6
Tool Support for Concern-Driven Measurement

The main characteristic of concern-driven metrics is that they are based on the representation of architectural driving concerns on the architectural design (Chapter 4). In order to apply these metrics architects must be able to specify, view and assign a system's concerns to its architecture elements. We call this set of tasks – specifying, viewing and assigning concerns – as *concern management*. Providing support for the application of concern-driven metrics requires, therefore, mechanisms and tools for managing architectural concerns. Tool support is essential in any metric-based evaluation approach. Concern management has its own particularities, thus specific tool support is important in this context. There are a number of concern-oriented software analysis tools nowadays, but they are limited, in the sense that they do not support measurement, especially at early stages of design. Section 6.1 describes these tools and discusses their limitations.

In order to support our concern-driven measurement approach, we developed an innovative tool called the Concern-Oriented Measurement Tool (COMET) (Section 6.2). COMET was developed in the context of the AOSD-Europe project (AOSD-Europe Project, 2007) and will be available as a public deliverable of the project. Before developing COMET, we defined a notation for supporting the concern-to-architecture mapping. This notation is used to describe *concern templates*. A concern template captures the architecture elements associated with a concern. It allows the architects to represent in a single place all the architectural implications related to a concern. Concern templates were very useful in our empirical studies involving architectural metrics (Section 7), because it supported the assignment of concerns to architecture elements. The notion of concern templates inspired the conception of COMET's concern management feature. We depict the concern template notation in Section 6.3.

6.1.
Limitation of Related Work

The recognition that concern identification and analysis are important through software design activities is not new. The need to document concerns in order to support software evolution was identified by Soloway et al (1998). They proposed an approach for explicitly documenting concerns on code, in order to cope with the difficulty of performing maintenance on code involving scattered concern (what they called delocalized plans). Their approach is a form of paper documentation, where source code is presented in parallel with pointers linking the code to other sections of a program.

The aspect-oriented paradigm has promoted a growing body of relevant work in the software engineering literature focusing on concern identification and documentation tools. The Aspect Browser (Griswold et al., 2001) is a tool proposed to help developers to find concerns using lexical searches of the program text. In Aspect Browser, concerns can be stored and viewed at different times to support the program evolution task. The Aspect Mining Tool (AMT) (Hannemann & Kiczales, 2001) is conceptually similar to Aspect Browser, however it supports additional forms of queries.

The Concern Manipulation Environment (CME) (Harrison et al., 2004) is a project whose purpose is to provide integrated support for creating and maintaining aspect-oriented software across the life cycle of a system. CME includes a concern explorer tool that can be used to represent concerns across different types of software engineering artifacts. CME supports its own (pattern-matching) language for code querying.

The Feature Exploration and Analysis Tool (FEAT) (Robillard & Murphy, 2007) supports the documentation of implementation concerns in a graph-based representation, called concern graphs. The tool incorporates browsing capabilities to investigate incoming and ongoing relations from the program elements in the concern graph, and based on these relations to add new elements to the concern. SoQueT (Marin et al., 2007) is another tool that supports the description and documentation of concerns in source code using queries. The difference from the others is that SoQueT is based on sort-specific queries. Sorts are categories used

to group crosscutting concerns according to typical implementation idioms and relations.

All these approaches and tools represent very relevant work on the area of concern documentation for supporting program understanding and identification of crosscutting concern. However, none of them makes use of concern documentation for measurement purpose. The assessment supported by these tools is merely qualitative. In addition, most of these tools only focus on representing concerns in source code. As a consequence, there is not much knowledge on the usefulness of using concern representation as a measurement abstraction, especially at early stages of design. Increasing the corpus of knowledge in this subject is a goal of this thesis, which proposes and evaluates a concern-driven measurement approach, partially supported by COMET.

6.2.
The Concern-Oriented Measurement Tool

The Concern-Oriented Measurement Tool (COMET) supports the task of mapping concerns to architectural design and applying concern-driven metrics and heuristic rules. Figure 20 shows an overview of COMET's modules: architecture measurement model, architecture model extractor, architecture manager, concern manager metric collector, and rule analyzer. Each module is described in the following.

Architecture Measurement Model. This model encompasses the data structure defined for architecture measurement purposes. It is a suitable representation of the architecture in order to make the measures collection easier. It follows a generic meta-model for representing aspect-oriented component-and-connector architectural views (Section 3.2). We call this meta-model as *architecture measurement meta-model* (Figure 21). The meta-model is simple in the sense that it only includes abstractions and composition mechanisms necessary for the metrics computation. It is also generic since it includes only basic abstractions of component-and-connector viewtype (Clements et al., 2003). Thus, all the existing aspect-oriented and conventional ADLs and component-and-connector graphical languages can be mapped to it.
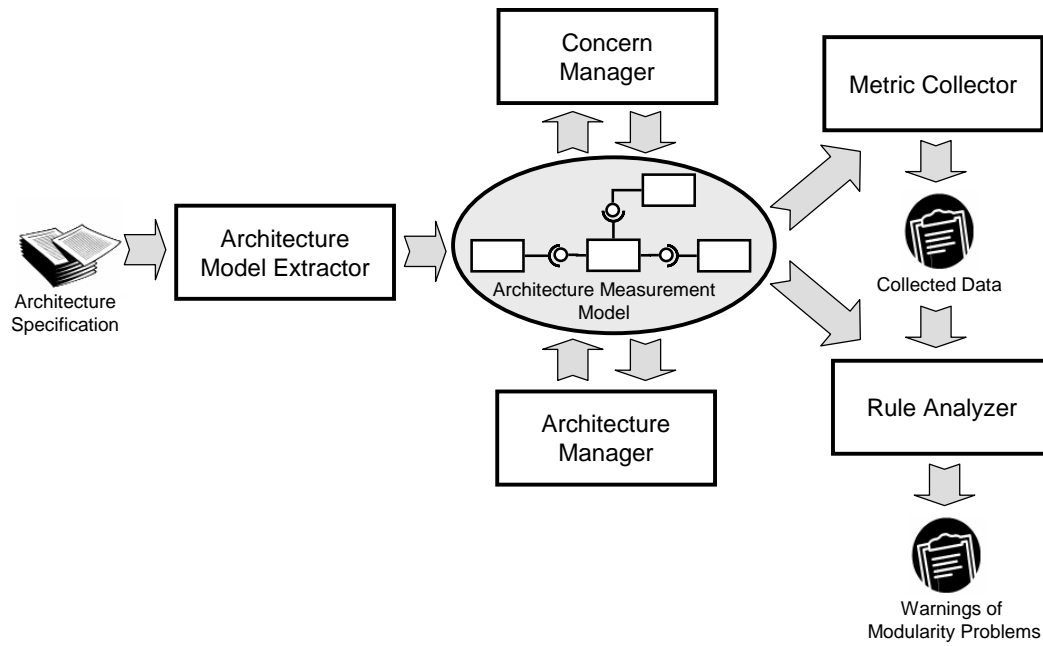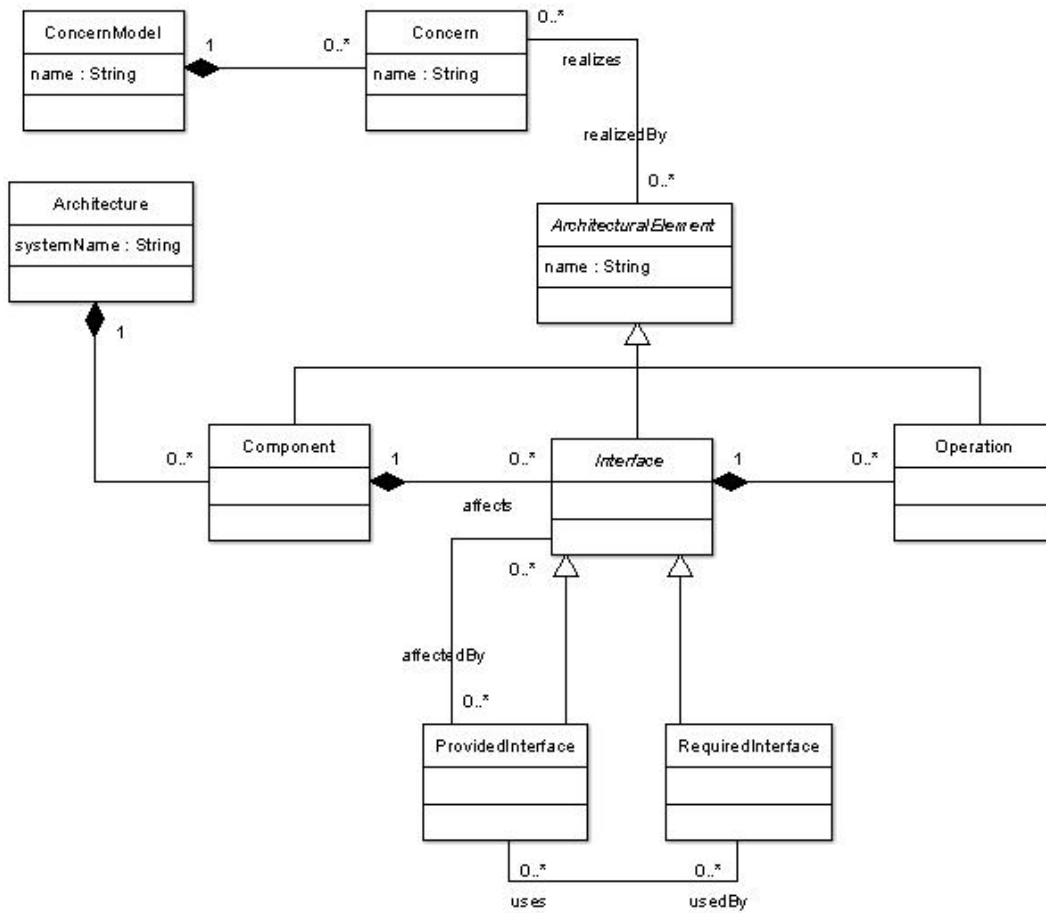
Figure 20: Overview of COMET's modules

Figure 21: Architecture measurement meta-model

A model following the architecture measurement meta-model includes *components*, *interfaces* and *operations*. Components are composed of interfaces, which, in turn, are composed of operations. Interfaces can be either *provided* or *required* interfaces. Required interfaces use provided interfaces, thus required interfaces are connected to provided interfaces (conventional connection (Section 3.2)). Provided interfaces affect provided or required interfaces (aspectual connection). An aspectual connection is a connection between an aspectual component and a conventional one (Section 3.2). Note that the meta-model does not encompasses the notion of connector, because it is not necessary for the metrics computation. The interfaces are directly attached to each other. Components, interfaces and operations are *architectural elements*. Concerns are realized by architectural elements. This is the relationship which supports the assignment of concerns to architectural elements.

Architecture Model Extractor. This module is in charge of generating the architecture measurement model. It takes as input the specification of an architecture and detects its structure in terms of their components, interfaces, and operations. It processes the architecture specification and builds the model. The architecture model extractor can be implemented targeting any architecture specification language as input, as long as it generates a model following the architecture measurement meta-model. Up to now, we implemented the architecture model extractor to take as input architecture specification described in AO-ADL (Pinto & Fuentes, 2007) (Section 3.2). This is because, similarly to COMET, AO-ADL was also defined in the context of the AOSD-Europe project (2007). In addition, tool support has been recently developed for describing aspect-oriented software architectures using AO-ADL.

Architecture Manager. This module allows the user to manipulate an architecture specification by including, removing or changing architecture elements (components, interfaces and operations) and their connection. Architects can even build an architecture specification from scratch using the architecture manager. However, they are limited to use the abstractions provided by the architecture measurement meta-model.

Concern Manager. This module supports the mapping of architecturally-relevant concerns to architecture elements. In particular, it allows the user to specify and manage the list of concerns to be considered in the measurement

process. It also allows the user to assign each concern to the architecture elements that realize it. In addition, the user can view all the architecture elements assigned to a concern in a single place, as is done by the *concern templates* mechanism (Section 6.3).

Metric Collector. This module is responsible for computing the architectural metrics (Section 4.3). It takes as input the architecture measurement model and computes the metrics for each concern and architecture elements specified in the model.

Rule Analyzer. This module computes the architectural heuristic rules. Although we have not thoroughly exploited heuristic rules at the architectural design level yet, COMET supports the application of certain rules at this level of abstraction. The supported rules are those whose definition can be easily adapted from the detailed to the architectural design context. Figure 22 presents the rules supported by COMET. The complete description about these rules can be found in Chapter 5.

Adapting the detailed design rules (Chapter 5) to the context of architectural design, required only slight modifications. For instance, Rules R3 and R4 rely on the Concern Diffusion over Architectural Components metric (CDAC) instead of Concern Diffusion over Components (CDC). The former counts architectural components while the latter counts classes and aspects. Besides, rules R4, R5, R6 and R7 are based only on counting the operations. These rules do not include metrics based on the number of attributes, differently from the equivalent rules for detailed design assessment. This occurs because the concept of architectural component considered in this thesis does not encompass attributes. Finally, the Number of Components metric (NC) counts the number of architectural components rather than classes and aspects.

6.2.1.
User Interface

To enable future integration of the measurement-specific tasks with normal software development activities, COMET was built as a plug-in for Eclipse platform (Object Technology International, 2001; Eclipse Foundation, 2007a). Eclipse is an integrated development environment with an infrastructure that

supports the inclusion of modules, called *plug-ins*, that add to the environment's functionality. In Eclipse, functionality is provided at two different levels: the workbench level, and the view level.

R01 - *Isolated*:
if CIBC = 0
then CONCERN is ISOLATED

R02 - *Tangled*:
if CIBC > 0
then CONCERN is TANGLED

R03 - *Little Scattered*:
if CDAC / NC of CONCERN < 0.5
then TANGLED CONCERN is LITTLE SCATTERED

R04 - *Highly Scattered*:
if CDAC / NC of CONCERN ≥ 0.5
then TANGLED CONCERN is HIGHLY SCATTERED

R05 - *Well Encapsulated*:
if (NCO / NOO ≥ 0.5) for every component with CONCERN
then LITTLE SCATTERED CONCERN is WELL-ENCAPSULATED

R06 - *Crosscutting*:
if (NCO / NOO < 0.5) for at least one component with CONCERN
then LITTLE SCATTERED CONCERN is CROSSCUTTING

R07 - *Well Encapsulated*:
if (NCO / NOO ≥ 0.5) for every component with CONCERN
then HIGHLY SCATTERED CONCERN is WELL-ENCAPSULATED

R08 - *Crosscutting*:
if (NCO / NOO < 0.5) for at least one component with CONCERN
then HIGHLY SCATTERED CONCERN is CROSSCUTTING

Figure 22: Heuristic rules supported by COMET

The workbench is the main application window. The workbench is the interface to a collection of resources, called the workspace. Resources in the workspace correspond to files or directories on a system. For example, Java source code files are typical Eclipse resources. Within the workspace, resources are organized into different projects. The workbench is the user interface that provides general-purpose functionality, such as opening and closing resources and performing searches. Within the workbench, more specialized functionality is

provided through different views. A view is a user interface window that displays some data and provides operations on this data.

Figure 23 shows the general layout of COMET views. These views are used to manage the architecture measurement model and constitute the user interface with the *architecture* and *concern manager* modules. The Projects view (Figure 23 – area 1) shows an example list of projects managed by COMET. The user must create a project per architecture to be assessed. Figure 24 shows the Projects view with the project for the Health Watcher architecture (simplified version). Each project in COMET encompasses two resources: Architecture.architecture and Concern_Model.architecture. These resources are XML files where the architecture measurement model is described and persisted. The Architecture.architecture resource encompasses the architecture elements, while the Concern_Model.architecture resource encompasses the concern list.
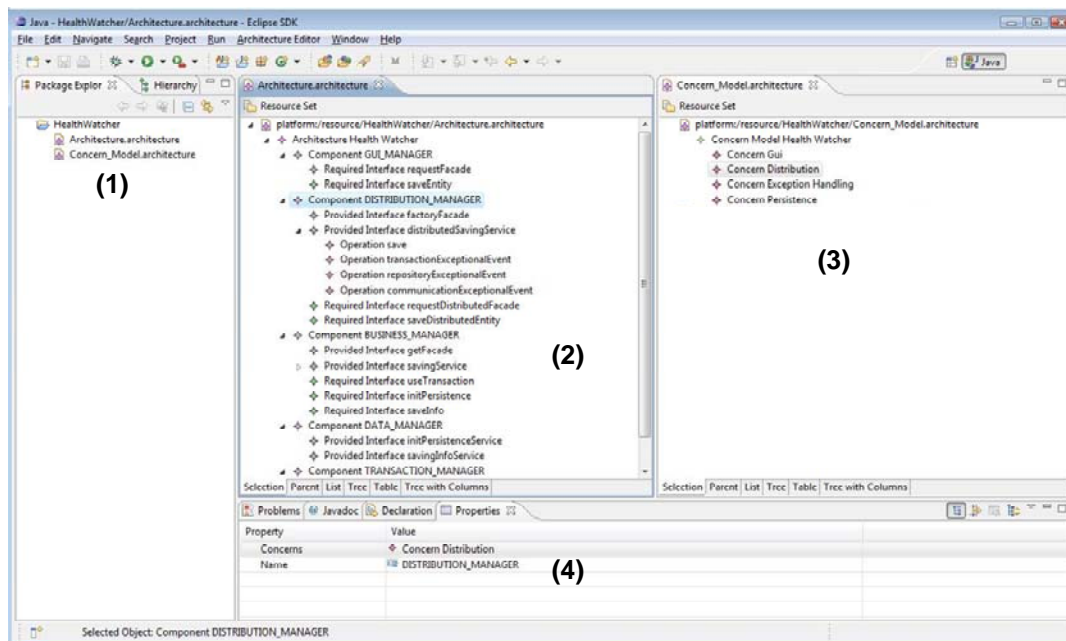

Figure 23: COMET Views. Area 1 holds the Projects View. Area 2 holds the Architecture View. Area 3 holds the Concern Model View. Area 4 holds the Properties View.

The contents of the Architecture.architecture and Concern_Model. architecture resources are shown in other two views (Figure 23 – areas 2 and 3). The view on the middle of the window is the Architecture View (Figure 23 – area

2). In this view, the architecture elements are displayed as a set of trees with components at the root of the trees. Interfaces are displayed as children of their declaring components, and operations are displayed as children of their declaring interfaces. From this view, users can add new architecture elements, and delete and rename existing ones.
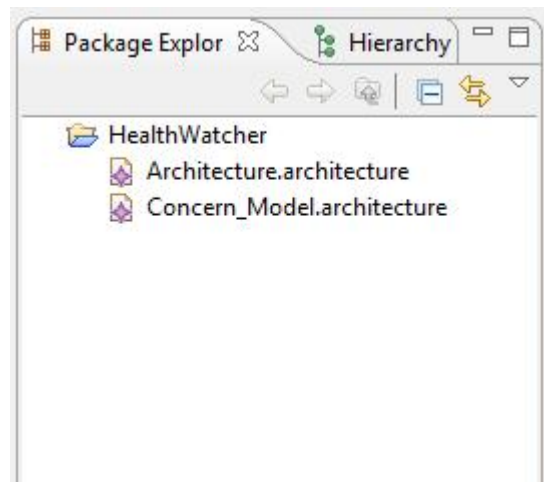
Figure 24: The Projects View

Figure 25 presents the Architecture View for the Health Watcher architecture. Selecting the name of any architecture element displays the element's properties in the Properties View (Figure 23 – area 4). Figure 26 shows the Properties View for the transactionExceptionalEvent operation. Note that, besides the element name, the Properties View presents the concerns which are assigned to the selected element. In the case of transactionExceptionalEvent operation, the concerns are persistence and exception handling. In addition, if the selected element is an interface, the Properties View presents the interfaces which the selected interface is connected to. Figure 27 presents, for instance, the Properties View for the saveEntity required interface. It shows that this interface is connected (uses) to the distributedSavingService provided interface, and is affected by none interface. See the definitions of the "uses" and "affects" relationships in Section 4.2.1.
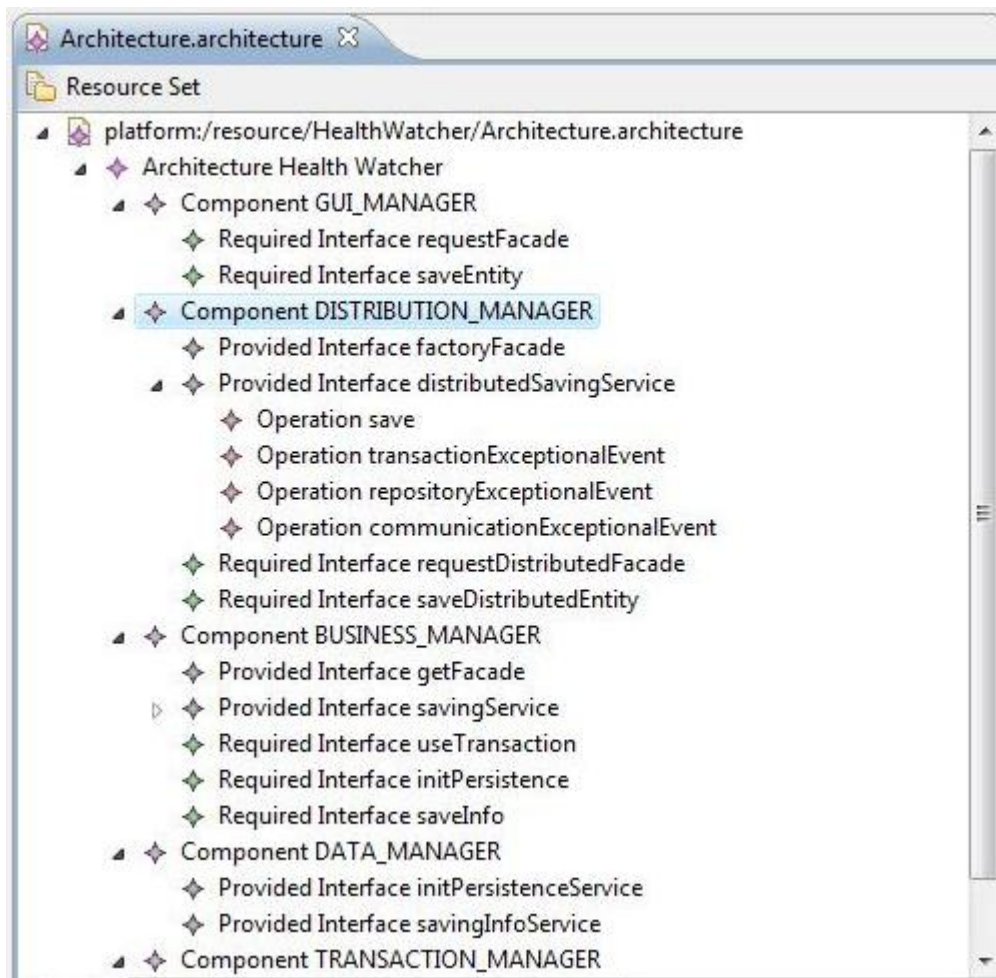
Figure 25: The Architecture View for the Health Watcher system

Figure 26: The Properties View for the transactionExceptionalEvent operation

Figure 27: The Properties View for the saveEntity interface

The view on the right side of the window is the Concern Model view (Figure 23 – area 3). This view presents the list of concerns that will be considered in the assessment process. Architecture elements are displayed as children of the concerns to which they are assigned. Figure 28 shows the Concern Model View for the Health Watcher architecture. The Properties View also displays the architecture elements which the selected concern is assigned to (Figure 29).
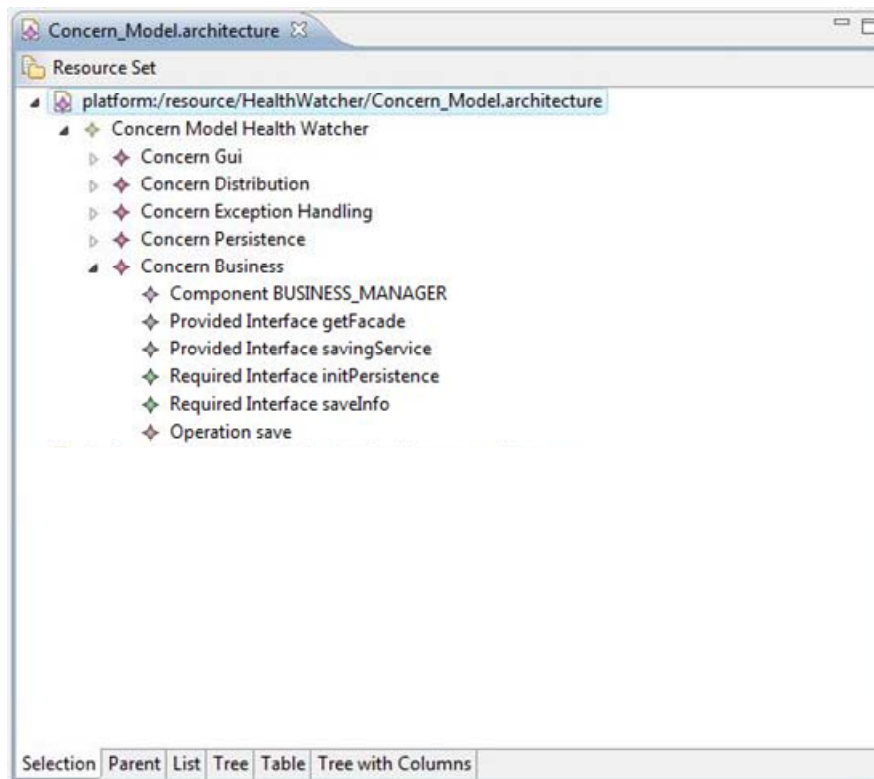
Figure 28: The Concern Model View for the Health Watcher system.
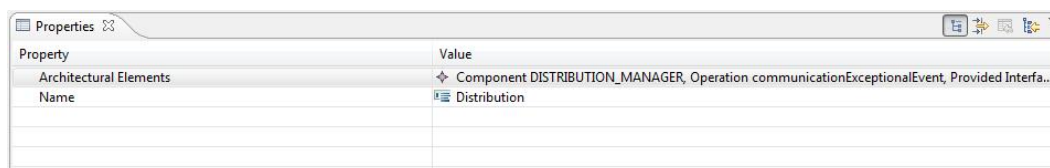
Figure 29: The Properties View for the Distribution concern

6.2.2.
Extracting an Architecture Specification

To extract an architecture specification and build a corresponding architecture measurement model, the user first uses an Eclipse wizard to create an empty general project. Figure 30 shows the window for starting the new general

project wizard. After creating a new project, the user uses the Eclipse file import wizard to select the file from which the architecture specification is to be extracted. COMET processes the file, generates the architecture measurement model and creates the Architecture.architecture resource where the model is persisted. As mentioned before, the architecture extractor module is currently able to extract architectures specified in AO-ADL (Section 3.2). However, the tool can be straightforwardly extended in order to support the extraction of architecture specified in other language. To this end, the architecture model extractor has to be implemented targeting this other language.
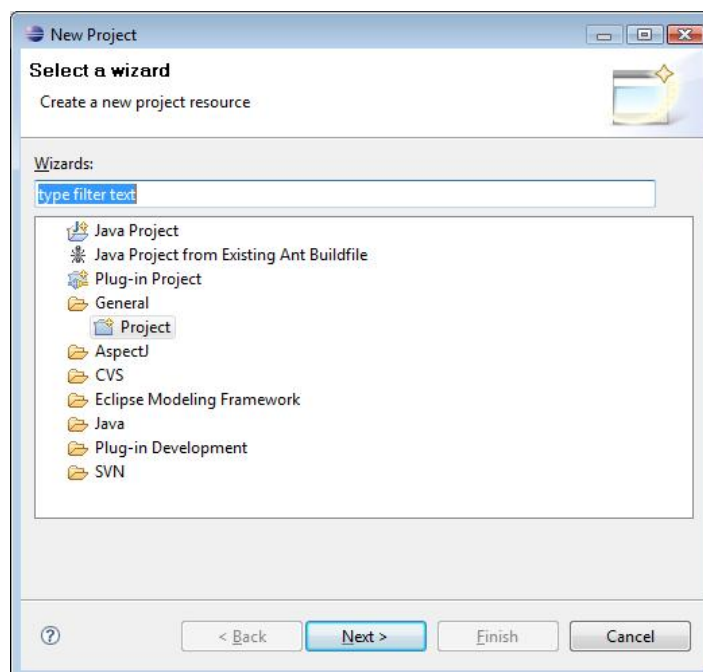
Figure 30: Wizard for creating a new project

6.2.3.
Managing the Architecture Model and Assigning Concerns

From within the Architecture View, it is possible to add or delete architecture elements from the architecture measurement model. To add a single architecture element, a user can select the new element's parent, right-click on it, select "New Child", and select the element to be added. As the element is added, the user can name it and set other information about the element in the Properties View. For example, in the case of Figure 31, if a user right-clicks on the Transaction_Manager component and selects "Provided Interface", a new provided interface will be added to this component. The user can also delete an

architecture element by right-clicking on it. See that the pop-up menu in Figure 31 also includes a "Delete" option. Similarly to the Architecture View, the Concern Model View also supports the inclusion and deletion of concerns (Figure 32).
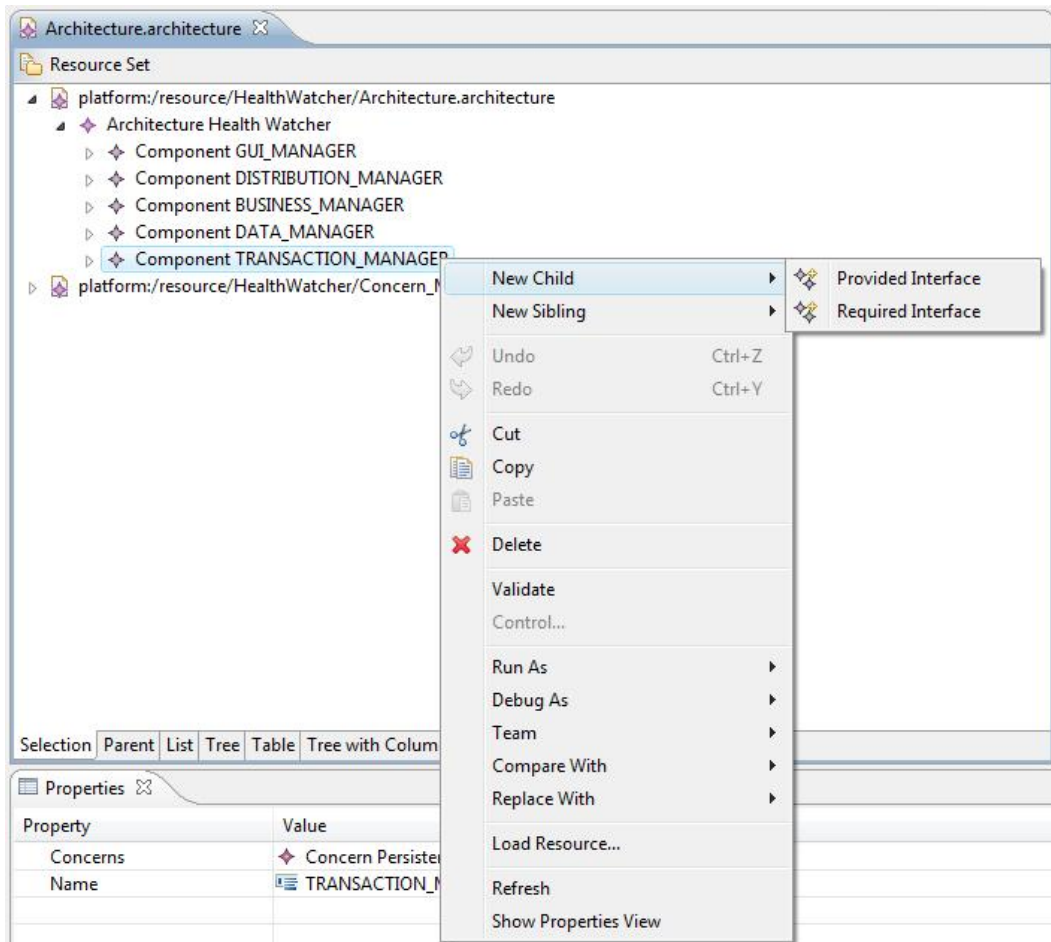
Figure 31: Adding new architecture element
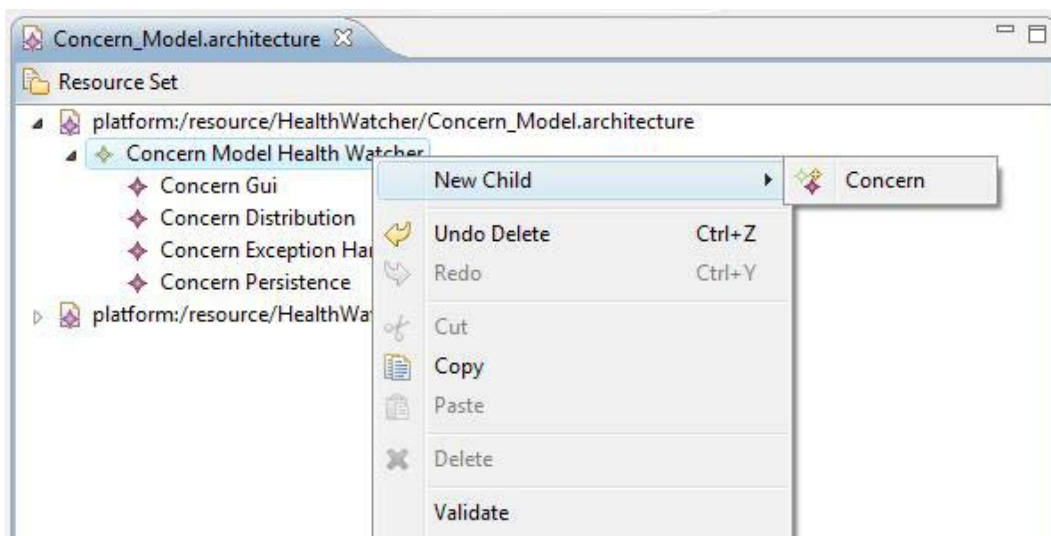
Figure 32: Adding new concerns

It is possible to assign the concerns to the architecture elements in two ways: (i) selecting the concerns related to an architecture element, or (ii) selecting the architecture elements related to a concern. In the first way, a user selects the architecture element in the Architecture View. Then, he or she selects the "Concerns" property field in the Properties View and clicks on the button that shows in this field in order to open a dialog box. From this dialog box (Figure 33), the user can select the concerns to be assigned to the selected architecture element.

In the second way of assigning concerns to architecture elements, the user selects a concern in the Concern Model View. The remainder of the process is similar to the one just described. Then, he or she selects the "Architectural Elements" property field in the Properties View and click on the button that shows in this field in order to open a dialog box. From this dialog box (Figure 34), the user can select the architecture elements that realize the selected concern.
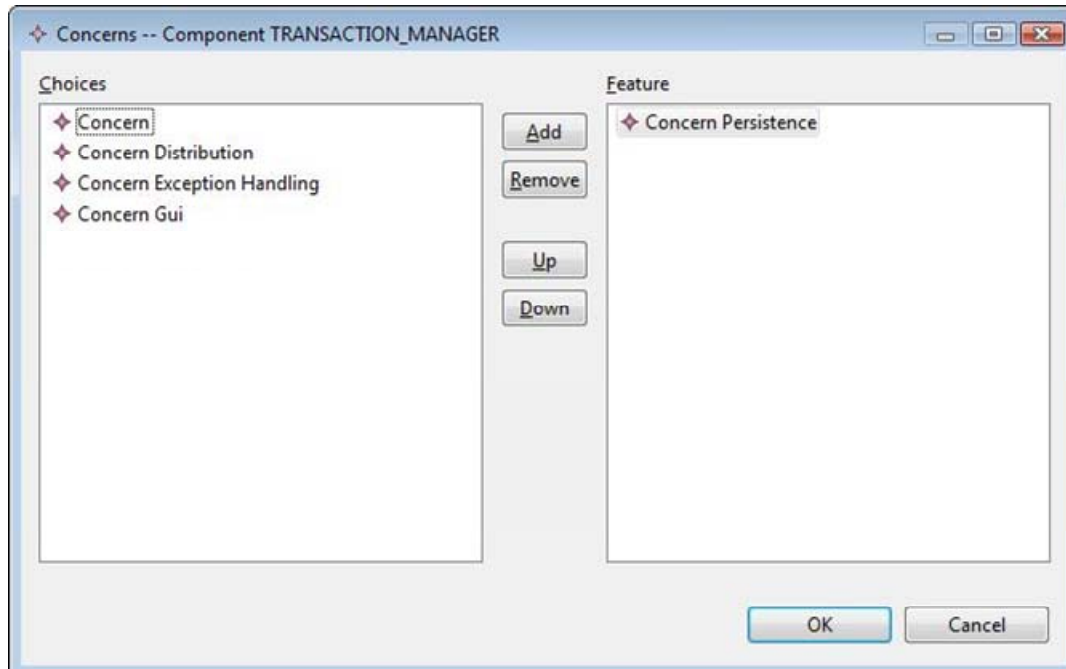
Figure 33: Selecting concerns related to an architecture element
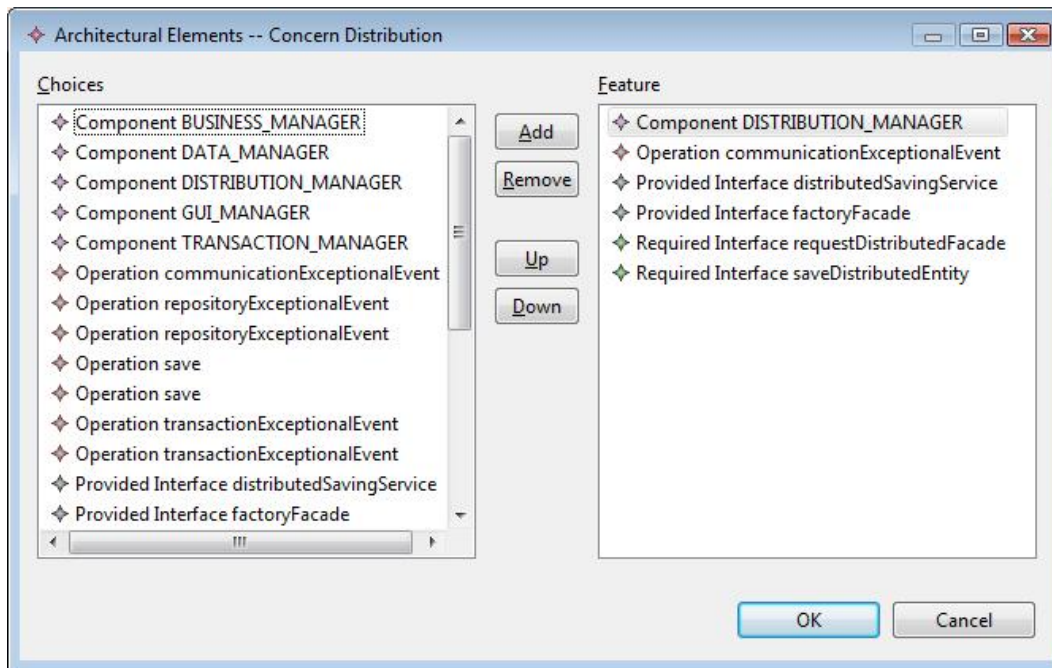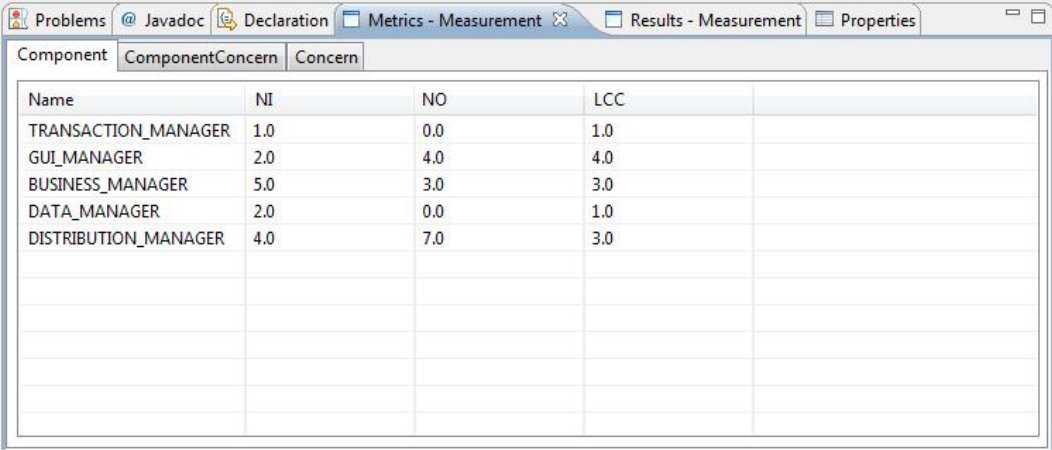
Figure 34: Selecting architecture elements related to a concern

6.2.4.
Applying metrics and heuristic rules

In order to apply the metrics and heuristic rules, a user selects the Architecture.architecture resource and clicks on the measurement button in the tool bar. Then, the metrics and rules are computed. The results are displayed in two views: the Metrics View and the Heuristic Rules View. Figure 35 shows the Metrics View. This view includes three tabs. The first tab presents the results obtained per component, such as Lack of Concern-based Cohesion (LCC), Number of Interfaces (NI), and so forth. The second tab shows the results per the pair component-concern. It includes metrics such as, Concern Sensitive Coupling (CSC), Number of Concern Operations (NCO), and so forth. Finally, the third tab presents the results per concern for metrics such as Concern Diffusion over Architectural Components (CDAC).
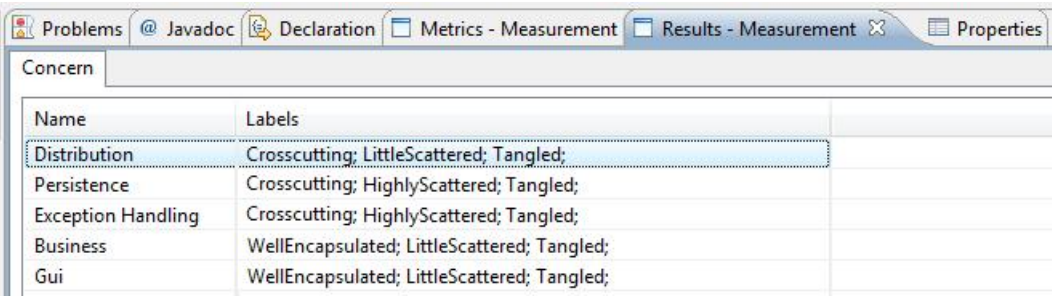
Figure 36 presents the Heuristic Rules View. This view shows the concerns and how each of them is classified by the rules. Note that not only the final classification is shown but also the partial ones. For instance, the final classification of the persistence concern is *crosscutting*. But before being classified as such, it was classified as *tangled* and *highly-scattered*. The distribution concern, in turn, is classified as *crosscutting* and *little-scattered*. It is

important to keep track of partial classifications, because they might enhance the interpretation of the final classification. A concern classified as *crosscutting* and *highly-scattered* might indicate a worse modularity problem than a *crosscutting* and *little-scattered* concern.

Figure 35: Metrics View

Figure 36: Heuristic Rules View

6.3.
Concern Templates

As mentioned in the beginning of this chapter, before developing COMET, we defined a notation which inspired the conception of COMET's concern management feature. We call this notation as *concern template*. A concern template is a documentation mechanism for capturing the architecture elements associated with key concerns in a single place. It was developed so as to support the mapping of concerns to the architecture elements and allow the application of concern-driven architectural metrics. However, the use of concern templates is not restricted to measurement purposes. Rather, the use of concern templates as a

documentation artifact can support architects on reasoning about architectural broadly scoped concerns and the implication of architectural decisions concerning to them.

A concern template includes the following information:

- name of the concern;
- architecture elements, such as components, interfaces, operations related to the concern;
- high-level composition rules to describe, in a informal and domain-dependent way, how the elements related to concern are composed with the other elements in the architecture,
- a reasoning section that captures the rationale behind the architectural decision related to the concern, and
- low-level composition rules to precisely describe how the elements related to a concern are composed with the other elements in the architecture.

Figure 37 shows how to use the notion of concern templates to support the modular description of the distribution concern in the Health Watcher architecture (simplified version). All the distribution-specific architectural decisions are clearly captured in the first template, including: (i) the creation of the Distribution_Manager component and its connection with the Business_Rules and GUI_Elements components, and (ii) the creation of an operation representing a distribution-specific exceptional event (CommunicationExceptionalEvent) and its assignment to the interfaces that raise or receive it. The rationale behind the distribution decisions are reported in the reasoning section of the template.

As a result, the template-based specification is a cohesive manner to describe the influence of a concern which otherwise could be spread over the architecture description. Notice that this approach is general and agnostic to different architectural representations that the software developers are relying on, whether textual or graphical, such as ADLs or UML-based notations. The software architect can also use the templates in conjunction with multiple architectural views, and any existing notations for reflective design, where design rationale is extensively recorded (Tyree & Akerman, 2005).

Concern: Distribution

Elements

Components and Interfaces

factoryFacade

distributedSaving
Service

DISTRIBUTION_MANAGER

requestDistributed
Facade

saveDistributed
Entity

Operations

CommunicationExceptionalEvent

High-level Composition Rules

distribute(Business_Rules, Distribution_Manager);

raiseException(distributedSavingService, CommunicationExceptionalEvent);

receiveException(saveEntity, CommunicationExceptionalEvent);

Reasoning

Distribution_Manager supports the remote distribution of the system services. It externalizes the services provided by the *Business_Rules* component at the server side and support their distribution to the clients. It makes the *Businnes_Rules* component's services remotely available to the *GUI_Elements* component.

CommunicationExceptionalEvent is a distribution-specific exceptional event raised by the *Distribution_Manager* component. It is received by the *GUI_Elements* component, which uses the services of *Distribution_Manager*.

Low-level Composition Rules

01 *// These mapping rules a related to the …*

02

03 *//… "distribute(Business_Rules)" composition rule*

04 **Connect** Business_Rules.getFacade to Distribution_Manager.requestDistributedFacade;

05 **Connect** Business_Rules.savingService to Distribution_Manager.saveDistributedEntity;

06 **Connect** Distribution_Manager.factoryFacada to GUI_Elements.requestFacade;

07 **Connect** Distribution_Manager.distributedSavingService to GUI_Elements.saveEntity;

08

09 *//… "raiseException(distributedSavingService, CommunicationExceptionalEvent)" composition rule*

10 **Add** operation CommunicationExceptionalEvent to distributedSavingService;

11

12 *//… "receiveException(saveEntity, CommunicationExceptionalEvent)" composition rule*

13 **Add** operation CommunicationExceptionalEvent to saveEntity;

Figure 37: Concern Template: Distribution

6.3.1.
Composition Rules

Concerns templates include two mechanisms for describing the relationship of a concern's architecture elements with the other elements in the architecture: high-level and low-level composition rules. The purpose of these mechanisms is support the architect to registry and reason about the influence of a concern in multiple parts of the architecture.

The high-level composition rules are optional and aim at facilitating the registration and communication of concern composition. For this end, the architects can define and use an informal and domain-specific high-level language to describe these rules. As domain-specific, we mean specific to the domain of the concern captured by the template concern. In this context, the high-level composition rules complement the low-level composition rules, which are more precise although more fine-grained and, as a consequence, harder to understand. The low-level composition rules are domain agnostic and its use in the concern template is mandatory.

Figure 37 shows how to work with a high-level language to describe composition rules related to the distribution concern in Health Watcher architecture. This is shown in the high-level composition rules section of the template. The naming of the rules is intuitive as it actually captures the architectural decisions associated with the concern. For example, the first composition rule in the template of Figure 37, named *distribute*, denotes the fact that services provided by the Business_Rules component are distributed by means of Distribution_Manager.

The composition rules can pick out different types of architecture elements, such as components, interfaces or operations. Figure 37 shows the *raiseException* and *receiveException* composition rules which affect the interfaces distributedSavingService and saveEntity, respectively. The *raiseException* composition rule denotes that CommunicationExceptionalEvent can be raised when services in distributedSavingService are used. The *receiveException* composition rule denotes that the saveEntity is aware that CommunicationExceptionalEvent can be propagated to it when requiring a service.

The low-level composition rules consist of a small set of reusable primitives. The BNF description of the low-level composition rules is presented in Figure 38. It assumes that no whitespace is necessary for proper interpretation of the rule. The item <elem-name> is to be substituted with an architecture element's name declared in the architecture description. The item <role-name> is to be substituted with a role's name specified by an architectural style. The entries *architectural_elem* and *plural_architectural_elem* should be defined according to the abstractions encompassed by the used architecture description approach. In our case, we defined them according to the component-and-connector view considered in this thesis (Section 4.2.1). Figure 37 shows how those low-level composition rules could be applied for the distribution concern in Health Watcher architecture.

rules ::= {*rule*}

rule ::= primitive | forall_statement | assignment_statement

primitive ::= add_primitive | connect_primitive | play_primitive

add_primitive ::= "Add" *architectural_elem* <elem-name> "to" <elem-name> ";"

connect_primitive ::= "Connect" <elem-name> "to" <elem-name> ";"

play_primitive ::= "Play" <elem-name> ", role" <role-name> ";"

forall_statement ::= "Forall" *variable* "in" *architecture_element_set rule_list* "end"

assignment_statement ::= architecture_element_set "=" (*all_statement* | <elem-name>) {"," (*all_statement* | <elem-name>)} ";"

all_statement ::= "All" *plural_architectural_elem* "in" (variable | <elem-name>)

variable ::= **A..Z** {**A..Z** | **0..9**}

architecture_element_set ::= **a..z** {**a..z** | **A..Z** | **0..9**}

architectural_elem :: = "component" | "interface" | "provided interface" | "required interface" | "operation" | "exception"

plural_architectural_elem :: = "components" | "interfaces" | "provided interfaces" | "required interfaces" | "operations" | "exceptions"

Figure 38: BNF description of language for low-level composition rules.

The description of each primitive and a graphical representation of its effects (when applicable) are presented below.

Add. This primitive describes the fact that the presence of the concern in the architecture makes an architecture element to be introduced to another. For instance, in Figure 37 the CommunicationExceptionalEvent operation is

introduced to interface distributedSavingService (line 10). This is due to the presence of the distribution concern (capture by the template) in the architecture. Figure 39 graphically presents the effects of using the *Add* primitive. In this example, the useTransaction interface is added to the Business_Rules component.

Add interface useTransaction to Business_Rules
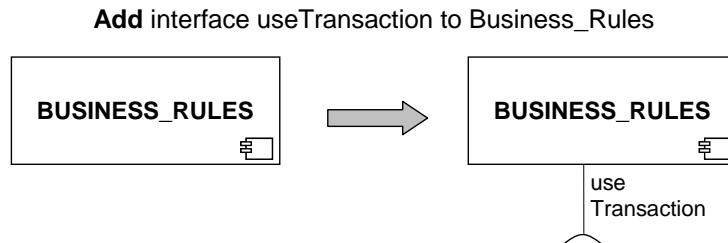
Figure 39: Add primitive

Connect. This primitive describes which elements are associated with each other. For example, it supports the description of how components' interfaces are bound. Specifically it describes the interconnection between interfaces of two different components. For instance, in Figure 37, the savingService interface of Business_Rules component is connected to saveDistributedEntity interface of Distribution_Manager (line 5). ADLs usually provide similar interconnection operations such as bind or connect. Figure 40 graphically presents the effect of this primitive based on a different example.

Connect Business_Rules.useTransaction to Transaction_Control.transService
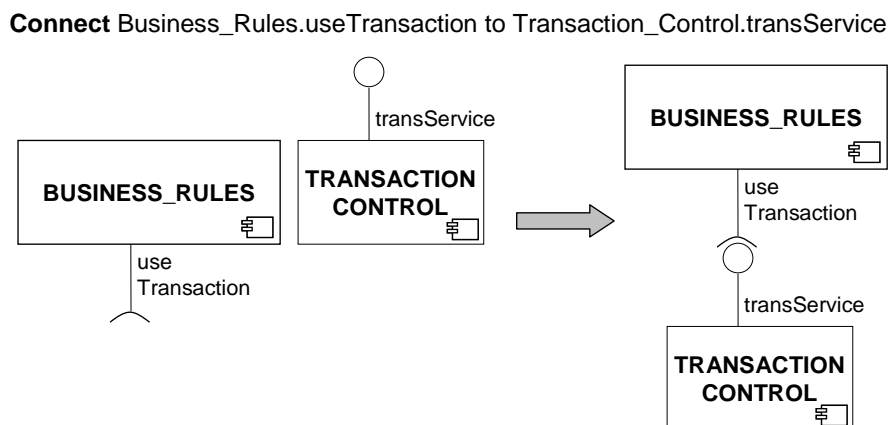
Figure 40: Connect primitive

Play. This primitive assigns a new role to an architectural element. The role is specified by a previously defined architectural style that contains architectural element types and properties. The assignment of a role to an element implies that such an element receives all the syntactic and semantic properties of the original

style. For instance, the Model-View-Controller architectural style (Buschmann et al., 1996) defines three roles: model, view and controller. The use of the *play* primitive mostly occurs when an architect decides to capture architectural styles as concerns in the concern templates. There is no example of the use of this primitive in the template shown in Figure 37. However, it is used in another example of template in Section 6.3.2 (Figure 45). Table 6 shows a summary of the set of mapping rules.

Primitive	Description
Add *<architectural_elem> <elem_name1>* **to** *<elem_name2>*	introduces an architectural element with name <elem_name1> to other architectural element with name <elem_name2>
Connect *<elem_name1>* **to** *<elem_name2>*	defines a relationship between the elements <elem_name1> and <elem_name2>
Play *<elem_name>,* **role** *<role_name>*	adds the responsibility denoted by the role <role_name> to the architectural element <elem_name>

Table 6: Primitives for defining low-level composition rules

6.3.2.
Using Concern Templates

In our empirical study involving the Health Watcher architecture (Section 7.3), we used concern templates to support the mapping of concern-to-architecture and allow the application of the metrics. In order to give a clearer vision of the use of this mechanism, we present in this section the templates for two concerns of the Health Watcher architecture: persistence and exception handling. It is important to highlight that we consider in this section the complete architecture of Health Watcher, instead of the partial and simplified version that has been used through the previous sections.

Before presenting the concern templates, we show in Figure 41 a graphical representation of the Health Watcher architecture description based on UML 2.0 notation (OMG, 2005). The Health Watcher architecture follows the combination of the client-server style with a layered style (Buschmann et al, 1996). Six main architectural concerns were considered in the Health Watcher system: GUI, distribution, business, persistence, concurrency and exception handling. In Section 7.3, we give further information about the Health Watcher system and the reasons we select it as one of our study objects.
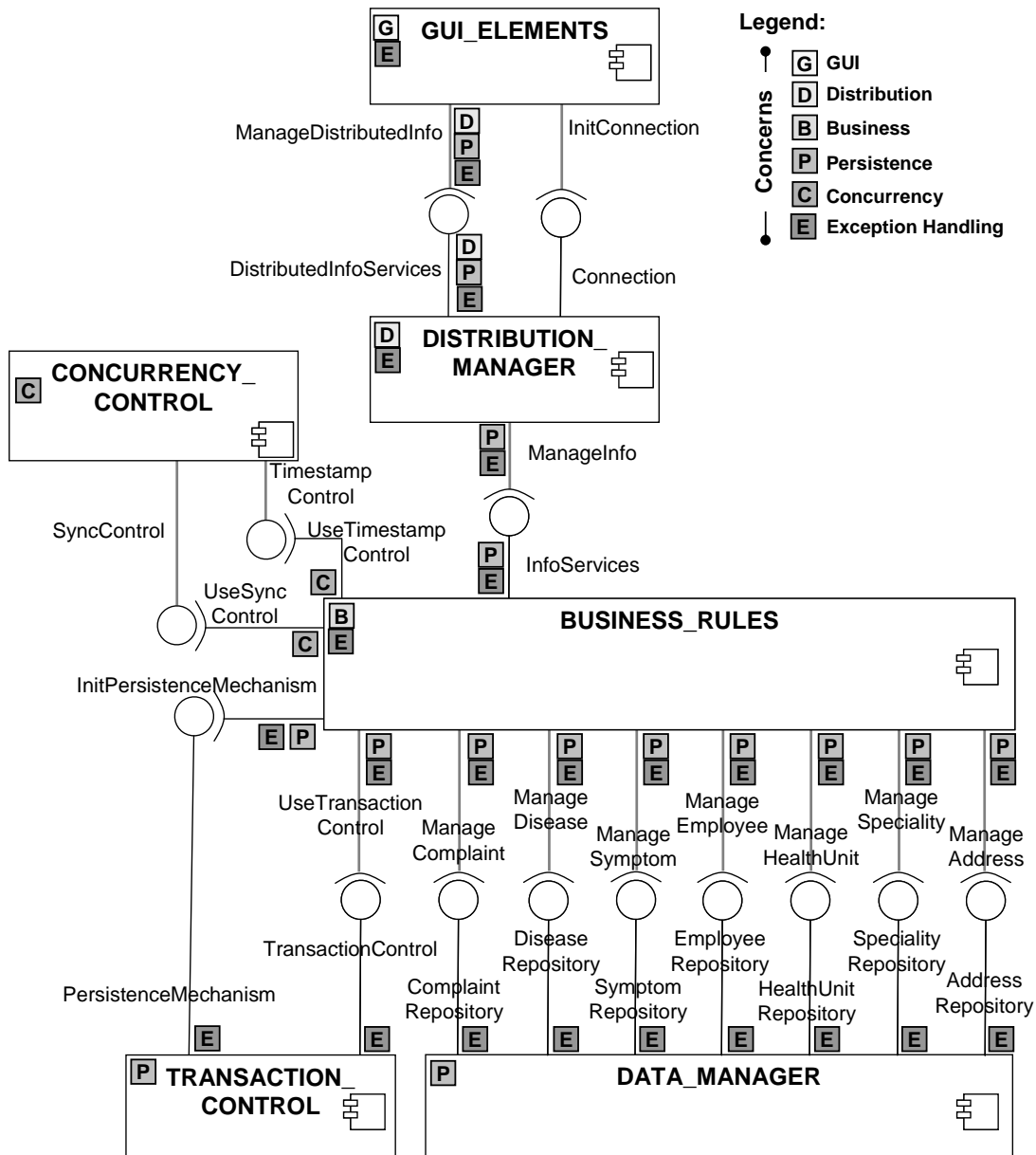
Figure 41: Health Watcher Architecture

Figure 41 also shows how the concerns are spread over the architectural elements of the Health Watcher system. The gray boxes placed over or near a component or interface indicate that the element is related to the concerns the boxes represent. For instance, the box with the letter "P" on the superior left corner of the Transaction_Control component means that this component is part of the persistence concern. Similarly, the "P" box near the UseTransactionControl required interface (in the Business_Rules component) indicates that this interface is related to the persistence component. The UseTransactionControl interface is

considered as related to the persistence concern because its only role is to require the transaction control service, which is a persistence related service.

A box near an interface also indicates that there is at least one operation in that interface that is related to that concern or raises or receives an exception related to that concern. For instance, there are three boxes near the DistributedInfoServices interface (in the Distribution_Manager component) because it contains at least: (i) one operation that raises exceptions ("E" box), (ii) one operation that raises persistence-specific exceptions ("P" box), and (iii) one operation that raises distribution-specific exceptions ("D" box). Similarly, the ManageDistributedInfo is also related to error handling, persistence and distribution concerns, but instead of raising exceptions, it receives exceptions raised by the DistributedInfoServices interface.

Persistence Concern Template

Figure 42 and Figure 43 present the concern template for the persistence concern. All the persistence-specific architecture elements are captured in that template, including: (i) the Data_Manager component and its connection with the Business_Rules component, (ii) the Transaction_Control component, (iii) the InitPersistenceMechanism and UseTransactionControl interfaces, their inclusion in the Business_Rules component and their connection with the Transaction_Control component, and (iv) two persistence-specific exceptions (TransactionException and RepositoryException) and their assignment to the operations that raise or receive them. The rationale behind the persistence decisions are reported in the reasoning section of the template.

Figure 43 shows the low-level composition rules of the persistence concern aspect (Figure 42). Each high-level composition rules (Figure 42) is translated to group of low-level composition rules in Figure 43. This practice is not mandatory, but helps the understanding of the high-level composition rules. The *persist(Business_Manager)* high-level composition rule (Figure 42) means that the information manipulated by the Business_Rules component should be persisted. It is translated into a number of **Connect** low-level rules (lines 04-10) which represent the connection between the provided interfaces of the Data_Manager component to the required interfaces of the Business_Rules component.

Concern: Persistence

Elements

Components

Interfaces

InitPersistence
Mechanism

UseTransaction
Control

Exceptions

TransactionException
RepositoryException

High-level Composition Rules

persist(Business_Rules);
controlTransaction(Business_Rules);

interfaceSet = All provided interfaces in Data_Manager, InfoServices, DistributedInfoServices;
raiseException(interfaceSet, RepositoryException);

interfaceSet = ManageComplaint, ManageDisease, ManageSymptom, ManageEmployee,
          ManageHealthUnit, ManageSpeciality, ManageAddress, ManageInfo,
          ManageDistributedInfo;
receiveException(interfaceSet, RepositoryException);

interfaceSet = TransactionControl, InfoServices, DistributedInfoServices;
raiseException(interfaceSet, TransactionException);

interfaceSet = UseTransactionControl, ManageInfo, ManageDistributedInfo;
receiveException(interfaceSet, TransactionException);

Reasoning

Data_Manager and *Transaction_Control* comprise the persistence services of the system. *Data_Manager* provide services, such as insert, update and search, for handling with persistent information manipulated by the system. This component depends on a specific persistence platform. *Transaction_Control* provides services to allow the transaction control for persisting information. These services – begin transaction, commit transaction and rollback transaction – are provided by the *TransactionControl* interface. This component is also in charge of the persistence services initialization via *the PersistenceMechanism* interface.

The *initPersistenceMechanism* is a required interface used to request the initialization of the persistence services. In the Health Watcher architecture it is realized by the *Business_Rules* component. Similarly, the *UseTransactionControl* interface requires transaction control services and is also realized by the *Business_Component*.

TransactionException and *RepositoryException* are persistence-related exceptional events raised by the *Transaction_Control* and *Data_Manager* components, respectively. They are received by the components that call the services of these components.
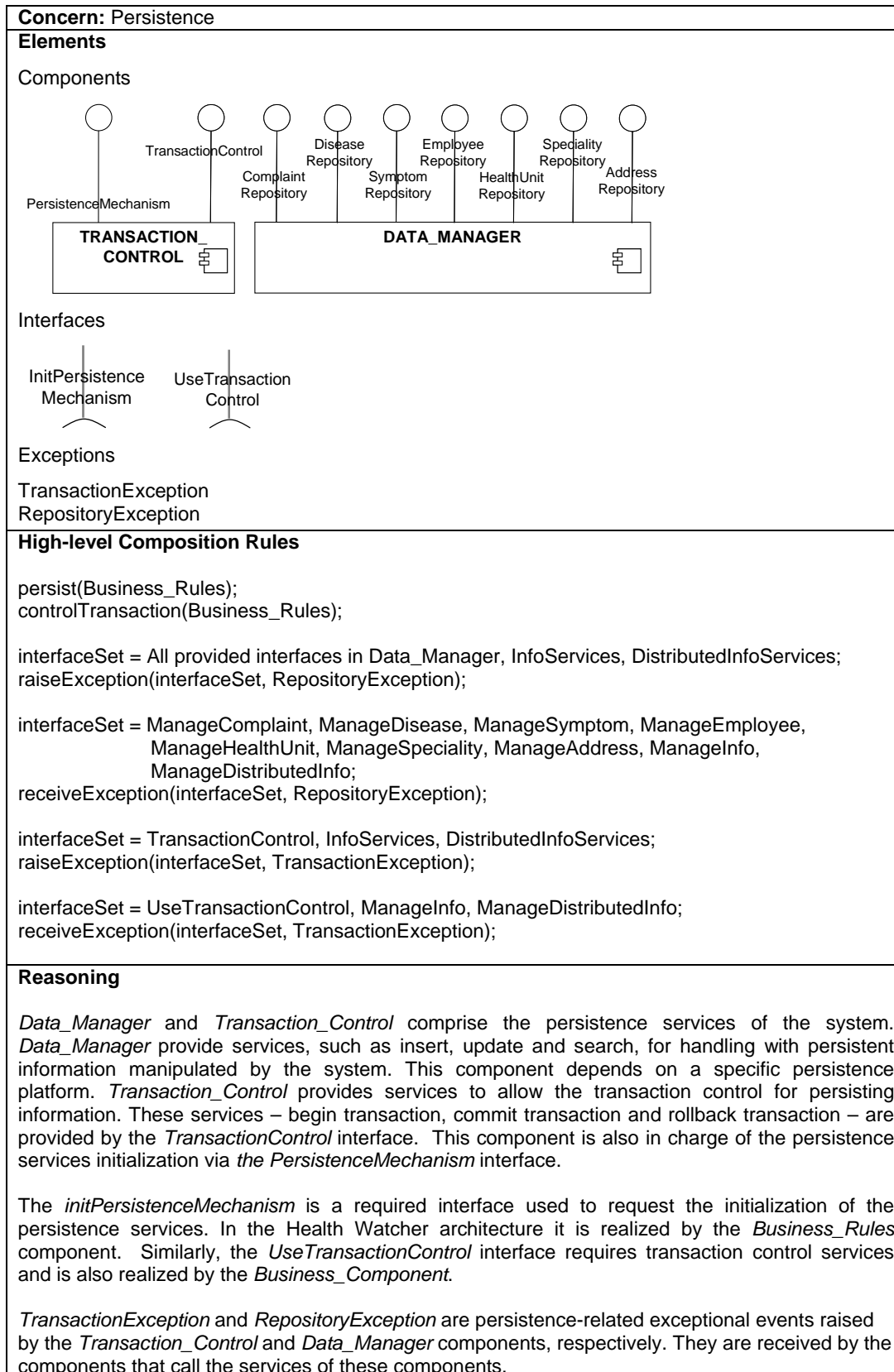
Figure 42: Concern template for the persistence concern

```
01 // These mapping rules a related to the …

02

03 //… "persist(Business_Rules)" composition rule
04 Connect Data_Manager.DiseaseRepository to Business_Rules.ManageDisease;
05 Connect Data_Manager.SymptomRepository to Business_Rules.ManageSymptom;
06 Connect Data_Manager.EmployeeRepository to Business_Rules.ManageEmployee;
07 Connect Data_Manager.HealthUnitRepository to Business_Rules.ManageHealthUnit;
08 Connect Data_Manager.SpecialityRepository to Business_Rules.ManageSpeciality;
09 Connect Data_Manager.ComplaintRepository to Business_Rules.ManageComplaint;
10 Connect Data_Manager.AddressRepository to Business_Rules.ManageAddress;
11

12 //… "controlTransaction(Business_Rules)" composition rule
13 Add interface initPersistenceMechanism to Business_Rules;
14 Connect Transaction_Control.PersistenceMechanism to Business_Rules.initPersistenceMechanism;
15 Add interface UseTransactionControl to Business_Rules;
16 Connect Transaction_Control.TransactionControl to Business_Rules.UseTransactionControl;
17

18 //… "raiseException(interfaceSet, RepositoryException)" composition rule
19 interfaceSet = All provided interfaces in Data_Manager, InfoServices, DistributedInfoServices;
20 Forall I in interfaceSet
21     operationSet = All operations in I;
22     Forall O in operationSet
23         Add exception RepositoryException to O; end
24 end
25 //… "receiveException(interfaceSet, RepositoryException)" composition rule
26 interfaceSet = ManageComplaint, ManageDisease, ManageSymptom, ManageEmployee,
                  ManageHealthUnit, ManageSpeciality, ManageAddress, ManageInfo,
                  ManageDistributedInfo;
27 Forall I in interfaceSet
28     operationSet = All operations in I;
29     Forall O in opeartionSet
30         Add exception RepositoryException to O; end
31 end
32 //... "raiseException(TransactionControl, TransactionException)" composition rule
33   interfaceSet = TransactionControl, InfoServices, DistributedInfoServices;
34   Forall I in interfaceSet
35       operationSet = All operations in I;
36       Forall O in operationSet
37           Add exception TransactionException to O; end
38   end
39 //… "receiveException(UseTransactionControl, TransactionException)" composition rule
40   interfaceSet = UseTransactionControl, ManageInfo, ManageDistributedInfo;
41   Forall I in interfaceSet
40       operationSet = All operations in I;
41       Forall O in operationSet
42           Add exception TransactionException to O; end
43   end
```

Figure 43: Low-level composition rules (continuation of the persistence concern template shown in Figure 42)

The *controlTransaction(Business_Rules)* rule captures the fact that the Business_Rules component should control the transaction while persisting the information it manipulates. This high-level rule is translated into two pairs of **Add** and **Connect** low-level rules. The first one (lines 13-14) represents the creation of the initPersistenceMechanism interface in the Business_Rules component and the connection of this interface to the PersistenceMechanism interface of the Transaction_Control component. The second pair of rules (lines 15-16) represents the creation of the UseTransactionControl interface in the Business_Rules component and the connection of this interface to the TransactionControl interface of the Transaction_Control component.

The following high-level composition rules in Figure 42 are regarding the persistence-specific exceptional events raised or received by a number of interfaces. The *raiseException(interfaceSet, RepositoryException)* high-level rule (Figure 42) specifies which interfaces raise the RepositoryException exception: (i) all the provided interfaces in Data_Manager, (ii) InfoServices in Business_Rules, and (iii) DistributedInfoServices in Distribution_Manager. The Data_Manager component raises the exception, and the Business_Rules and Distribution_Manager components propagate that exception. This rule is mapped to two loop blocks of low-level rules which add the RepositoryException to every operation in the aforementioned interfaces (lines 19-23).

In a similar way, the *receiveException(interfaceSet, RepositoryException)* high-level rule specifies which interfaces receive the RepositoryException exception. It is translated to low-level rules which add the RepositoryException to (i) specific required interfaces in the Business_Rules component, (ii) ManageInfo in Distribution_Manager, and (iii) ManageDistributedInfo in GUI_Elements (lines 26-30). Likewise, the TransactionException is added to the interfaces that raise or receive it (lines 32-42). Note that adding an exception to a provided interface means that the interface raises the exception. On the other hand, adding an exception to a required interface means that the interface receives the exception from a provided interface connected to it.

Exception Handling Concern Template

All the architecture elements related to the exception handling concern are captured in the template shown in Figure 44, including: (i) the

TransactionException, RepositoryException, and CommunicationException exceptions, (ii) the attachment of the exceptions to the interfaces that raise or receive them, (iii) the fact that GUI_Elements handles exceptions, and (iv) the fact that Distribution_Manager and Business_Rules propagate exceptions.

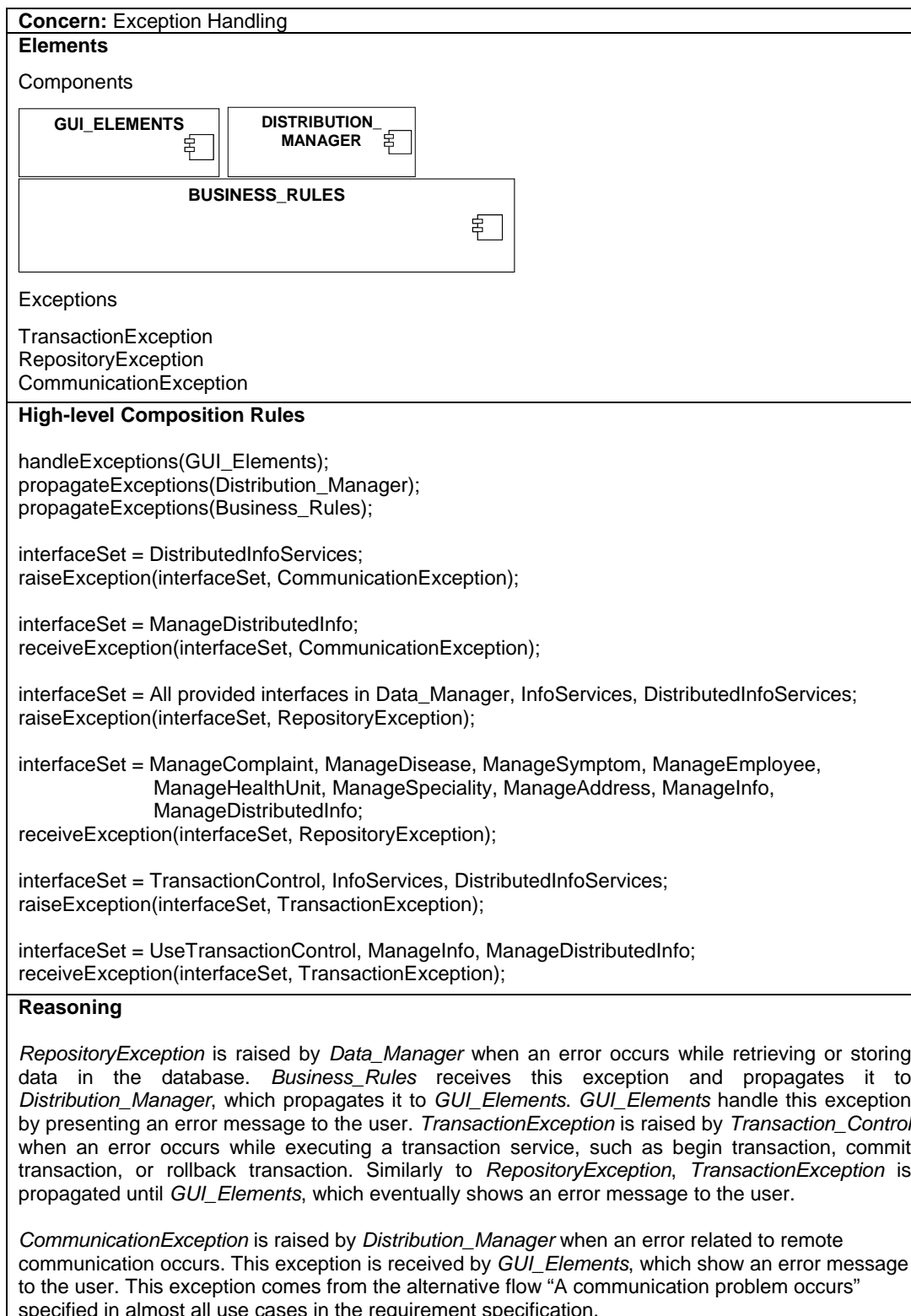| **Concern:** Exception Handling |
|---|
| **Elements**

Components

GUI_ELEMENTS   DISTRIBUTION_ MANAGER

BUSINESS_RULES

Exceptions

TransactionException
RepositoryException
CommunicationException |
| **High-level Composition Rules**

handleExceptions(GUI_Elements);
propagateExceptions(Distribution_Manager);
propagateExceptions(Business_Rules);

interfaceSet = DistributedInfoServices;
raiseException(interfaceSet, CommunicationException);

interfaceSet = ManageDistributedInfo;
receiveException(interfaceSet, CommunicationException);

interfaceSet = All provided interfaces in Data_Manager, InfoServices, DistributedInfoServices;
raiseException(interfaceSet, RepositoryException);

interfaceSet = ManageComplaint, ManageDisease, ManageSymptom, ManageEmployee,
           ManageHealthUnit, ManageSpeciality, ManageAddress, ManageInfo,
           ManageDistributedInfo;
receiveException(interfaceSet, RepositoryException);

interfaceSet = TransactionControl, InfoServices, DistributedInfoServices;
raiseException(interfaceSet, TransactionException);

interfaceSet = UseTransactionControl, ManageInfo, ManageDistributedInfo;
receiveException(interfaceSet, TransactionException); |
| **Reasoning**

RepositoryException is raised by *Data_Manager* when an error occurs while retrieving or storing data in the database. *Business_Rules* receives this exception and propagates it to *Distribution_Manager*, which propagates it to *GUI_Elements*. *GUI_Elements* handle this exception by presenting an error message to the user. *TransactionException* is raised by *Transaction_Control* when an error occurs while executing a transaction service, such as begin transaction, commit transaction, or rollback transaction. Similarly to *RepositoryException*, *TransactionException* is propagated until *GUI_Elements*, which eventually shows an error message to the user.

CommunicationException is raised by *Distribution_Manager* when an error related to remote communication occurs. This exception is received by *GUI_Elements*, which show an error message to the user. This exception comes from the alternative flow "A communication problem occurs" specified in almost all use cases in the requirement specification. |

Figure 44: Concern template for the exception handling concern

Figure 45 shows the low-level rules of the exception handling concern template (Figure 44). Again, each high-level composition rules (Figure 44) is translated to group of low-level composition rules in Figure 45. The low-level rules related to RepositoryException and TransactionException are omitted because they are identical to the ones shown for the persistence concern template (Figure 43). The *handleExceptions(GUI_Elements)* high-level composition rule (Figure 44) means that the GUI_Elements component handles the exceptions it receives. It is translated into the **Play** low-level rule (line 04) which indicates that GUI_Elements plays the role of exception handler.

The *propagateExceptions(Distribution_Manager)* and *propagateExceptions (Business_Rules)* composition rules mean that Distribution_Manager and Business_Rules, respectively, propagate the exceptions they receive. Each of them is also translated to the **Play** low-level rule (lines 07-10) which specifies that they play the role of exception propagator.

The next high-level composition rules in the template (Figure 44) determine which interfaces raise or receive exceptions. As previously explained for the persistence concern template, these composition rules are translated to blocks of the **Add** mapping rule (Figure 45 - from line 17 on).

```
01 // These mapping rules a related to the …
02
03 "… handleExceptions(GUI_Elements)" composition rule
04 Play GUI_Elements, role Exception Handler
05
06 "… propagateExceptions(Distribution_Manager)" composition rule
07 Play Distribution_Manager, role Exception Propagator
08
09 "… propagateExceptions(Business_Rules)" composition rule
10 Play Business_Rules, role Exception Propagator
11
12 "… raiseException(interfaceSet, CommunicationException)" composition rule
13 operationSet = All operations in DistributedInfoServices;
14 Forall O in operationSet
15    Add exception CommunicationException to O; end
16
17 "… receiveException(interfaceSet, CommunicationException)" composition rule
18 operationSet = All operations in ManageDistributedInfo;
19 Forall O in operationSet
20    Add exception CommunicationException to O; end
21 …
```

Figure 45: Low-level composition rules (continuation of the exception handling concern template shown in Figure 44)

6.3.3.
Related Work

The architectural perspectives approach (Woods & Rozanski, 2005) is closely related to concern templates in the sense that it considers broadly-scoped concerns at software architecture specification. Architectural perspectives provide a framework for structuring about how to design systems to achieve particular quality attribute. An architectural perspective attempts at providing advice relating to the cross view concerns of a particular quality attribute, such as security. It includes activities, checklists, tactics and guidelines to guide the process of ensuring that a system exhibits a particular set of closely related quality properties that require consideration across a number of the system's architectural views. However, the use of a perspective does not explicitly record the architectural elements related to a concern in a particular architecture. Therefore, concern templates can be complementarily used to record the architectural elements (and their rationale) made as a result of applying a perspective. Moreover, architectural perspectives are only about concerns related to quality attributes, whereas concern templates can include other concerns, such persistence.

Architectural tactics (Bachmann et al, 2003; Bass et al, 2003) are also related to concern templates. An architectural tactic is a characterization of architectural decisions that are used to achieve a desired quality attribute response. For instance, *break the dependency chain* is a key modifiability tactic that prescribes inserting an intermediary between the publisher and consumer of data and service in order to prevent propagation of change. The decisions associated to an architectural tactic can impact different parts of a system architecture specification. Nevertheless, likewise architectural perspectives, the architectural tactics approach does not provide a support for recording the architectural elements derived from a tactic. In fact, architectural perspectives (mentioned before) embrace and extend tactics by providing advice relating to what the architect should know, do and be aware of, as well as the specific solution advice provided by an architectural tactic (Woods & Rozanski, 2005). An architectural perspective can include a set of architectural tactics.

More recently, Bass et al (2004) claimed that the design decisions derived from an architectural tactic can be viewed as an architectural aspect. In other words, each use of a tactic can be considered as an architectural aspect, where the join points are the places in the architecture where the tactic was applied. They defined architectural join points as well-defined points in the specification of the software architecture. Architectural pointcuts are means of referring to collections of architectural join points. An architectural advice is a specification of transformations to perform at architectural join points. Architectural aspects are architectural views consisting of architectural pointcuts and architectural pieces of advice. This definition is based on the AspectJ programming language (Kiczales et al, 2001, The AspectJ Team, 2007) terms. Nonetheless, they do not define a systematic way for describing an architectural aspect. Besides, this approach is also restricted to concerns related quality attributes.