:

(i) the set of all operations of a component *c*, represented as $O(c) = \bigcup_{i \in I(c)} O(i)$,

(ii) the set of all interfaces of an architecture *S*, represented as $I(S) = \bigcup_{c \in C(S)} I(c)$,

(iii) the set of all operations of an architecture *S*, represented as $O(S) = \bigcup_{i \in I(S)} O(i)$.

Example. The Health Watcher architecture in Figure 8 consists of five components – GUI_Elements, Distribution_Manager, Business_Rules, Transaction_Control, and Data_Manager. The Distribution_Manager component has a provided interface called distributedSavingService, which is connected to a required interface of the GUI_Elements component, called saveEntity. The distributedSavingService interface has four operations.

Component Interaction

We consider that there are two kinds of component interaction in an aspect-oriented architecture. In the first kind, a component *c* invokes an operation from another component *c'*. We say that *c* uses *c'*. In this case, a required interface of *c* is linked to a provided interface of *c'* by means of a conventional connector (Section 3.2). In Figure 8, the GUI_Elements component uses the Distribution_Manager component.

The second kind of interaction occurs between an aspectual component and a component. In this kind of interaction, a provided interface of an aspectual component *c* is linked to either a required or a provided interface of a component *c'* by means of an aspectual connector (Section 3.2). The aspectual component *c* executes one of its operations when the component *c'*:

(i) invokes an operation of another component (*c* is linked to a required interface of *c'*); or

(ii) has an operation invoked by another component (c is linked to a provided interface of *c'*).

We say that *c* affects *c'*. In order to illustrate this kind of component interaction, Figure 10 shows an aspect-oriented alternative to the Health Watcher architecture, described based on the AOGA notation (Section 3.2.1). As in Figure 8, this is a simplified representation.

In this example, part of the persistence concern, namely the transaction control service, is addressed by the Transaction_Control aspectual component. The transService interface of this component is connected to the savingService interface of the Business_Rules component by means of a crosscuts relationship (or aspectual connector in the AO Visual Notation (Section 3.2.2)). This means that an operation of the transService interface is executed whenever an operation of savingService is invoked. This is an example of the "affects" interaction: the Transaction_Control component affects the Business_Rules component.

In order to define the metrics for concern-sensitive coupling (Section 4.3.4) and coupling between components (Section 4.3.6), it is necessary to define first the set of components used and affected by a given component. It is also necessary to define the set of components used by a given required interface.

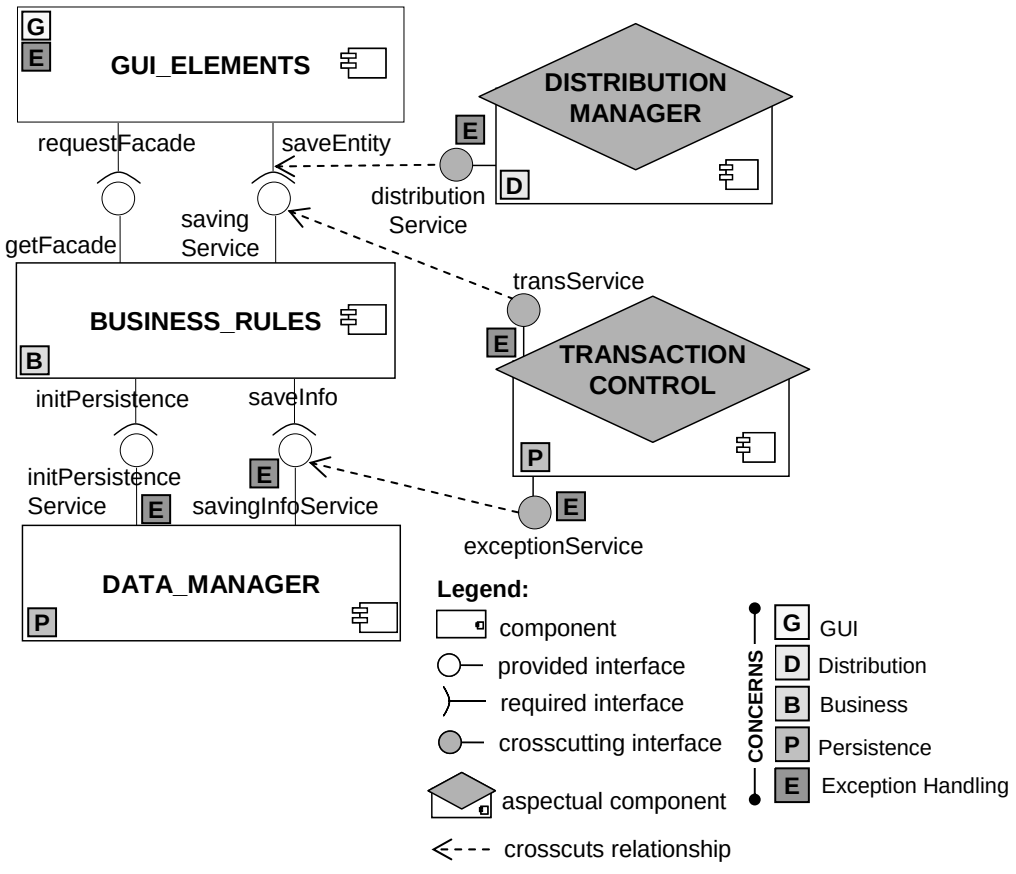

Figure 10: Aspect-oriented design alternative of Health Watcher architecture

Definition 2: Used Components per Required Interface. We define the set of components used by a required interface *i* as *UC*(*i*). Let *S* be the architecture of a system, $c \in C(S)$ be a component of *S*, and $i \in RI(c)$ be a required interface of *c*.

Then $c' \in UC(i) \Leftrightarrow \exists i' \in PI(c')$ such that i is connected to i' by means of a conventional connector.

Example. In Figure 8, Transaction_Control is the component used by the useTransaction required interface.

Definition 3: Used Components per Component. We define the set of components used by a component c as UC(c). Let S be the architecture of a system, $c \in$ C(S) be a component of S. Then $UC(c) = \bigcup_{i \in RI(c)} UC(i)$.

Example. In Figure 8, Transaction_Control and Data_Manager are the components used by the Business_Rules component.

Definition 4: Affected Components per Component. We define the set of components affected by a component c as AC(c). Let S be the architecture of a system, and $c \in$ C(S) be a component of S. Then $c' \in$ AC(c) $\Leftrightarrow \exists i \in$ PI(c) $\wedge \exists i' \in$ I(c') such that i is connected to i' by means of an aspectual connector.

Example. In Figure 10, Business_Rule and Data_Manager are the components affected by the Transaction_Control component.

The definitions presented in this section are used in the definition of architectural concern adopted in our approach (Section 4.2.2). Now, it is possible, for instance, to define the sets of components, interfaces and operations to which a concern is mapped. In addition, some of the definitions presented here are also directly used in the metrics definitions (Section 4.3).

4.2.2. Architectural Concern

With the definition of the elements of the architectural design representation considered in our approach (Section 4.2.1), it is possible to define the notion of architectural concern representation (or simply, architectural concern). A concern *con* consists of a list of architecture elements assigned to it. These elements can be

components, interfaces or operations. An architectural element can be responsible for realizing more than one concern, or parts of more than one concern. Therefore, an architectural element can be assigned to more than one concern.

Definition 5: Architectural Concern. Let *S* be the architecture of a system, for each *c* ∈ C(S), the set of concerns to which *c* is assigned is denoted as *Con(c)*. Let *i* ∈ I(c) be an interface of *c*, the set of concerns to which *i* is assigned is denoted as *Con(i)*. Let *o* ∈ O(i) be an operation of *i*, the set of concerns to which *o* is assigned is denoted as *Con(o)*. *Con(S)* is the set of all concerns in the architecture and is represented as:

$$Con(S) = \bigcup_{c \in C(S)} Con(c) \cup \bigcup_{i \in I(S)} Con(i) \cup \bigcup_{o \in O(S)} Con(o)$$

Let S be the architecture of a system, for each *con* ∈ Con(S), the set of components assigned to *con* is denoted as:

$$C(con) = \{c \mid c \in C(S) \wedge con \in Con(c)\}.$$

Similarly, the set of interfaces assigned to *con* is denoted as:

$$I(con) = \{i \mid i \in I(S) \wedge con \in Con(i)\}.$$

Finally, the set of operations assigned to *con* is denoted as:

$$O(con) = \{o \mid o \in O(S) \wedge con \in Con(o)\}.$$

Example. The gray boxes in Figure 8 represent the mapping of the concerns to the Health Watcher architecture elements. In this case, if a component is assigned to a concern, all the interfaces of this component are also considered assigned to this concern, except those which are explicitly assigned to other concerns. The following examples are obtained from the architecture of Figure 8: *Con*(Business_Rules) = {business}, *Con*(useTransaction) = {persistence}, and *Con*(repositoryExceptionalEvent) = {persistence, exception handling}, *C*(persistence) = {Transaction_Control, Data_Manager}, *I*(persistence) = {savingInfoService, initPersistenceService, transService, useTransaction}. The operations assigned to the persistence concern (*O*(persistence)) are: (i) all the operations in the interfaces savingInfoService, initPersistenceService, transService, useTransaction, (ii) the operations transactionExceptionalEvent and repositoryExceptionalEvent of the interfaces savingService,

saveDistributedEntity, distributedSavingService and saveEntity, and (iii) the repositoryExceptionalEvent of the saveInfo interface (this operation is not shown in the figure).

Concern Interaction

Nevertheless, note that even though the main purpose of the Business_Rules component is to address the business concern, it also includes elements assigned to other concerns such as the useTransaction interface (persistence concern) and exception handling operations. Note also that some elements, such as the repositoryExceptionEvent operation, are assigned to more than one concern (exception handling and persistence). This occurs because some concerns are not well modularized and, as a consequence, are not totally localized in components whose only purpose is to address them. As a result, concerns interact to each other not only by means of the relationship between components, but also because sometimes more than one concern is present in the same architecture element. Some of the metrics in our approach target at assessing the interaction between concerns. In order to define them, we first define here three forms of concern interaction which the metrics take into account.

Definition 6: Component-level Interlacing. A concern *con* is interlaced at the component level with another concern *con'* if *con* and *con'* have one or more components in common. This situation can occur in several different ways:

(i) a component is assigned to both *con* and *con'*, or

(ii) a component is assigned to *con*, and at least one interface of the same component is assigned to *con'*, or

(iii) a component is assigned to *con*, and at least one operation in any interface of the same component is assigned to *con'*, or

(iv) at least one interface of a component is assigned to *con*, and at least one interface of the same component is assigned to *con'*.

(v) at least one interface of a component is assigned to *con*, and at least one operation in the same interface or in any other distinct interface of the same component is assigned to *con'*, or

(vi) at least one operation in any interface of a component is assigned to *con*, and at least one operation in any interface of the same component is assigned to *con'*.

In order to represent that two concerns are interlaced at the component-level, we define the Boolean function ComponentInterlaced(con, con'), where con $\in$ Con(S) and con' $\in$ (Con(S) – con), as:

$$\begin{aligned} ComponentInterlaced(con,con') \Leftrightarrow\ & (\exists c \in C(con) : c \in C(con')) \vee \\ & (\exists c \in C(con) : \exists i \in I(c) : i \in I(con')) \vee \\ & (\exists c \in C(con) : \exists o \in O(c) : o \in O(con')) \vee \\ & (\exists c \in C(S) : \exists i \in I(c) : \exists i' \in I(c) : i \in I(con) : i' \in I(con')) \vee \\ & (\exists c \in C(S) : \exists i \in I(c) : \exists o \in O(c) : i \in I(con) : o \in O(con')) \vee \\ & (\exists c \in C(S) : \exists o \in O(c) : \exists o' \in O(c) : o \in O(con) : o' \in O(con')) . \end{aligned}$$

Example. In Figure 8, the business concern is interlaced at the component level with the persistence concern, once the Business_Rules component is assigned to the former, but it also has one interface (useTransaction) assigned to the latter.

Definition 7: Interface-level Interlacing. A concern *con* is interlaced at the interface level with another concern *con'* if *con* and *con'* have one or more interfaces in common. This can happen in two manners:

(i) an interface is assigned to *con*, and at least one operation of the same interface is assigned to *con'*, or

(ii) at least one operation of an interface is assigned to *con*, and at least one operation of the same interface is assigned to *con'*.

In order to represent that two concerns are interlaced at the interface level, we define the Boolean function InterfaceInterlaced(*con*, *con'*), where *con* $\in$ Con(S) and *con'* $\in$ (Con(S) – con), as:

$$\begin{aligned} InterfaceInterlaced(con,con') \Leftrightarrow\ & (\exists i \in I(con) : \exists o \in O(i) : o \in O(con')) \vee \\ & (\exists i \in I(S) : \exists o \in O(i) : \exists o' \in O(i) : o \in O(con) : o' \in O(con')). \end{aligned}$$

Example. In the example of Figure 8, the business concern is interlaced with the exception handling concern at the interface level as the business-related

interface savingService includes two operations assigned to the exception handling concern.

Definition 8: Operation-level Overlapping. A concern *con* is overlapped at the operation level with a concern *con'* if at least one operation is assigned to both *con* and *con'*. This interaction is different from the previous ones because here the same element is entirely assigned to both concerns. In order to represent that two concerns are overlapped at the operation level, we define the Boolean function OperationOverlapped(*con*, *con'*), where *con* ∈ Con(S) and *con'* ∈ (Con(S) – con), as:

$$OperationOverlapped(con, con') \Leftrightarrow \left(O(con) \bigcap O(con')\right) \neq \varnothing .$$

Example. In the architecture shown in Figure 8, the persistence concern is overlapped with the exception handling because the repositoryExceptionalEvent and the transactionExceptionalEvent operations are assigned to both concerns.

4.3. Suite of Concern-Driven Architecture Metrics

This section is targeted at defining a suite of concern-driven metrics for assessing architecture modularity. The main goal of the proposed metrics is to support the software engineers to:

- identify architectural design flaws caused by the poor modularization of architecturally-relevant concerns both in development and evolution scenarios, and
- allow the comparison of alternatives of architecture design solutions in terms of how well architecturally-relevant concerns are modularized.

To this end, our concern-driven approach complements conventional architecture metrics by explicitly promoting concern as a measurement abstraction. As claimed in Section 1.2, the main limitations of existing metrics are: (i) inaccuracy on identifying non-localized concerns, (ii) inaccuracy on identifying dependence between concerns, (iii) inaccuracy on identifying instabilities, and (iv) overemphasized use of traditional modularity-related attributes such coupling and cohesion.

Therefore, in order to tackle these limitations, our approach includes metrics for quantifying: (i) concern diffusion, (ii) interaction between concerns, (iii) concern-based cohesion, (iv) concern-sensitive coupling, and (v) concern-sensitive interface size. For instance, some of our metrics quantify the scattering of a concern realization over elements of an architectural design, such as components and interfaces. The metrics suite also evaluates how a particular concern realization affects traditional modularity-related attributes, such as coupling, cohesion and interface complexity.

The metrics presented in this chapter focus on the evaluation of software architectural design represented by means of specification approaches, such as UML [OMG, 2005] or ADLs. In particular, the metrics are defined upon abstractions and composition mechanisms of component-and-connector architecture views (Bass et al., 2003, Clements et al., 2003). However, the metrics definition is agnostic to specific graphical notations or ADLs. Therefore, in order to apply the metrics, it might be necessary to adapt their definition to specific abstractions of the architecture specification approach in use.

Before defining the architecture metrics in details, we present in Table 1 a summary of them. It provides a catalog with brief definitions for the metrics and their association with distinct modularity attributes they measure. The goal is to provide the reader with a big picture of our measurement approach and also make it easier for them to refer to the metrics' definitions while reading the remainder of the text.

In the following sections (Sections 4.3.1 to 4.3.6), each metric is described in terms of: (i) an informal definition, (ii) a formal definition based on set theory, and (iii) a simple didactic example. Also, in the preamble of each of the following sections, we present the reasoning and assumptions that motivated the use of the metrics in our approach. The metrics are defined in terms of the terminology and definitions introduced by our concern representation model (Section 4.2). The formal definition expresses the metrics consistently and unambiguously. Moreover, in order to facilitate the understanding and use of the metrics, in Section 4.4, we classify the metrics according to the criteria of a concern-oriented measurement framework (Figueiredo et al., 2008a).

Attribute	Metric	Definition
Concern Diffusion	Concern Diffusion over Architectural Components (CDAC)	It counts the number of architectural components that contribute to the realization of a given concern.
	Concern Diffusion over Architectural Interfaces (CDAI)	It counts the number of interfaces that contribute to the realization of a given concern.
	Concern Diffusion over Architectural Operations (CDAO)	It counts the number of operations that contribute to the realization of a given concern.
Interaction Between Concerns	Component-level Interlacing Between Concerns (CIBC)	It counts the number of other concerns with which a given concern shares at least a component.
	Interface-level Interlacing Between Concerns (IIBC)	It counts the number of other concerns with which a given concern shares at least an interface.
	Operation-level Overlapping Between Concerns (OOBC)	It counts the number of other concerns with which a given concern shares at least an operation.
Concern-based Cohesion	Lack of Concern-based Cohesion (LCC)	It counts the number of concerns addressed by a given component.
Coupling Between Components	Concern-Sensitive Coupling (CSC)	It counts the number of components used by a given component by means of required interfaces entirely assigned to a given concern.
	Architectural Fan-in (AFI)	It counts the number of components that use or affect a given component. It counts the number of components, not connections.
	Architectural Fan-out (AFO)	It counts the number of components used or affected by a given component. It counts the number of components, not connections.
Interface Complexity	Number of Concern Interfaces (NCI)	It counts for a given component the number of interfaces assigned to a given concern.
	Number of Interfaces (NI)	It counts the number of interfaces of a given component.
	Number of Operations (NO)	It counts the number of operations in all interfaces of a given component.

Table 1: Summary of the suite of concern-driven architectural metrics

Looking again to the Health Watcher architecture (Figure 11)[3], using the proposed metrics (Table 1) we can now quantify, for instance, the effects of the exception handling concern in the architecture. After documenting the operations related to the exception handling concern (e.g. transactionExceptionalEvent), we can compute the concern-driven metrics. The results will show that the exception handling concern is spread over several components and interfaces. Moreover, the results of the Lack of Concern-based Cohesion metric for the GUI_Elements, Distribution_Manager and Business_Rules components will show that there is

[3] The examples given in the metrics definitions are based on the Health Watcher architecture, thus we repeat this figure here to make it easier to the reader to refer to it.

more than one concern present in each of those components. In this way, the architect will be warned that in the Business_Rules component, for instance, besides the business concern, there are other concerns contributing for the complexity of the component.

4.3.1. Metrics for Concern Diffusion

This section defines the proposed metrics for concern diffusion, namely Concern Diffusion over Architectural Components (CDAC), Concern Diffusion over Architectural Interfaces (CDAI), and Concern Diffusion over Architectural Operations (CDAO). They are based on the notion of concern representation presented in Section 4.2. These metrics are defined on counting, for each architectural concern, the number of architecture elements assigned to it. They are devoted to calculate the degree to which a single concern in the system maps to distinct architectural elements.

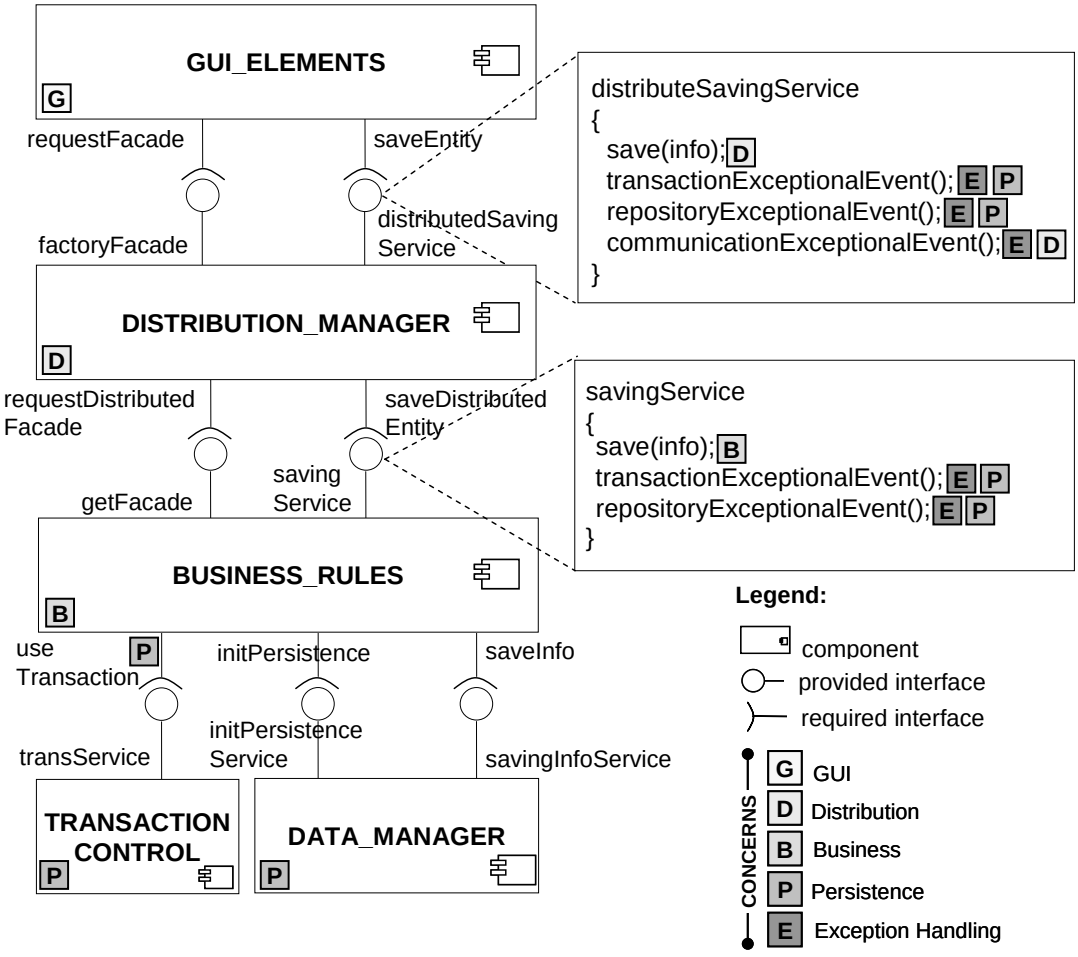

Figure 11: Simplified representation of the Health Watcher system architecture

The assumption behind this category of metrics is that a concern spread over a high number of design elements is detrimental to modularity. The understanding of a highly spread concern demands the understanding of a large part of the design. In addition, a change related to that concern may affect a large number of design elements. Therefore, these metrics aim at identifying highly scattered concerns, and quantifying its potential influence in the design.

Concern Diffusion over Architectural Components (CDAC)

Definition 9: Concern Diffusion over Architectural Components (CDAC). CDAC for a concern *con* counts the number of components in the architecture entirely assigned to *con*. The counting also includes the number of components where there is at least one interface assigned to *con*, and the number of components where there is at least one operation assigned to *con*.

Formal Definition of CDAC: Let *S* be the architecture of a system, and *con* $\in$ *Con(S)* be a concern in *S*, CDAC is represented as:

$$CDAC(con) = \Big| C(con) \bigcup \{c \mid c \in C(S) \wedge I(c) \bigcap I(con) \neq \varnothing\} \bigcup \{c \mid c \in C(S) \wedge O(c) \bigcap O(con) \neq \varnothing\} \Big|$$

Example. According to Figure 11, the value of CDAC for the persistence concern is five. This because this concern is present in: (i) the Transaction_Control and Data_Manager components, (ii) the useTransaction interface of the Business_Rules component, (iii) the two persistence-related operations in the distributedSavingService interface of the Distribution_Manager component, and (iv) the two persistence-related operations in the saveEntity interface of the GUI_Elements component (not shown in the figure). Therefore, the persistence component is spread over five components.

Concern Diffusion over Architectural Interfaces (CDAI)

Definition 10: Concern Diffusion over Architectural Interfaces (CDAI). CDAI for a concern *con* counts the number of interfaces in the architecture entirely assigned to *con*. This includes the interfaces of components entirely

assigned to *con*, plus the number of interfaces where there is at least one operation assigned to *con*.

Formal Definition of CDAI: Let *S* be the architecture of a system, and *con* ∈ Con(*S*) be a concern in *S*, CDAI is represented as:

$$CDAI(con) = \left| I(con) \bigcup \{ i \mid i \in I(S) \wedge O(i) \bigcap O(con) \neq \varnothing \} \right|.$$

Example. According to Figure 11, the CDAI value for the persistence concern is nine as four interfaces are entirely assigned to it – transService, initPersistenceService, savingInfoService and useTransaction, and there are also operations assigned to it in five interfaces: saveInfo, savingService, saveDistributedEntity, distributedSavingService and saveEntity.

Concern Diffusion over Architectural Operations (CDAO)

Definition 11: Concern Diffusion over Architectural Operations (CDAO). CDAO for a concern *con* counts the number of operations in the architecture assigned to *con* (which includes the operations of interfaces entirely assigned to *con*).

Formal Definition of CDAO: Let *S* be the architecture of a system, and *con* ∈ Con(*S*) be a concern in *S*, CDAO is represented as:

$$CDAO(con) = \left| O(con) \right|.$$

Example. In Figure 11, CDAO for the persistence concern counts all the operations in the interfaces of the Data_Manager and Transaction_Control components, plus all the operations in the useTransaction operation, and plus the repositoryExceptionalEvent and transactionExceptionalEvent operations in the five interfaces that handle persistence-specific exceptional events.

4.3.2. Metrics for Interaction between Concerns

The measures for interaction between concerns are defined based on the kinds of concern interactions defined in Section 4.2.2. These metrics target at

assessing concern dependences caused by concerns that are not well modularized, and, as a consequence, do not have well defined boundaries. They deal with interaction between concerns which are not introduced by the dependence between components.

The assumption behind this category of metrics is that much of the dependence between concerns does not occur only by means of dependence between components. Some concerns are not entirely encapsulated by components and do not have well defined boundaries. These concerns may somehow influence other concerns with which they share design elements. A change in one of these interacting concerns may ripple effects to the other. Therefore, concern that interacts with a large number of other concerns is detrimental to modularity.

Component-level Interlacing Between Concerns (CIBC)

Definition 12: Component-level Interlacing Between Concerns (CIBC). CIBC for a concern *con* counts the number of other concerns with which *con* is interlaced at the component level (Component-level Interlacing – see Definition 6 in Section 4.2.2).

Formal Definition of CIBC. Let *S* be the architecture of a system, and $con \in Con(S)$ be a concern in *S*, CIBC is represented as:

$$CIBC(con) = \left|\{con' \mid con' \in Con(S) - \{con\} \wedge ComponentInterlaced(con, con')\}\right|.$$

Example. In Figure 11, the CIBC value for the business concern is one because it is interlaced with the persistence concern at the component level, since the Business_Rules component include an interface entirely dedicated to persistence (useTransaction interface).

Interface-level Interlacing Between Concerns (IIBC)

Definition 13: Interface-level Interlacing Between Concerns (IIBC). IIBC for a concern *con* counts the number of other concerns with which *con* is interlaced at the interface level (Interface-level Interlacing – see Definition 7 in Section 4.2.2).

Formal Definition of IIBC. Let *S* be the architecture of a system, *con* ∈ Con(*S*) be a concern in *S*, IIBC is represented as:

$$IIBC(con) = \left|\{con' \mid con' \in Con(S) - \{con\} \wedge InterfaceInterlaced(con, con')\}\right|.$$

Example. In Figure 11, the IIBC value for the business concern is two, since there are operations assigned to the persistence and exception handling concerns in the savingService interface of the Business_Rules component.

Operation-level Overlapping Between Concerns (OOBC)

Definition 14: Operation-level Overlapping Between Concerns (OOBC). OOBC for a concern *con* counts the number of other concerns with which *con* is overlapped at the operation level (Operation-level Overlapping – see Definition 8 in Section 4.2.2).

Formal Definition of OOBC. Let *S* be the architecture of a system, *con* ∈ Con(*S*) be a concern in *S*, OOBC is represented as

$$OOBC(con) = \left|\{con' \mid con' \in Con(S) - \{con\} \wedge OperationOverlapped(con, con')\}\right|.$$

Example. In Figure 11, the OOBC value for the exception handling concern is two because there are operations that, besides being assigned to the persistence concerns, are also assigned to the distribution (communicationExceptionalEvent) and persistence (transactionExceptionalEvent and repositoryExceptionalEvent) concerns.

4.3.3. Concern-based Cohesion

Here we define a concern-sensitive metric for cohesion. This metric also relies on the mapping of the system concerns to the architecture elements. However, differently from the metrics presented in Sections 4.3.1 and 4.3.2, it is measured from the component point of view. The results of this metric are obtained per component, and not per concern as in the metrics defined in the previous sections. Our cohesion metric is defined on counting, for each component, the number of concerns it addresses.

The reasoning behind this metric is that a component that encompasses a large number of concerns is unstable. This is because it may suffer from effects coming from changes related to any of the concerns within it.

Lack of Concern-based Cohesion (LCC)

Definition 15: Lack of Concern-based Cohesion (LCC). LCC for a component *c* counts the number of concerns to which *c* is assigned, plus the number of distinct concerns to which the interfaces of *c* are assigned, plus the number of distinct concerns to which the operations in the interfaces of *c* are assigned.

Formal Definition of LCC. Let *S* be the architecture of a system, $c \in C(S)$ be a component in *S*, LCC can be represented as:

$$LCC(c) = \left| Con(c) \cup \bigcup_{i \in I(c)} Con(i) \cup \bigcup_{o \in O(c)} Con(o) \right| .$$

Example. In the architecture of Health Watcher (Figure 11), the LCC value for the Business_Rules component is three because it is assigned to three concerns: (i) the entire component is assigned to the business concern, (ii) its useTransaction interface is assigned to the persistence concern, and (iii) two operations in one of its interfaces are assigned to the exception handling concern.

4.3.4. Concern-Sensitive Coupling Metric

Our architecture metrics suite includes one concern-sensitive coupling metric. This metric targets at quantifying the contribution of a given concern to the coupling of a given component. As stated before, a component can encompass more than one concern. In particular, distinct interfaces of a component can realize different concerns. In this context, this metric is based on the assumption that if a given component *c* uses another component *c'* by means of a required interface entirely related to a given concern, the coupling between the two components is due to the presence of that concern in component *c*. Note that

differently from the other metrics, the values for this metric are gathered per a pair of component and concern.

The reasoning here is that the higher the number of concerns realized by a component, the higher the number of other components to which that component is coupled. This occurs because the realization of a concern by a component usually requires the use of other components. In this context, this metric aims at quantifying the amount of coupling imposed to a given component due to the realization of a given concern. This information may be useful, for instance, for the architect to analyze how much of coupling would be eliminated with the removal of a concern from a component.

Concern-Sensitive Coupling (CSC)

Definition 16: Concern-Sensitive Coupling (CSC). CSC for a component *c* and a concern *con* counts the number of distinct components used by *c* by means of required interfaces entirely assigned to *con*.

Formal Definition of CSC. Let *S* be the architecture of a system, $c \in C(S)$ be a component in *S*, and *con* be a concern in Con(S), CSC can be represented as:

$$CSC(c, con) = \left| \bigcup_{i \in CI} UC(i) \right|, \text{ where } CI = I(c) \bigcap I(con).$$

Example. In the architecture of Figure 11, the useTransaction required interface is the only interface entirely assigned to the persistence concern in the Business_Rules component. This interface is responsible for the coupling of Business_Rules to the component Transaction_Control. Therefore, the value of CSC for the Business_Rules component and the persistence concern is one (CSC(Business_Rules, persistence) = 1).

4.3.5. Number of Concern Interfaces Metric

The measurement approach includes a metric for quantifying concern-sensitive interface complexity: *Number of Concern Interfaces* (NCI). The goal of this metrics is to quantify the contribution of a given concern to the size of a given

component in terms of number of interfaces. Therefore, NCI counts, for a component, the number of interfaces responsible for realizing a given concern.

The motivation for using this metric is that a concern that comprises only few interfaces in a component (in comparison to the total number of interfaces of that component) might not be localized in that component. In addition, the information provided by this metric may be useful, for instance, for the architect to analyse how many interfaces would be eliminated with the removal of a concern from a component.

Definition 17: Number of Concern Interfaces (NCI). NCI for a component *c* and a concern *con* counts the number of interfaces in *c* assigned to *con*.

Formal Definition of NCI. Let *S* be the architecture of a system, $c \in C(S)$ be a component in *S*, and *con* be a concern in *Con(S)*, NCI can be represented as:

$$NCI(c, con) = \left| I(c) \bigcap I(con) \right| .$$

Example. In the architecture of Health Watcher (Figure 11), the useTransaction required interface is the only interface entirely assigned to the persistence concern in the Business_Rules component. Therefore, the value of NCI for the Business_Rules component and the persistence concern is one (CSC(Business_Rules, persistence) = 1).

4.3.6. Metrics for Coupling and Interface Complexity

Our metrics suite also includes metrics for quantifying conventional coupling between components and interface complexity. These metrics are inspired on traditional metrics already defined (Briand et al, 1993; Lung & Kalaichelvan, 1998; Martin, 1997). We have only adapted them to comply with our terminology (Section 4.2.1). The coupling metrics (Definitions 17 and 18) are based on definitions 3 and 4 presented in Section 4.2.1.

The reason for including conventional metrics in our approach is that we believe that they can be more useful if used together with the concern-driven ones.

This complementary use can improve the hybrid analysis of the impact of different concern modularization alternatives in conventional attributes.

Architectural Fan-in (AFI)

Definition 18: Architectural Fan-in (AFI). AFI for a component *c* is the number of distinct components which use or affect *c* (see Definitions 3 and 4 in Section 4.2.1).

Formal Definition of AFI. Let *S* be the architecture of a system, $c \in C(S)$ be a component in *S*, AFI can be represented as:

$$AFI(c) = \left|\{c' \mid c \in UC(c') \vee c \in AC(c')\}\right|.$$

Example. In the architecture of Health Watcher (Figure 11), the Business_Rules component is used only by the Distribution_Manager component. The Distribution_Manager is the only component which invokes operations from Business_Rules. Thus, the value of AFI for Business_Rules is one.

Architectural Fan-out (AFO)

Definition 19: Architectural Fan-out (AFO). AFO for a component *c* is the number of distinct components used or affected by *c* (see Definitions 3 and 4 in Section 4.2.1).

Formal Definition of AFO. Let *S* be the architecture of a system, $c \in C(S)$ be a component in *S*, AFO can be represented as:

$$AFO(c) = \left|UC(c) \cup AC(c).\right|$$

Example. In Figure 11, the Business_Rules component uses two components: Transaction_Control and Data_Manager. The Business_Rules component invokes operations from these two components. Thus, the value obtained for AFO for Business_Rules is two.

Number of Interfaces (NI)

Definition 20: Number of Interfaces (NI). NI for a component *c* counts the number of interfaces of *c*.

Formal Definition of NI. Let *S* be the architecture of a system, $c \in C(S)$ be a component in *S*, NI can be represented as:

$$NI(c) = |I(c)|$$

Example. The component DistributionManager in Health Watcher architecture (Figure 11) has four interfaces: factoryFacade, distributedSavingService, requestDistributedFacade, and saveDistributedEntity. Hence, the value of NI for this component is four.

Number of Operations (NO)

Definition 21: Number of Operations (NO). NO for a component *c* counts the number of operations of all interfaces of *c*.

Formal Definition of NO. Let *S* be the architecture of a system, $c \in C(S)$ be a component in *S*, NO can be represented as:

$$NO(c) = |O(c)|$$

Example. Figure 11 shows only a simplified representation of Health Watcher architecture. It does not show the operations of all interfaces. Therefore, we are not able to precisily calculate the value of NO for the components shown in that figure. However, just for the sake of having an example, we can assume that all the operations of the interface distributedSavingService are shown in the box on the top right. Therefore, distributedSavingService has four operations: save(info), transactionExceptionalEvent(), repositoryExceptionalEvent(), and communicationExceptionalEvent(). We can also assume that each of the other three interfaces of the Distribution_Manager component – factoryFacade, requestDistributedFacade, and saveDistributedEntity – has four operations as well. Therefore, the the value of NO for the Distribution_Manager component is 16.

4.4. Classification of the Metrics

This section aims at complementing the definition of our architecture metrics suite by classifying them according to the criteria of a measurement framework. Due to the lack of standard terminology, it is often difficult to determine how software metrics relate to one another (Briand et al., 1998, 1999). Moreover, it is also unclear what the potential uses of existing measures are and how different metrics might be used in a complementary manner (Briand et al., 1998, 1999). As a result, it is difficult for software engineers to obtain a clear picture of the state-of-the-art in order to select or define software measures.

To address and clarifying our understanding of software metrics, measurement frameworks have been proposed to support the definition, comparison, and selection of software measures (Briand et al., 1998, 1999; Kitchenham et al., 1995, Bartolomei et al., 2006). These frameworks provide a series of criteria upon which properties of the metrics should be classified. Kitchenham et al. (1995) defined a generic measurement framework that identifies elementary properties for measures validation. According to their framework, one of the criteria that a metric definition must specify is the unit of measurement. For example, you may use different units to measure temperature (e.g., Fahrenheit, or Celsius). Likewise, code length might be measured by counting the lines of code or the lexical tokens in a program listing.

Measurement frameworks specific for coupling (Briand et al., 1999) and cohesion (Briand et al., 1998) in object-oriented systems have also been developed. According to the former, one of the criteria a coupling metric must specify is the type of connection it considers as coupling. We have already mentioned this criterion in Section 2.4. Bartolomei et al. (2006) extended Briand and colleagues' coupling framework (Briand et al., 1999) to deal with aspect-oriented abstractions and new composition mechanisms.

None of the aforementioned frameworks can be directly applied to concern-driven measurement. They mainly lack criteria related to the mapping of concerns to the system modular structure (Figueiredo et al., 2008a). To cope with this limitation, a framework specific for concern-driven measurement (Figueiredo et

al., 2008a) has been developed as an adaptation of the frameworks defined by Kitchenham et al (1995) and Bartolomei et al (2006).

In this context, we classify here our architecture metrics according to the criteria of Figueiredo and colleagues' framework (Figueiredo et al., 2008a). This classification supports the software engineer in understanding our metrics and facilitates more rigorous decision making regarding the selection and use of them.

Before presenting the classification, we introduce the chosen framework and explain each of its criteria. In order to facilitate the comprehension of the criteria, we use some of our own metrics as example of each criterion. In the end of this section, we present the classification of all architecture metrics (Table 2).

4.4.1. Measurement Framework Criteria

Figueiredo and colleagues' framework (Figueiredo et al., 2008a) encompasses five criteria: entities of concern measurement, concern-aware attributes, units, concern granularity, and concern projection. We now describe each of the criteria in the order given above.

Entities of Concern Measurement

The entity of measurement determines the elements that are going to be measured. When we choose a certain element type as the entity of measurement, it means that we are interested in characteristics of this type and, therefore, the values for the metric are going to be obtained per that element type. For example, if we choose component, it means we are interested in concern-related information about components.

Usually concern measures use concerns as the entity of measurement, but other selections are also possible. For example, the metrics Concern Diffusion over Architectural Component (CDAC) (Section 4.3.1) and Lack of Concern-based Cohesion (LCC) (Section 4.3.3) have distinct entities of measurement. While CDAC has concern as entity, the entity of measurement of LCC is component. Although the most common entities of concern measurement are concern and component, other elements, such interface and operation, can be chosen for the definition of new metrics. It is also important to highlight that the

entity of measurement of the metric Concern-Sensitive Coupling (CSC) (Section 4.3.4) is the tuple “component, concern”.

Concern-Aware Attributes

Attributes are the properties that an entity possesses. For a given attribute, there is a relationship of interest in the empirical world that we want to capture formally in the mathematical world (Kitchenham et al., 1995). For instance, if we observe two concerns we can say that one is more spread than the other. A concern measure allows us to captures the “is more spread than” relationship and maps it to a formal system, enabling us to explore the relationship mathematically. An entity possesses many attributes, while an attribute can qualify many different entities (Kitchenham et al., 1995).

In the attribute selection, we may choose any property of the entity that we want to measure. For example, the metric Concern Diffusion over Architectural Components (CDAC) (Section 4.3.1) quantifies the attribute of scattering, while the metric Component-level Interlacing Between Concerns (CIBC) (Section 4.3.2) quantifies the attribute of tangling. Possible values of a measurement attribute include: (i) scattering, (ii) tangling, (iii) coupling, (iv) cohesion, and (v) size.

Units

A measurement unit determines how we measure an attribute. An attribute may be measured in one or more units, and the same unit may be used to measure more than one attribute (Kitchenham et al., 1995). Our architecture concern-driven metrics have different units of measurement. For instance, the metrics Concern Diffusion over Architectural Components (CDAC) and Concern Diffusion over Architectural Operations (CDAO) (Section 4.3.1) have “components” and “operations” as their measurement units, respectively. The metric Lack of Concern-based Cohesion (LCC) (Section 4.3.3) counts the number of concerns addressed by a given component. Therefore, its unit of measurement is “concerns”. We may choose any countable elements as measurement units, for example, (i) concerns, (ii) components, (iii) interfaces, and (iv) operations.

Concern Granularity

The granularity of a measure is the level of detail at which information is gathered. The granularity factor specifies what is counted, i.e., which elements

aggregate values. For example, in the metric Lack of Concern-based Cohesion (LCC) (Section 4.3.3) the entity is component but what we count is the number of concerns; therefore the granularity is concern.

The difference between element granularity and measurement unit is clear because all measures have to define an element to be counted. However, the measurement unit can either be omitted or be coarser than the granularity. Some metrics are defined as an equation which divides two values with the same measurement unit. For instance, we could have a metric defined as the quotient between components addressing a concern and the total components of the system. In this case this metric do not specify any unit of measurement. However, its granularity is still "component". Possible values of element granularity are, for example: (i) concern, (ii) component, (iii) interface, and (iv) operation.

Concern Projection

One of the most sensitive parts in concern-driven measurement is the mapping of concerns onto elements in the design. Figueiredo et al (2008a) call this mapping as "concern projection". At least two aspects related to concern projection have to be specified in a concern-driven measure definition. First, the level of abstraction to which the concerns have to be mapped must be specified. In our architecture metrics suite, for instance, a mapping of concerns to components is enough for computing the Concern Diffusion over Architectural Components (CDAC) metric (Section 4.3.1). However, the Concern Diffusion over Architectural Operations (CDAO) metric (Section 4.3.1) requires a mapping on the level of operations. Of course, CDAC also accepts a mapping to a finer level of abstraction such as operations. The mapping to components can be easily derived from the mapping to operations, as components encompass operations.

The other aspect related to concern projection is whether or not the metric computation allows overlapping of concerns onto the same design element. In other words, the definition of a concern-driven metric should specify whether two or more different concerns can be projected onto the same design element. For instance, our Concern Diffusion over Architectural Interfaces (CDAI) (Section 4.3.1) allows that two or more concerns are assigned to the same interface. In this case, this interface is counted in the result for each of the concerns assigned to it. Table 2 presents the classification of all our architecture metrics according to the

five criteria we have just described. Note that the projection criterion does not apply to the last four metrics, since they are not concern-driven metrics.

Metric	Entity	Attribute	Unit	Granularity	Projection: Level of Abstraction/ Overlapping
Concern Diffusion over Architectural Components (CDAC)	Concern	Scattering	Components	Component	Component/ Yes
Concern Diffusion over Architectural Interfaces (CDAI)	Concern	Scattering	Interfaces	Interface	Interface/ Yes
Concern Diffusion over Architectural Operations (CDAO)	Concern	Scattering	Operations	Operation	Operation/ Yes
Component-level Interlacing Between Concerns (CIBC)	Concern	Tangling	Concerns	Concern	Component/ Yes
Interface-level Interlacing Between Concerns (IIBC)	Concern	Tangling	Concerns	Concern	Interface/ Yes
Operation-level Overlapping Between Concerns (OOBC)	Concern	Tangling	Concerns	Concern	Operation/ Yes
Lack of Concern-based Cohesion (LCC)	Component	Tangling	Concerns	Concern	Component/ Yes
Concern-Sensitive Coupling (CSC)	(Component, Concern)	Coupling	Components	Component	Interface/ Yes
Number of Concern Interfaces (NCI)	(Component, Concern)	Size	Interfaces	Interface	Interface/ Yes
Architectural Fan-in (AFI)	Component	Coupling	Components	Component	n/a
Architectural Fan-out (AFO)	Component	Coupling	Components	Component	n/a
Number of Interfaces (NI)	Component	Size	Interfaces	Interface	n/a
Number of Operations (NO)	Component	Size	Operations	Operation	n/a

Table 2: Classification of our architecture metrics according to Figueiredo and colleagues' measurement framework (Figueiredo et al., 2008a)

4.5. Related Work

The most closely related works to our concern-driven architectural metrics are suites of metrics also developed to capture information about concerns traversing one or more structural modularity units (Sant'Anna et al., 2003; Ducasse et al., 2006; Wong et al., 2000; Eaddy et al., 2007). However, the main difference from these metrics to our architecture metrics is that almost all of them

are defined upon abstractions of implementation or detailed design level. As a consequence, they cannot be applied at early stages of software design. The only exception is the metrics proposed Ducasse et al (2006). Their metrics rest on a very generic representation model, which has to be mapped to the software design representation model to be assessed.

In addition, most of these metrics are only devoted to quantifying concern scattering. There is only one exception that is a metric proposed by Sant'Anna et al (2003), which measures tangling among concerns in terms of lines of code. In the following, we give a brief description of each of these metrics and when necessary we discuss some specific limitation beyond the ones aforementioned.

4.5.1. Metrics by Sant'Anna et al.

Sant'Anna et al (2003) defined three metrics for assessing separation of concerns in aspect-oriented detailed design and code: Concern Diffusion over Components (CDC), Concern Diffusion over Operations (CDO), and Concern Diffusion over Lines of Code (CDLOC)[4]. In fact, the first two metrics directly inspired the definition of our architectural metrics for concern diffusion (Section 4.3.1). In addition, we use CDC in our heuristic rules for assessment of detailed design modularity (Section 5.4).

Concern Diffusion over Components (CDC) counts the number of classes and aspects whose main purpose is to contribute to the implementation of a given concern. These classes and aspects are called as the primary components of the concern. Furthermore, CDC counts the number of classes, interfaces and aspects that access the primary components by calling their methods, or using them in attribute declarations, formal parameters, return types, "throws" declarations or local variables. The CDC metric enables the designer to assess the degree of concern scattering.

Concern Diffusion over Operations (CDO) counts the number of methods and pieces of advice whose main purpose is to contribute to the implementation of a given concern. In addition, it counts the number of methods and pieces of advice

[4] These metrics are not contribution of this thesis. They were proposed in the context of Sant'Anna's master dissertation.

that access any primary component of the concern by accessing their attributes, calling their methods or using them in formal parameters, return types, throws declarations and local variables. Constructors also are counted as operations. The goal of CDO is quantify the scattering of a concern in terms of how many operations are affected by it.

Concern Diffusion over Lines of Code (CDLOC) counts the number of transition points for each concern through lines of code. The use of this metric requires a shadowing process that separates the code into shadowed areas and non-shadowed areas. The shadowed areas conform to lines of code that implement a given concern. Transition points are the points in the code where there is a transition from a non-shadowed area to a shadowed area and vice-versa. The intuition is that they are points in the program text where there is a "concern switch". Figure 12 illustrates the occurrence of transition points (or concern switch). For each concern, the program text has to be analyzed line by line in order to count transition points. This is a measure of tangling of the assessed concern with the other concerns in the system.

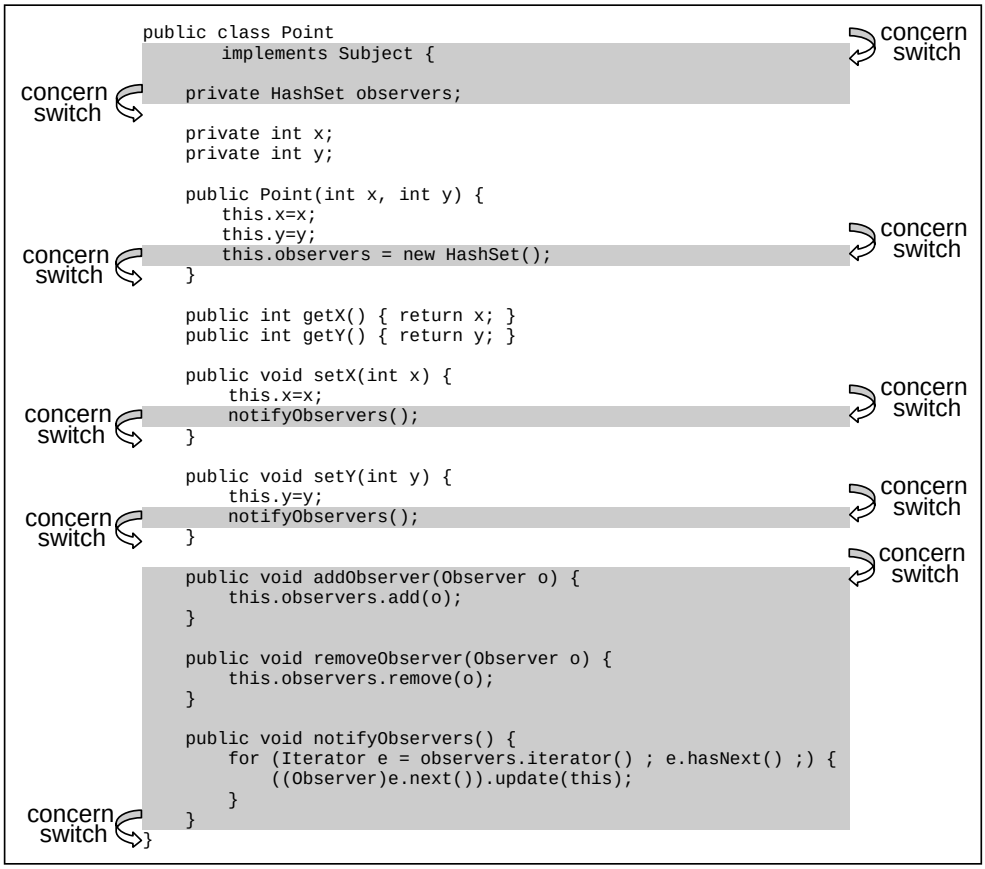

Figure 12: Transition points

4.5.2.
Metrics by Ducasse et al.

Ducasse et al (2006) proposed a very generic technique, called Distribution Map, to visualize and analyze properties of a system. Based on this technique, they defined two concern-driven measures: Spread and Focus. Their visualisation approach is composed of large rectangles containing small squares in different colours (Figure 13). The rectangles and boxes represent the system design structure. They call this representation as *reference partition*. The large rectangles, for instance, can be used to represent classes, whereas small squares can correspond to internal members of classes (operation and attributes). The colours filling the small squares represent mutually exclusive properties associated with elements of the system. They call this representation as *comparison partition*. In the context of concern-driven metrics, the comparison partition elements, i. e. the colours, represent the concerns of the system.

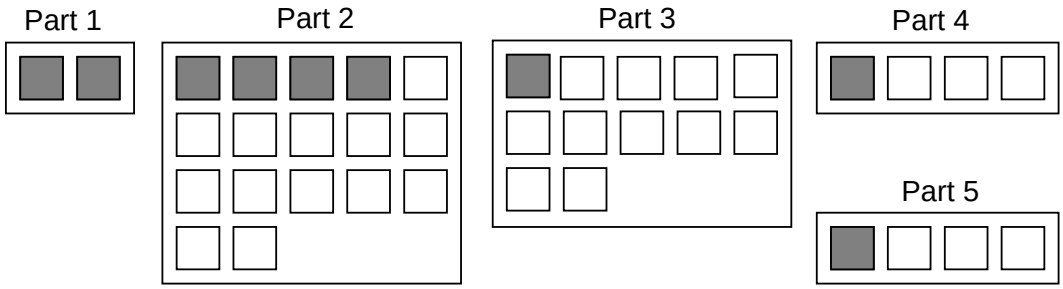

Figure 13: Distributed Map

In order to describe their metrics, let P denotes the reference partition, and Q denotes the comparison partition. Thus they say that each software artefact s_i belongs to a part p_n of P and is attributed with a property q_m of Q. In the case of concern-driven assessment of object-oriented design, s_i represents a method or attribute, p_n denotes a class, and q_m represent a concern. On the visualization (Figure 13), for each part p_n there is a large rectangle and within that rectangle, for each element $s_i \in p_n$ there is a small square whose colour refers to the property q_m attributed to that element.

In order to define the metrics, Ducasse et al (2006) first define that the set of elements in part $p \in P$ that have property q is the intersection between property q and part p. The relative size of $q \cap p$ in relation to p is denoted as:

$$touch(q, p) = \frac{|q \cap p|}{|p|}.$$

The metric *spread* of a property q over P is defined as the number of "touched" parts:

$$spread(q, P) = \sum_{p_i \in P} \begin{cases} 1, touch(q, p_i) > 0 \\ 0, touch(q, p_i) = 0 \end{cases}$$

Therefore, the *spread* metric counts the number of rectangles containing squares filled by the colour the represents the property q. In the context of object-oriented design, this metric counts the number of classes with attributes or methods which concern q is assigned to. Thus, *spread* is similar to the Concern Diffusion over Components (CDC) metric (Section 4.5.1)

The other metric defined by Ducasse et al (2006) is called *focus*:

$$focus(q, P) = \sum_{p_i \in P} touch(q, p_i) \times touch(p_i, q)$$

The *focus* is a number between 0 and 1 and measures the distance between the property q and the partition P: the larger the number, the more the parts touched by q are touched entirely by q. Although not specific for concern-driven measurement, Ducasse et al (2006) metrics suite can be consider as devoted to quantify concern scattering. However, the visualization approach upon which they are defined brings an important limitation. Each small square can only be filled by one color, which means that a design element (e.g. method) can only have one concern assigned to it. Nevertheless, in many cases, an element of software design can be responsible for addressing more than one concern simultaneously.

4.5.3. Metrics by Wong et al.

Wong et al (2000) introduced three concern measures, namely Disparity, Concentration, and Dedication. Disparity measures how many "blocks" related to a feature are localized in a particular component. For the authors, a component

can have many different meanings, depending on the system being analyzed (e.g. a single file, a group of files, a single function, or a group of functions). A feature is the functionality exercised by a given input and a block is a sequence of consecutive statements, so that if one statement is executed, all are (Wong et al., 2000). The more blocks in either a component *c* or a feature *f*, but not in both, the larger the disparity between *c* and *f*. *Concentration* and *Dedication* are also defined in terms of blocks and they quantify how much a feature is concentrated in a component and how much a component is dedicated to a feature, respectively. *Concentration* (CONC(*f*, *c*)) measures how many of the blocks related to a feature *f* are contained within a specific component *c*, and is defined as:

$$\text{CONC}(f, c) = \frac{\text{blocks in component } c \text{ related to feature } f}{\text{blocks related to feature } f}$$

Dedication (DEDI(*f*, *c*)) measured how many of the blocks contained within a component *c* are related to a feature *f*, and is defined as:

$$\text{DEDI}(c, f) = \frac{\text{blocks in component } c \text{ related to feature } f}{\text{blocks in component } c}$$

4.5.4. Metrics by Eaddy et al.

Eaddy et al (2007) presented two concern metrics based on lines of code that capture different facets of concern concentration and component dedication: *Degree of Scattering* and *Degree of Focus*. These metrics are defined based on Wong and colleagues' metrics (Section 4.5.3). However, instead of using the term "feature", Eaddy et al use the term "concern". In addition, instead of using the concept of "blocks", they use lines of codes in the definition of their metrics. Therefore, the application of their metrics demands the mapping of concerns to the source lines of code.

Degree of Scattering (*DOS*) is defined based on Wong and colleagues' Concentration metric (Section 4.5.3). DOS is a measure of the variance of the concentration of a concern over all components with respect to the worst case (i.e., when the concern is equally scattered across all components). Let *C* be a set of components, and *con* be a concern. DOS(*con*) is defined as:

$$DOS(con) = 1 - \frac{|C|\sum_{c}^{C}\left(CONC(con,c) - {}^{1}/_{|C|}\right)^2}{|C| - 1}$$

Degree of Focus (*DOF*) is defined based on Wong and colleagues' Dedication metric (Section 4.5.3). DOF is a measure of the variance of the dedication of a component to every concern with respect to the worst case (i.e. when the component is equally dedicated to all concerns). Let *Con* be a set of concerns, and *c* be a component. DOF(*c*) is defined as:

$$DOF(c) = 1 - \frac{|Con|\sum_{con}^{Con}\left(DEDI(c,con) - {}^{1}/_{|Con|}\right)^2}{|Con| - 1}$$

Table 3 classifies the metrics described in the section according to Figueiredo and colleagues' measurement framework (Figueiredo et al., 2008a).

Metric	Entity	Attribute	Unit	Granularity	Projection: Level of Abstraction/ Overlapping
Concern Diffusion over Components (CDC)	Concern	Scattering	Components	Component (classes/aspects)	Component/ Yes
Concern Diffusion over Operations (CDO)	Concern	Scattering	Operations	Operation (methods/pieces of advice)	Operation/ Yes
Concern Diffusion over LOC (CDLOC)	Concern	Tangling	Concern Switches	Line of Code	Line of Code/ Yes
Spread	Concern	Scattering	Parts	Part (Rectangle)	Part element (small square)/ No
Focus	Concern	Scattering	None	Part element (small square)	Part element (small square)/ No
Disparity	Concern, Component	Scattering	None	Block of statements	Block of statements/ Yes
Concentration (CONC)	Concern	Scattering	None	Block of statements	Block of statements/ Yes
Dedication (DEDI)	Component	Scattering	None	Block of statements	Block of statements/ Yes
Degree of Scattering (DOS)	Concern	Scattering	None	Line of Code	Line of Code/ No
Degree of Focus (DOF)	Component	Scattering	None	Line of Code	Line of Code/ No

Table 3: Classification of related metrics according to Figueiredo and colleagues' measurement framework (Figueiredo et al., 2008a)