Universidad Nacional De La Plata (UNLP)
Facultad De Informática
Centro Internacional Franco Argentino de Ciencias de la Información y Sistemas (CIFASIS)
CONICET–UNICAM III–UNR

Tesis Doctoral

The Context Aware Contract to Dynamic Hypermedial Devices

Alejandro R. Sartorio

Director: Dro.
Co-Director: Dra.
Asesor Cientíico: Ms. Maximiliano Critía

Miembros del jurado: Jurado 1
Jurado 2
Jurado 3

Tesis presentada en la Facultad de Informática de La Plata, en cumplimiento parcial de los requisitos para optar al título de:

Doctor en Informática

FECHA DE PRESENTACIÓN

Versión Borrador: 1.0 – May 9, 2010 – 1:05

Certifico que el trabajo incluido en esta tesis es el resultado de tareas de investigación originales y que no ha sido presentado para optar a un título de postgrado en ninguna otra Universidad o Institución.

Alejandro R. Sartorio
Alejandro R. Sartorio

Contents

List of Figures

Índice de tablas

CHAPTER 1

Introduction

1.1 Background

1.2 Motivation

1.3 Objectives

1.4 Scope

1.5 Approach and Structure

CHAPTER 2

State of the Art

2.1 Introduction

2.2 Context

2.2.1 Context

Context is a concept that appears in various disciplines [85]. In philosophy and cognitive sciences, Davies and Thomson [27] highlighted the importance of understanding and studying context since "organisms, objects and events are integral parts of the environment and cannot be understood in isolation of that environment". In linguistics [43, 95], researchers seek to understand the impact of context in, for example, choosing utterance style and utterance interpretation. In psychology [112, 128], researchers are interested in how changes in context affect various cognitive processes, such as perception, reasoning, decision making, and learning.

In computer science the notion of context has been first mentioned in artificial intelligence [49, 73]. In this field, context appears as a means of partitioning a knowledge base into manageable sets or logical constructs that facilitate reasoning activities. More recently, with advances in mobile computing, context has become a topic of interest to other areas of computer science, such as telematics and ubiquitous computing. In these areas, context is usually regarded as the set of environmental conditions that can be used to adapt mobile applications to their users current situation and needs. These areas are particularly interested in context due to mobility, which generates opportunities to explore context: since the context of the user changes frequently, applications running on the users mobile devices may adapt their behaviour according to these changes. In this scope, definitions of context found in the literature focus on either the application or the applications user, as indicated by [85]. With the term applications user we refer to the end-user, i.e. a human being that uses the application. Chen and Kotz [18], for example, define context from the perspective of the application. They regard context as the set of environmental states and settings that either determines an applications behaviour or in which an application event occurs that is interesting to the user. Conversely, Dey, Abowd, and Wood [25] define context as the users physical, social, emotional, or informational state, thus focusing on the applications user, as opposed to focusing on the application. Similarly, Zetie [129] describes context as the knowledge about the goals, tasks, intentions, history and preferences of the user that a software application applies to optimize the effectiveness of the application. The most referred definition of context given by Dey et al. [22] balances the focus on both, users and applications: context is any information that can be used to characterize the situation of entities (i.e. whether a person, place, or object) that are considered relevant to the interaction between a user and an application, including the user and the application themselves.

Context versus context information

The definitions given above actually refer to context information as opposed to context. We distinguish these concepts in our approach. We regard context as the real world phenomena, while context information refers to the representation of (constituents of) context in an application, such that this representation can be manipulated and exchanged. In section 2.2.2 we further elaborate on the concept of context information. In this thesis we consider the following definition of context [79]: Context: the set of, possibly interrelated, conditions in which an entity exists. This definition reveals that context is only meaningful with respect to a thing that exists, which we call here an entity. The concept of entity is fundamentally different from the concept of context: context is what can be said about an entity in its environment, i.e. context does not exist by itself. Examples of entities are persons, computing devices and buildings. The context of an entity may have many constituents, which are defined here as

the "possibly interrelated conditions", or just context conditions. Context conditions reflect particular aspects of the circumstances in which entities exist. Examples of some context conditions of a person are the persons location, mental state, and activity. Together, these context conditions form the entity's context.

Context modelling

The process of identifying relevant context consists of determining the "conditions" of entities in the applications universe of discourse (e.g., a user or its environment) that are relevant for a context-aware application or a family of such applications. Context model: the representation of context conditions or circumstances which are relevant for a context-aware application or a family of such applications. Context modelling: the process of producing context models.

We distinguish two modelling phases in our context modelling approach, namely, conceptual context modelling and context information modelling. Conceptual context modelling aims at producing models which are conceptual models of context. Conceptual models are, in the sense of [51, 80], representations of a "given subject domain independent of specific design or technological choices". In this scope, only the concept of context (as opposed to context information) is relevant, since these models abstract from how context is sensed, provided, learned, produced and/or used. Context information modelling, on the contrary, aims at delivering models that extend the conceptual models of context by introducing technology aspects to the model, such as how context is sensed, gathered or learned. Although only few definitions of context in the literature explicitly distinguish between context and context information, and therefore also between conceptual context modelling and context information modelling, we argue that this distinction is fundamental in the development of contextaware applications. We justify this argument based on the importance of conceptual modelling in the application design process, as emphasized by [51, 80]: "conceptual specifications are used to support understanding, problem-solving, and communications, among stakeholders about a given subject domain. Once a sufficient level of understanding and agreement about a domain is accomplished, then the conceptual specification is used as a blueprint for the subsequent phases of a system's development process". Therefore, the quality of context-aware applications depends on the quality of the conceptual context models upon which their development is based.

2.3 Context Aware System

Context-aware applications differ from traditional applications since they use sensed information to adapt the service provisioning to the current context of the user. In order to achieve that, context-aware applications, in general, should be capable of:

Sensing context from the environment. The tele-monitoring application, for example, should be capable of sensing Mr. Janssen's physical conditions, such as heart rate, as well as sensing his current location and the location of relatives and healthcare professionals;

- Observing, collecting and composing context information from various sensors. In the tele-monitoring scenario, for example, the application should be capable of observing Mr. Janssens heart rate variations and inferring whether he is having an epileptic seizure. Furthermore, the system may collect location information from various sensors to detect the closest healthcare professionals and relatives;

- Autonomously detecting relevant changes in the context. The tele-monitoring application should autonomously detect the occurence of an epileptic alarm;
- Reacting to these changes, by either adapting their behaviour or by invoking (composition of) services. Upon a seizure alarm, the tele-monitoring application should invoke services to contact the closest available healthcare professionals and relatives; and
- Interoperating with third-party service providers. Location and epileptic seizure alarms may be context information offered by services designed and implemented by third-party stakeholders. In addition, services used to contact Mr. Janssens relatives and healthcare professionals are most probably offered by third-party telecommunication providers. These and other characteristics pose many challenges to the development of context-aware applications, such as:

    1. Support for context modelling abstractions that are appropriate to promote common understanding, problem-solving, and communication among the various stakeholders involved in application development [51];
    2. Bridging the gap between information sensed from the environment and information that is actually syntactically and semantically meaningful to the applications;
    3. Gathering and processing context information from distributed context services;
    4. Dynamically adapting application behaviour (reactively and proactively); and,
    5. Tailoring service delivery as needed by the user and his context. These challenges require proper abstractions and methodologies that facilitate the development process of context-aware applications. In the particular case of large-scale context-aware applications, the following aspects have so far been (partially) neglected in the literature:

- As applications become more complex and interconnected, there is an increasing need for abstract context modelling techniques to facilitate the specification of context models that are clearer and easier to understand. What are the fundamental concepts that can be beneficially used in context modelling?
- Information from several physically distributed context services may be aggregated and interpreted to yield newly produced context information, at application runtime. How to address runtime context information aggregation and interpretation?
- In a ubiquitous computing world where the environment is equipped with all kinds of sensors, applications may profit from context services that are unknown to the application beforehand. How to address ad hoc networking of context services?

2.4 Context Aware E-learning Web Applications

We discuss relevant aspects of context and context-aware applications in the sequel.

2.4.1 Definition

Figure 2-1 depicts an intuitive view of a user in his/her context, and a context-aware application (focusing on a single user only). We define a context-aware application as follows. Context-aware application: a distributed application whose behaviour is affected by its users' context.

In Figure 2-1, the arrow a shows that the user and the context-aware application interact. Similarly, the arrow b shows that the users context and the context-aware application interact. The interactions represented by arrow a enable, for example, users input to be provided to the application, such as user commands and preferences, and the use of the service delivered by the context-aware application. The interactions represented by arrow b enable the contextaware application to capture particular context conditions from the users context. As depicted in Figure 2-2, interactions take place at intersection points, which are shared mechanisms between interacting entities. Two types of shared mechanism are shown in this figure: one between a user and the context-aware application, and the other between the user's context and the context-aware application. These shared mechanisms are represented in Figure 2-2 by the intersections between a user and the context-aware

application (ip-a1.. ip-an), and between his/her context and the contextaware application (ip-b1.. ip-bn), respectively. Each of these intersections symbolizes an interaction point (ip) [40, 106]. context-aware application

usern ip-an ip-bn users contextn user1 ip-a1 ip-b1 users context1 ... The intersection between a users context and a context-aware application (ip-b1.. ip-bn) includes sensors that detect the context conditions used by applications to respond accordingly. An example of a sensor which is useful for context-aware applications is a Global Positioning System (GPS) device, which can be used to continuously track a users location. Context information exchanged in interactions with the users context consists of geographical coordinates for the users current location. Another example of a sensor is a body thermometer, which can be used to monitor a patients body temperature. Context information exchanged in interactions with the users context in this case consist of the temperature measurements in degrees Celsius. Figure 2-3 depicts the graphical representation used in this thesis, which is defined in [40, 106]. An interaction point is expressed by oval shapes that overlap with the entities that share the interaction point. Interactions taking place in interaction points ip-ai and ip-bi model the activities performed in cooperation between the user and the context-aware application, and between the users context and the context-aware application, respectively. As defined in [40, 106], an interaction models the establishment of the result, abstracting from the possible complex mechanism that lead to the establishment of the result. All possible information types referring to context that may be established in both ip-a and ip-b are defined in a context information model.

Capturing context

In our definition of context-aware application we do not distinguish between manually provided information and automatically acquired (sensed) information. We prefer to give a wider definition of context information, which encompasses a broader range of context information types, regardless whether information is sensed or manually provided. Our intention is to provide guidelines (by means of our context modelling approach) for identifying context information types, rather than fixing predefined sets of context information types, or prescribing whether information is acquired or not.

It is actually the responsibility of the application designer to decide whether context information is relevant to the application, since this decision depends on the application's universe of discourse and the application's state-of-affairs of interest. In addition, the computational capabilities (sensor technology) available for building the context-aware application typically determine the constraints for acquiring context automatically or manually. Whether manually provided or automatically acquired, we assume context information is always provided to applications through interaction points of type ip-b (see Figure 2-3).

In this sense, the definition of context information mentioned in section 2.1 given by Dey et al. [22] suffices, since it allows one to (i) abstract from whether information is manually or automatically acquired, and (ii) define a broad range of context information categories. In this thesis we rely on this definition of context information. When considering context-aware applications, context consists of possibly interrelated conditions in the real world (represented in our figures by the users context). Applications still need to quantify and capture these conditions in terms of context information in order to reason about context. Figure 2-4 shows an intuitive representation of the user's context conditions in the context-aware application by means of context information. Context information values are depicted inside the context-aware application. These values are approximations of the real world context conditions of the user's context. Only specific context conditions are relevant to the context-aware application. The process of identifying relevant conditions is part of the context modelling activities.

Figure 2-4 Intuitive view of context in the real world (user's context) and context information in a context-aware application

Situations

revisar

Modelling the context conditions in the application's universe of discourse allows application designers to represent all possible state-of-affairs in the application's universe of discourse without discriminating particular situations that may be of interest to applications. For example, while we may capture in a context model that a person may be married to another person, it is not the objective of the context model to make statements about particular instances of persons. Therefore, we do not

explicitly represent in a context model that John is married to Alice, or that John has been married to Alice for 10 years. In order to enable the representation of particular state-of-affairs, we introduce the concept of situation. Situations are defined as follows: Situation: particular state-of-affairs that is of interest to applications. A situation is a composite concept whose constituents may be (a combination of) entities and their context conditions. Situations build upon context models since they can be composed of more elementary kinds of context conditions, and in addition can be composed of existing situations themselves. Examples of situations that may be of interest to a context-aware application are user is running and he/she has access to

Quality of context

Context-aware applications depend strongly on the availability and the quality of sensors, or more generally, context sources. Context-aware applications also depend on the availability and capabilities of portable (mobile) devices that can be used to interact with the user. In the last years, sensors and devices of higher quality are proliferating due to advances in hardware and miniaturization. However, current sensors and devices are not sufficiently accurate, and they may introduce noise, delays and imperfections to the context information being sensed and/or measured. Therefore, the quality of context information is strongly dependent of the mechanisms used to capture the corresponding context conditions from the users environment. Some context conditions may have to be measured, and the measuring mechanisms may have a limited accuracy; other context conditions may vary strongly in time, so that the measurement may quickly become obsolete. Decisions based on context information taken in contextaware applications may also consider the quality of this information, and therefore context-aware applications also need meta-information about the context condition values, revealing their quality.

Figure 2-5 Diagram of the concepts related to context (real world and applications)

Figure 2-5 depicts a UML class diagram that summarizes the concepts presented so far. This model represents context in the real world (users environment) and in applications (as context information). Context always refers to an entity, and can be represented by a collection of conditions. In context-aware applications, context is referred to as context information, and its corresponding conditions are represented by condition values in the applications. In addition, these context condition values are related to some quality measures, which determine the quality of the information (e.g., how precise, accurate or fresh the information is). The figure also represents situations, which are particular compositions of entities and context conditions. Context conditions in a situation may belong to different entities. A situation is represented as situation information in context-aware applications. Although we discuss context and situation information from the point of view of condition values, context modelling can only be re-used and generalized when (i) condition and situation types, and (ii) their semantics and relationships are clearly defined. Chapter 4 focuses on context modelling issues, which also include situation modelling.

2.5 Dynamic Hypermidial Divices

2.5.1 Service-Oriented Architectures

Service-Oriented Architectures (SOAs) use an architectural style centred on the concept of service that can be applied in the design of distributed applications [77]. In this thesis, we adopt this architectural style for the design of context-aware applications. In this sense, we can say that a context-aware application provides a context-aware service to its users. In order to understand the concept of service we consider the following definition of system [79]:

2.6 Principle Design of Web E-learning Application

CHAPTER 3

Target Models and Architectures

3.1 Introduction

This chapter describe the fundamentals aspect an decisions

3.2 Models

3.2.1 Techniques for modeling context

Key-Value Models

Markup Sheme Models

3.2.2 Graphical Models

Object Oriented Models

3.2.3 Logic Based Models

3.2.4 Ontology Based Models

3.2.5 Our Technique to Model Context

Context Structure Model (CSM)

As learner contexts are variant concepts that do not share a common closed structure across different application scenarios in different organizational settings, the structure of a context needs to be split up into: (i) Universally applicable attributes. These include for example date/time and location. (ii) Application-specific attributes. These have to be defined as by requirement of the modeler's application domain, e.g. attended lectures, learning progress, etc.

The CSM defines a number of generic context classes (see Figure 2), which only include universally applicable attributes and which serve as base classes for applicationspecific context structures. Thereby, the modeler has to derive one or more custom context classes from the base structure, thus extending it to include required applicationspecific context classes and attributes. These custom context classes subsequently serve as templates for objects (i.e. context instances) modeled in the LAM. Figure 2 shows the base CSM above the dotted separator line, including a taxonomic, overlapping decomposition of Context into physical (resource, device, personal) and digital (resource, personal) context structures. The applicationspecific context class (i.e., LearnerApplicationContext below the dotted line), which serves as the context template for the example course LAM depicted in Figure 10, is derived from the base model's PersonalContext class.

Learning Activity Model (LAM)

Contextual knowledge is included in the LAM by annotating the learning activities with context objects. There are different options for considering context in learning design, namely from the process, the organizational, or the learner's point of view [19]. In our approach, the LAM shows primarily the learner's view on course activities. Nevertheless, the overall process (e.g., including the instructor's activities) needs to be co-considered to allow for constructing a complete, integrated course model. In real environments context is always implicitly, concurrently present, which raises some problems in modeling a concrete instance of relevant context [22]. In the LAM, therefore, only relevant context changes as well as the influence of a context object on the "learnflow" (and vice versa) are modeled. At an abstract modeling level, three cases are differentiated:

Activity updates context. For example, passing an exam updates the learner's context in a way that extends the learner's prior knowledge in the subject area. In such a case (generically depicted in Figure 3) a dependency relationship is drawn from the learning activity (action state) to the context object, optionally with the context update reason denoted as the dependency name.

Context alters activity. An activity is conducted with minor variations depending on some context's attributes. For example, the learner's current digital context may suggest the usage of an online learning object rather than a book on some topic. In such a case (Figure 4) a dependency relationship is drawn from the context object to the respective learning activity, optionally with a short statement about the particular influence on the learning activity depicted as the dependency name; or any constraining effect on the learning activity written in proper UML constraint form (i.e., expression enclosed in curly brackets).

Note that an immediate combination of cases (1) and (2) resembles "context flow". In this case (Figure 5), a context update is a direct consequence of a learning activity, and that updated context object directly affects the subsequent activity. This relationship mirrors the semantics of "object flow", which is defined in the UML specification [21] as substitute for a simple transition.

Context as guard. The learner context guards the firing of one or more transitions between nodes in the activity diagram. In this case, the guard condition on the respective transition refers to some context object's attribute(s) using standard UML transition syntax (i.e., condition in squared brackets). Figure 6 shows a case where a context object acts as the decisive element for multiple possible control flows after a decision node.

3.2.6 Comparison

3.3 The Architectures of DHD Systems

3.3.1 The Anatomy of a DHD Component-Connectors Architecture

An outline of the component model, which is based on Lancaster's OpenCOM [2], is illustrated in Figure 2. Components are encapsulated units of functionality and deployment that interact with other components exclusively through "interfaces" and "receptacles" (see below). Interfaces are expressed in terms of sets of operation signatures and associated datatypes; OMG IDL is used for interface specification to give language independence. (Note, however, that this does not imply the overhead of CORBA-like stubs and skeletons.) Components can support multiple interfaces: this is useful in embodying separations of concern (e.g. between base functionality and component management). Receptacles are "required" interfaces that are used to make explicit the dependencies of a component on other components: when deploying a component into a capsule, one knows by looking at its receptacles precisely which other components must be present to satisfy its dependencies. Capsules are containing entities that offer the above-mentioned runtime API. As implied above, capsules can be implemented differently on different devices—e.g. they might be implemented as a Unix or Windows process on a PDA or PC; or as a RAM chip on a sensor mote. This flexibility is key to handle the heterogeneity of the RUNES scenarios. Finally, bindings are associations between a single interface and a single receptacle. Like deployment, the creation of a binding is inherently third-party in nature. That is, it can be performed by any party within the capsule, not only by the first-party components that will

themselves participate in the binding. The component model itself is complemented by two further architecture elements: component frameworks and reflective extensions.

Interface Definition Language-3 (IDL-3) [CCM99] is used to define components and interfaces in the software architecture because it provides support for explicit dependency management through provides and uses interfaces. These explicit dependencies are used to help generate the configuration graph of components and connectors. IDL-3 emits and consumes events [CCM99] are used to specify adaptation events. Connectors are implemented as typed objects relating provides and uses interfaces, i.e. ports, on components. Connectors are automatically generated from IDL-3 component definitions by specialising and templating abstract connectors in the KComponent framework. Connectors provide a reconfiguration interface, with operations such as link component and unlink component, and the configuration manager uses this interface to implement its graph rewrite operations.

Context-Aware Support

In human-human interaction, a great deal of information is conveyed without explicit communication. Gestures, facial expressions, relationship to other people and objects in the vicinity, and shared histories are all used as cues to assist in understanding the explicit communication. These shared cues, or context, help to facilitate grounding between participants in an interaction [4]. We define context to be any information that can be used to characterize the situation of an entity, where an entity can be a person, place, or physical or computational object. In human-computer interaction, there is very little shared context between the human and the computer. Context in human-computer interaction includes any relevant information about the entities in the interaction between the user and computer, including the user and computer themselves. Humans naturally provide context in the form of signs of frustration or confusion, for example, but computers cannot sense or use it. We define applications that use context to provide task-relevant information and/or services to a user to be context-aware. For example, a context-aware tour guide may use the user's location and interests to display relevant information to the user. The increased availability of commercial, off-the-shelf sensing technologies is making it more viable to sense context in a variety of environments. The prevalence of powerful, networked computers makes it possible to use these technologies and distribute the context to multiple applications, in a somewhat ubiquitous fashion. So what has hindered applications from making greater use of context and from being context-aware? A major problem has been the lack of uniform support for building and executing these types of applications. Most context-aware applications have been built in an ad hoc manner, heavily influenced by the underlying technology used to acquire the context. This results in a lack of generality, requiring each new application to be built from the ground up. To enable designers to easily build contextaware applications, there needs to be architectural support that provides the general mechanisms required by context. This paper describes an architecture to support contextaware applications. We provide a discussion of how context has been handled in previous work and why we would like to handle it as a generalized form of user input. In earlier work [15], we presented the concept of context widgets, which allow us to handle context in a manner analogous to user input. This paper discusses the architectural support necessary for using context widgets. We derive the requirements for the architecture by examining the differences between the uses of context and input. Next, we present our architectural solution for supporting the design and execution of context-aware applications. We demonstrate the use of the architecture through an example application, the Conference Assistant. Finally, we discuss the benefits and limitations of the architecture, based on our experiences in building context-aware applications.

DISCUSSION OF CONTEXT HANDLING We have demonstrated the importance of context in interactive computing. The reason why context is not used more often is that there is no common way to acquire and handle context. In this section, we discuss how context has previously been acquired and discuss some concepts that will allow us to handle context in the same manner as we handle user input.

Current Context Handling In general, context is handled in an improvised fashion. Application developers choose whichever technique is easiest to implement, at the expense of generality and reuse. We will now look at two common ways in which context has been handled: connecting sensor drivers directly into applications and using servers to hide sensor details. With some applications [7,14], the drivers for sensors used to detect context are directly hardwired into the applications themselves.

In this situation, application designers are forced to write code that deals with the sensor details, using whatever protocol the sensors dictate. There are two problems with this technique. The first problem is that it makes the task of building a context-aware application very burdensome, by requiring application builders to deal with the potentially complex acquisition of context. The second problem with this technique is that it does not support good software engineering practices. The technique does not enforce separation of concerns between application semantics and the low-level details of context acquisition from individual sensors. This leads to a loss of generality, making the sensors difficult to reuse in other applications and difficult to use simultaneously in multiple applications. The original Active Badge research took a slightly different approach [19]. In this work, a server was designed to poll the Active Badge sensor network and maintain current location information. Servers like this abstract the details of the sensors from the application. Applications that use these servers simply poll the servers for the context information that they collect. This technique addresses both of the problems outlined in the previous technique. It relieves application developers from the burden of dealing with the individual sensor details. The use of servers separates the application semantics from the low-level sensor details, making it easier for application designers to build contextaware applications and allowing multiple applications to use a single server. However, this technique has two additional problems. First, applications that use these servers must be proactive, requesting context information when needed via a polling mechanism. The onus is on the application to determine when there are changes to the context and when those changes are interesting. The second problem is that these servers are developed independently, for each sensor or sensor type. Each server maintains a different interface for an application to interact with. This requires the application to deal with each server in a different way, much like dealing with different sensors. This may affect an application's ability to separate application semantics from context acquisition.

Current Input Handling Ideally, we would like to handle context in the same manner as we handle user input. User interface toolkits support application designers in handling input. They provide an important abstraction to enable designers to use input without worrying about how the input was collected. This abstraction is called a widget, or an interactor. The widget abstraction provides many benefits. The widget abstraction has been used not only in standard keyboard and mouse computing, but also with pen and speech input [1], and with the unconventional input devices used in virtual reality [11]. It facilitates the separation of application semantics from low-level input handling details. For example, an application does not have to be modified if a pen is used for pointing rather than a mouse. It supports reuse by allowing multiple applications to create their own instances of a widget. It contains not only a polling mechanism but also possesses a notification, or callback, mechanism to allow applications to obtain input information as it occurs. Finally, in a given toolkit, all the widgets have a common external interface. This means that an application can treat all widgets in a similar fashion, not having to deal with differences between individual widgets. Analogy of Input Handling to Context Handling There have been previous systems which handle context in the same way that we handle input [2, 16]. These attempts used servers that support both a polling mechanism and a notification mechanism. The notification mechanism relieves an application from having to poll a server to determine when interesting changes occur. However, this previous work has suffered from the design of specialized servers, that result in the lack of a common interface across servers [2], forcing applications to deal with each server in a distinct way. This results in a minimal range of server types being used (e.g. only location [16]). Previously, we demonstrated the application of the widget abstraction to context handling [15]. We showed that context widgets provided the same benefits as GUI widgets: separation of concerns, reuse, easy access to context data through polling and notification mechanisms and a common interface. Context widgets encapsulate a single piece of context and abstract away the details of how the context is sensed. We demonstrated their utility and value through some example applications. The use of the widget abstraction is clearly a positive step towards facilitating the use of context in applications. However, there are differences in how context and user input are gathered and used, requiring a new architecture to support the context widget construct. The remainder of this paper will describe the requirements for this architecture and will describe our architectural solution.

ARCHITECTURE REQUIREMENTS Applying input handling techniques to context is necessary to help application designers build context-aware applications more easily. But, it is not sufficient. This is due to the difference in characteristics between context and user input. The important differences

are: • the source of user input is a single machine, but context can come from many, distributed machines • user input and context both require abstractions to separate the details of the sensing mechanisms, butCurrent Context Handling In general, context is handled in an improvised fashion. Application developers choose whichever technique is easiest to implement, at the expense of generality and reuse. We will now look at two common ways in which context has been handled: connecting sensor drivers directly into applications and using servers to hide sensor details. With some applications [7,14], the drivers for sensors used to detect context are directly hardwired into the applications themselves. In this situation, application designers are forced to write code that deals with the sensor details, using whatever protocol the sensors dictate. There are two problems with this technique. The first problem is that it makes the task of building a context-aware application very burdensome, by requiring application builders to deal with the potentially complex acquisition of context. The second problem with this technique is that it does not support good software engineering practices. The technique does not enforce separation of concerns between application semantics and the low-level details of context acquisition from individual sensors. This leads to a loss of generality, making the sensors difficult to reuse in other applications and difficult to use simultaneously in multiple applications. The original Active Badge research took a slightly different approach [19]. In this work, a server was designed to poll the Active Badge sensor network and maintain current location information. Servers like this abstract the details of the sensors from the application. Applications that use these servers simply poll the servers for the context information that they collect. This technique addresses both of the problems outlined in the previous technique. It relieves application developers from the burden of dealing with the individual sensor details. The use of servers separates the application semantics from the low-level sensor details, making it easier for application designers to build contextaware applications and allowing multiple applications to use a single server. However, this technique has two additional problems. First, applications that use these servers must be proactive, requesting context information when needed via a polling mechanism. The onus is on the application to determine when there are changes to the context and when those changes are interesting. The second problem is that these servers are developed independently, for each sensor or sensor type. Each server maintains a different interface for an application to interact with. This requires the application to deal with each server in a different way, much like dealing with different sensors. This may affect an application's ability to separate application semantics from context acquisition. Current Input Handling Ideally, we would like to handle context in the same manner as we handle user input. User interface toolkits support application designers in handling input. They provide an important abstraction to enable designers to use input without worrying about how the input was collected. This abstraction is called a widget, or an interactor. The widget abstraction provides many benefits. The widget abstraction has been used not only in standard keyboard and mouse computing, but also with pen and speech input [1], and with the unconventional input devices used in virtual reality [11]. It facilitates the separation of application semantics from low-level input handling details. For example, an application does not have to be modified if a pen is used for pointing rather than a mouse. It supports reuse by allowing multiple applications to create their own instances of a widget. It contains not only a polling mechanism but also possesses a notification, or callback, mechanism to allow applications to obtain input information as it occurs. Finally, in a given toolkit, all the widgets have a common external interface. This means that an application can treat all widgets in a similar fashion, not having to deal with differences between individual widgets. Analogy of Input Handling to Context Handling There have been previous systems which handle context in the same way that we handle input [2, 16]. These attempts used servers that support both a polling mechanism and a notification mechanism. The notification mechanism relieves an application from having to poll a server to determine when interesting changes occur. However, this previous work has suffered from the design of specialized servers, that result in the lack of a common interface across servers [2], forcing applications to deal with each server in a distinct way. This results in a minimal range of server types being used (e.g. only location [16]). Previously, we demonstrated the application of the widget abstraction to context handling [15]. We showed that context widgets provided the same benefits as GUI widgets: separation of concerns, reuse, easy access to context data through polling and notification mechanisms and a common interface. Context widgets encapsulate a single piece of context and abstract away the details of how the context is sensed. We demonstrated their utility and value through some example applications. The use of the widget abstraction is clearly a positive step towards facilitating the use of context in applications. However, there are differences in how context and user input are gathered and used, requiring a new architecture to support the context widget construct. The remainder of this paper will describe the requirements for this architecture and

will describe our architectural solution.

ARCHITECTURE REQUIREMENTS Applying input handling techniques to context is necessary to help application designers build context-aware applications more easily. But, it is not sufficient. This is due to the difference in characteristics between context and user input. The important differences are: • the source of user input is a single machine, but context can come from many, distributed machines • user input and context both require abstractions to separate the details of the sensing mechanisms, but

context requires additional abstractions because it is often not in the form required by an application • widgets that obtain user input belong to the application that instantiated them, but widgets that obtain context are independent from the applications that use them While there are some applications that use user input that have similar characteristics to context, (groupware [12] and virtual environments [5] deal with distributed input and user modeling techniques [8] abstract input, for example), they are not the norm. Because of the differences between input and context, unique architectural support is required for handling context and context widgets. We will now derive the requirements for this architecture.

DESCRIPTION OF ARCHITECTURE Our architecture was designed to address the requirements from the previous section. We used an object-oriented approach in designing the architecture. The architecture consists of three main types of objects: • Widget, implements the widget abstraction • Server, responsible for aggregation of context • Interpreter, responsible for interpretation of context Figure 1. Relationship between applications and the context architecture. Arrows indicate data flow. Figure 1 shows the relationship between the objects and an application. Each of these objects is autonomous in execution. They are instantiated independently of each other and execute in their own threads, supporting our requirement for independence. These objects can be instantiated all on a single computing device or on multiple computing devices. Although our base implementation is written in Java, the mechanisms used are programming language independent, allowing implementations in other languages as we will describe later. It is important to note that the architecture provides scaffolding for context-aware computing. By this we mean that it contains important abstractions and mechanisms for dealing with context, but it is clearly not a complete solution, nor is it meant to be. The architecture supports the building of widgets and interpreters required by an application, but will not necessarily have them already available. Compared to input, there are a larger variety of possible sensors used to sense context and a larger variety of possible context. This makes it very difficult to provide all possible combinations of widgets and interpreters. Context Widgets A context widget, as mentioned earlier, has much in common with a user interface widget. It is defined by its attributes and callbacks. Attributes are pieces of context that it makes available to other components via polling or subscribing. Callbacks represent the types of events that the widget can use to notify subscribing components. Other components can query the widget's attributes and callbacks, so they don't have to know the widget capabilities at design time. A context widget supports both the polling and notification mechanisms to allow components to retrieve current context information. It also allows components to retrieve historical context information. The basic Widget object provides these services automatically for context widgets that subclass it. Creating a new widget is very simple. A widget designer has to specify what attributes and callbacks the widget has, provide the code to communicate with the sensor being used, and when new data from the sensor is available, call 2 methods: sendToSubscribers()and store(). The Widget class provides both of these methods. The first method validates the data against the current subscriptions. Each time it finds a match, it sends the relevant data to the subscribing component. For example, multiple applications have subscribed to a Meeting Widget with different callbacks, attributes, and conditions. When the widget obtains new meeting information it sends it to the appropriate subscribers. The second method adds the data to persistent storage, allowing other components to retrieve historical context information. This addresses our requirement for the storage of context history. The Widget class provides a default implementation for persistent storage using MySQL, a freeware database. The persistent storage mechanism is "pluggable". A widget designer not wanting to use the default mechanism can provide a class that implements a temporary cache and allows the storage and retrieval of information from some persistent storage. The name of the class is given to the widget at run time, allowing the new storage mechanism to be used. Context Servers Context servers implement the aggregation abstraction, which is one of our requirements. They are used to

collect all the context about a particular entity, such as a person, for example. They were created to ease the job of an application programmer. Instead of being forced to subscribe to every widget that could provide information about a person of interest, the application can simply communicate with a single object, that person's context server. The context server is responsible for subscribing to every widget of interest, and acts as a proxy to the application.

The Server class is subclassed from the Widget class, inheriting all the methods and properties of widgets. It can be thought of, then, as a compound widget. Just like widgets, it has attributes and callbacks, it can be subscribed to and polled, and its history can be retrieved. It differs in how the attributes and callbacks are determined. A server's attributes and callbacks are "inherited" from the widgets to which it has subscribed. When a server receives new data, it behaves like a widget and calls store() and sendToSubscribers(). When a designer creates a new server, she simply has to provide the names of the widgets to subscribe to. In addition, she can provide any attributes or callbacks in addition to those of the widgets and a Conditions object. The Conditions object is used in each widget subscription, so the server only receives information it is interested in. For example, the Anind User Server would have the subscription condition that the name must equal "Anind".

Run-time Support

Components can be deployed at any time during run-time, and their loading can be requested from within any component within the capsule (this is called third-party deployment). An abstract of the runtime API offered by each capsule is as follows: template load(comp type name); comp inst instantiate(template t); status unload(template t); comp inst bind(ipnt inst interface, ipnt inst receptacle); status destroy(comp inst comp); status putattr(ID entity, ID key, any value); any getattr(ID entity, ID key); Load() loads a named component "template" from a local repository into the capsule, and unload() unloads a template. Templates can be instantiated (using instantiate()) to yield component instances (comp insts); this can be done multiple

Talk about architecture

Adaptability is one of the most important requirements for DHD, since such environments are highly dynamic, characterized by frequent and unpredictable changes in different contexts. Hence, applications need to be capable of adapting their behavior to ensure they continue to offer the best possible level of tool's service to the user. Adaptation should be driven by awareness of a wide range of issues including communication performance, link reconfiguration (seccion **??**), navegation, presentation, tools integrationt, and application preference.

The current approach to providing adaptable services or applications is based upon the classic layered architectural model where adaptation is provided at the various layers (data link, network, transport or application layers) in isolation [3]. One of the main limitations of current approaches is that applications themeselves are responsible for triggering and adaptive mechanism when the underling infrastructure notifies them about any changes [4]. Hence, we need a sophisticated approach, which can manipulate mixed or customized adaptation in contextual changes. It is more desirable and effective

Coordination Support

Service Support

3.3.2 Connector View

A connector view of an architecture concentrates on the mechanics of the communication between components. For a REST-based architecture, we are particularly interested in the constraints that define the generic resource interface. Client connectors examine the resource identifier in order to select an appropriate communication mechanism for each request. For example, a client may be configured to connect to a specific proxy component, perhaps one acting as an annotation filter, when the identifier indicates that it is a local resource. Likewise, a client can be configured to reject requests for some subset of identifiers. Although the Web's primary transfer protocol is HTTP, the architecture includes seamless access to resources that originate on many pre-existing network servers, including FTP [Postel and Reynolds 1985], Gopher [Anklesaria et al. 1993], andWAIS [Davis et al. 1990]. However, interaction with these services is restricted to the semantics of a REST connector. This constraint sacrifices some of the advantages of other architectures, such as the stateful interaction of a relevance feedback protocol like WAIS, in order to retain the advantages of a single, generic interface for connector semantics. This

generic interface makes it possible to access a multitude of services through a single proxy connection. If an application needs the additional capabilities of another architecture, it can implement and invoke those capabilities as a separate system running in parallel, similar to how the Web architecture interfaces with "telnet" and "mailto" resources.

3.3.3 Details of the Role of Connection in DHD Architecture

3.3.4 Connectors in Adaptive Environments

3.3.5 Dynamic Aspects of the DHD Architecture

Separating Adaptation Code from Computational Code

Role of Architecture in Run Time DHD System Reconfiguration

Implicit Adaptation

Explicit Adaptat

Run Time Modifications in DHD Architectures

3.3.6 Architectures Assumtions on COTS

3.4 Dynamic context aware Web Applications

3.4.1 Providing e-learning transactional behavior to services coordination

- *Application logic* must be captured by using a co-ordination approach (i.e. workow technology).
- *Transactional behavior* must be defined indepen- dently of the application logic by using atomicity contracts and associating a well defined behavior to coordination participants. For example the ac- tivities bank authorization and process order of the e-commerce application (see Figure 1) need to be treated as an execution unit that must be atomically executed because according to the ap- plication, orders can be processed only if the pay- ment has been authorised by the bank.

3.4.2 Elrn-transactional behavior model

1. Activity. It abstracts a service method call. It can be classified according to its behavior on failure [16] by three scenarios: (1) the side efects that can be caused by undoing the activity (e.g. an extra charge for cancelling the activity debit account), (2) the possibility or not of undoing an activity (e.g. the activity ship item cannot be un- done when the order contains underclothes), and (3) the possibility of trying several times an activ- ity (e.g. send invoice cannot be retried because the invoice cannot be expedited several times). The behavior of an activity depends on the ap- plication semantics. In our example, the activity bank authorization is vital because no shipment will be authorized without the corresponding payment.
2. Sphere. It groups a set of activities. It has an associated state, behavior and contract. The notion of sphere is used for modeling atomic properties for execution paths in workows.
3. Contract. It specifies the transactional behavior of a sphere and a reaction in case of failure. The behavior is related to well known transactional requirements (e.g. atomicity, isolation, durability, etc.).

3.4.3 Execution engine

We have specified the general architecture of an exe- cution engine that consists of two main modules:

1. *Behavior evaluator* detects execution failures and generates actions for ensuring the required trans- actional behavior according to given contracts for a target application.

2. *Coordination engine* enacts the coordination spec- ification. We are not interested in proposing a new engine, we define components that can inter- act with existing engines (i.e. transactional man- agers, security controllers, etc.).

Components and Component Frameworks

Reference scenario

3.4.4 Reflective Extensions

Reflective Extensions The essence of "reflection" is to establish and manipulate causally-connected meta-models of an underlying target system [3]. Such meta-models are representations of some aspect or view of the target system, and they expose a so-called meta-interface through which the representation can be inspected and manipulated. The main purpose of reflection is to maintain an architectural separation of concerns between system building or configuration (sometimes called base-level programming), and system adaptation or reconfiguration (sometimes called meta-programming). Note that there is a potentially-confusing terminological clash here between the "meta-models" used when modeling applications and "reflective meta-model" used in when working with computational reflection. These two concepts are entirely distinct;We will use the term reflective extension instead of reflective meta-model to avoid confusion with UML meta-models. Reflection is a powerful technique, and its use should ideally be constrained to minimize programmer errors. Our approach is to deploy reflective extensions in close association with CFs. The idea is that CFs can encapsulate metainterfaces, and appropriately restrict access to them according to policy. Furthermore, such encapsulation can also

Architecture reflective extension

Interception reflective extension

Interface reflective

CHAPTER 4

Models and Architectures

4.1 Introduction

This chapter describe the fundamentals aspecto an dicisions

4.2 Models

4.2.1 Context Model

Key-Value Models

Markup Sheme Models

4.2.2 Graphical Models

Object Oriented Models

4.2.3 Logic Based Models

4.2.4 Ontology Based Models

4.3 Architectures

4.3.1 Model Drives

4.3.2 Dynamic Architecture

Role of Architecture in Run time System Reconfiguration

Run Time Modifications in Architectures

4.4 Connectors

4.4.1 The Role of Connection in Architecture

4.4.2 Connectors in Adaptive Environment

4.4.3 Connectors in Adaptive Environment

4.4.4 Dynamic Connectors

4.5 Dynamic context aware Web Applications

CHAPTER 5

Context Aware Contract

5.1 Contract Definition: Toward

In order to achieve software reliability, Design by Contract advocates that methods and classes can have assertions [Hoare69], namely:

- method preconditions: assertions that must be true prior to a method execution; - method postconditions: assertions that must be true as a result of a method execution; - class invariants: assertions that must be always held during the lifetime of the class objects.

A classic analogy to explain Design by Contract is expressed through the responsibilities expressed by a business contract. In a business, there are (at least) two interested parties: the client and the supplier2. In order to be able to provide a service properly, the supplier expects that the client fulfils some stated obligations: the preconditions (an obligation to the client, but a benefit to the supplier). On the other hand, at the end of the service, the client expects to benefit from some stated results, achieved by the service provided by the supplier: the postconditions (an obligation to the supplier, but a benefit to the client). Additionally, the contract can have some clauses that specify certain conditions that both parties must guarantee throughout the duration of the service: the invariants. The pre/postcondition dichotomy is summarized in Table 4.1.

Furthermore, according to Liskov's Substitution Principle [Liskov93], if a class B is a subtype of A, then in any situation that an instance of A can be used, it can replaced by an instance of B. To achieve this in a contract, the concept of contract inheritance must be introduced. As such, preconditions are contravariant and postconditions are covariant [Liskov99] [Castagna95]. This means that when extending a class, it is only possible to maintain or weaken preconditions (with less restrictive preconditions) and maintain or strengthen postconditions (with more restrictive postconditions).

Invariants must be held for all classes in the hierarchy. Using a "real-life" example, imagine that a person is using a mail service to send a package. In order to do so, a contract must exist between the person (client) and the mail service (supplier). If the person wants to send a package, it must have a correct stamp, a correct receiver address and is accepted only through 8h to 17h (precondition); on the other hand the person is expecting that the package will be delivered in two days (postcondition). Both accept that the service will only be provided in working days, not weekends (invariant). The previous example can be expanded to explain contract inheritance. In this example, inheritance is represented through work delegation. In a mail service, the work is divided through several departments. There are two departments, one that accepts packages and another that delivers them. While delegating the package from the first department to the second, it is not a problem to weaken the precondition (e.g. the package can be sent until 20h), but it is not acceptable to strengthen it (e.g. demanding that the address is written in black ink). On the other hand, it is acceptable to strengthen the postcondition (e.g. delivering the package in the same day), but not weaken it (e.g. delivering the package in five days). Further invariants can apply (e.g. the package must have valid transport

5.1.1 Design by contract Framework

Existing Approaches to Contract Support

All approaches to contract support require programmers to explicitly specify the semantic interface for each class. This interface, informally called the contract, consists of assertions that denote the invariant property for each class, the preconditions for each method, and the postconditions for each method. A number of programming languages allow programmers to specify the semantic interface using keywords provided by the programming language. These include Euclid [13], Alphard [21], Anna [14] , SR 0, D [5], and recently Python [6] and Java [12] have been extended. Unfortuately, most of the languages that do support contracts are not widely-used. While built-in support is the easiest and most elegant approach from a programmer's perspective, to so drastically change an existing programming language requires time, effort, and resources for rewriting the compiler. The fact that so few languages include it is evidence of significant effort needed. Since contract support is not part of the most commonly-used languages, four categories of approaches have been taken to implementing contracts in languages that do not support them intrinsically: 1) Code Libraries, 2) Preprocessor, 3) Behind-the-Scenes, and 4) Programming Language-Independent.

The Code Library One approach is to create libraries of functions needed to support contracts and prescribe conventions for calling these functions. In general, it is the programmer's responsibility to call these functions in the appropriate places and to provide at least some of the exception handling. One such approach was created by McFarlane for the C language [15]. Another more sophisticated library approach is that used by the creators of jContractor for Java [9] [25]. Programmers follow a naming convention to write contracts as methods in each class definition. jContractor identifies these patterns and uses the Java Reflection API to synthesize the original code with contract-checking code. jContractor can add the contract-checking code to the bytecode or as the classes are loaded. Some scholars categorize jContractor as a Behind-the-Scenes (metaprogramming/reflective) approach because of its use of the Java Reflection API [19]. Library approaches have the advantage of allowing programmers to use standard language syntax—and they work with any implementation of the language. No special specification language is needed for the assertions, and the naming conventions or function names are intuitive. No changes to the development, test, or deployment environment are needed, and no special tools such as modified compilers, runtime systems, or preprocessors are required. Library techniques, however, tend to be burdensome for programmers to use. Programmers must either actually write methods for each assertion while follow naming conventions or they must remember to call functions in certain places. In addition, for McFarlane's approach, inherited assertions must be handled by the programmer. Another problem may be the lack of integration with pre-existing or subsequently changed libraries. Furthermore, this approach does not take into account the situation in which a single program contains multiple instances of an object, which must be distinguished at runtime for trace purposes.

Preprocessor Approaches. A common approach to implementing contracts is the use of a preprocessor to transform code containing formal comments, macros, or keywords into compilable code with contracts. The iContract tool for the Java programming language uses just such an approach [11]. Programmers add contracts as JavaDoc comments written in a special specification language. The iContract tool converts these comments into assertion-checking code. Another example of a preprocessor approach is Jass (Java with Assertions) [24]. Programmers specify contracts using the extended language Jass and use a preprocessor to translate the extensions to ordinary Java. Preprocessor approaches have a number of advantages. First and foremost, the compiler need not be rewritten. Second, the resulting code is compatible with any implementation of its particular language. Third, because contracts and code are written simultaneously, contracts are part of the design and implementation phases. In addition, preprocessor approaches often allow multiple options for regulating the degree of the translation, and they need not be run at all. Furthermore, the developer has complete control of the code and can make changes to the processed code if they wish to be involved with the details of contract implementation. Several disadvantages to preprocessor approaches are also commonly cited. First, because the original source code is changed, the line numbers of compiler errors, debugging output, and exceptions do not correspond with line numbers of the original program [19]. The contract-instrumented code is also more difficult to read because of the extra code added to enforce contracts. In addition,

programmers must learn the particular specification language for expressing the assertions. As others have pointed out, the programmer is unable to add new ways of handling violations to the assertions at runtime and unable to change the level of contract checking at runtime [2][22]. Some consider the use of an external preprocessor to be burdensome. In addition, the preprocessor approach is also not appropriate for real time systems, where the runtime for a debug build differs from that of a release build.

Code translation, a variation of the preprocessor approach, involves translating code with contracts written in a language that supports DBC into the code of a language that does not support it. This method is used by Plösch for implementing DBC in C++ based on Python [18]. This approach has some distinct disadvantages. Using this approach requires that all programmers learn both languages and development environments, which is inefficient and more expensive. Also, numerous technical problems arise during the transformation process [2].

Behind-the-Scenes Approaches. Another category of approaches makes use of DLLs to incorporate the contract-checking code into the source code at load time, in a behind-the-scenes manner, so that the programmer never sees any contract-checking code. Two tools make use of this approach for implementing contracts in the Java programming language: JMSAssert and HandShake. JMSAssert [7] uses a preprocessor similar to iContract, in which programmers write the contracts using special tags in JavaDoc comments. However, it also incorporates a dynamically linked library or "assertion runtime." The preprocessor is used to process the source code, resulting in contract files written in JMScript (a proprietary scripting language). A special extension DLL, containing the JMScript interpreter, enforces the contracts by executing the "triggers" contained within these JMScript files. A benefit to this approach is that only the contract tags and assertions are visible because contracts are enforced "behind-the-scenes." JMSAssert does have its drawbacks, however. First, programmers must learn and be disciplined enough to use the tags for expressing the Boolean assertions. Second, when implementing interfaces or inherited classes, programmers must ensure that inheritied preconditions are only weakened and post conditions are strengthened. Third, JMScript is a proprietary language, and JMSAssert is apparently compatible only with JDK 1.2. Compatibility may be a concern with future versions. JMSAssert also requires some changes to the development environment. The HandShake tool requires that the programmer create a separate "contract file" associated with each class and interface [8]. A preprocessor, the "HandShake compiler," converts the contract file to a binary file. At load time, a special DLL called the HandShake library merges the contract binary and the original file to create a class file with contracts. The HandShake tool has the advantage that contracts can be added without modifying their source code; however, requiring that contracts for each class be written in a separate contract file does not encourage the design and development of code in unison with contracts. In addition, the task of creating a separate file, written in a special syntax, is time-consuming and tedious for programmers. Furthermore, the HandShake Library may not be supported on all platforms, since it is a non-java system [9].

Programming Language-Independent Approaches. Recently with the release of Microsoft's .NET development environment, the idea of using a programming language-independent method for implementing contracts has become more prominent. Some of these proposals sound promising [22] [25]; however, the only tool currently available is the Contract Wizard [2], which reads a .NET Assembly as its input. It uses the Eiffel compiler to access the appropriate information from the .NET metadata and provides fields for the user to input assertions. It then generates Eiffel classes containing the specified contracts, and using the Eiffel compiler, generates another .NET assembly—this time containing contracts. On the plus side, the source code contains no extra code for implementing the contracts, the programmer does not need to use special tags to specify the contracts, and the GUI for adding contracts is easy to use. More importantly, it can be used for any language supported by .NET. The Contract Wizard does, however, have several disadvantages. It requires some type of Eiffel development environment—in addition to an environment for whatever language is being used to write the source code. The main disadvantage to the Contract Wizard is that contracts cannot be specified during the design or even the implementation phase. Adding contracts is done in a separate phase after implementation is complete [22]. It is not clear what happens to the contracts when the programmer goes back to modify the original language source code [2]. It appears that the contracts would have to be recreated. As a result, contracts would be added as late as possible in the development cycle. Despite

its disadvantages, this "late-binding" type of approach makes changes easier.

5.1.2 The Contract as Connector

Most software technologies support component-based design. Generally, this support exists only in the sense that components are syntactic modules (black boxes showing only their interfaces): systems can be readily constructed from components that are designed and implemented by other parties. Components can be compiled separately, and they can be composed at compiling time or at run time. One problem with current component-based software engineering (CBSE) practice is that the component interfaces are generally just a syntactic definition and not behavioral modules. Therefore, normal situations such as software systems developed by teams, the replacement of individual parts of a software application, and the component software market, require the definition of the behavior, interaction and coordination among components. However, such information cannot be extracted from the signature description of the component (the so-called interface). In addition, there are many problems to be considered when designing and implementing components for open systems where the life of a component is usually shorter than the life of the application and some components could be inserted, modified and dropped during the lifetime of the application. Furthermore, these components may be developed by different teams, technologies, platforms, etc. Therefore, such problems should be dealt with if we want to have a component market where developers only need plug and play components. Frequently, the component designer keeps in mind what the component does (its functionality) but he/she knows neither the users nor the application where it will be used. So, it is suitable to have a clear separation between the functional aspects of the components and other requirements such as synchronization, coordination, persistence, replication, distribution, real time, etc., specific to the application scenario where the components will be used and known by the whole system designer. Allen and Garlan [1], when modeling software architectures, distinguish between the implementation and interaction relationships of software modules or components: 'Whereas the implementation relationship is concerned with how a component achieves its computation, the interaction relationship is used to understand how that computation is combined with

others in the overall system' [1]. They propose a formal model for software design that makes the interaction relationships between components explicit by using the abstraction of a connector. Describing software architectures in terms of interaction relationships among components brings us closer to a compositional view, and hence a more flexible or open view of an application. Thus, our work proposes the connectors as a way of viewing an application's architecture as a composition of independent components. Connectors support better flexibility, since each component could engage in a number of different agreements, thereby increasing the potential reuse of individual components. Separating connectors from components also promotes reuse and refinement of typical interaction relationships, allowing the refinement of connectors and the construction of complex connectors out of simpler ones. But interaction relationships are rarely captured by programming language constructs as method calls. In this sense, traditional object-oriented languages provide little support for explicit representation of software architecture. Type hierarchies are the only design elements explicitly visible at implementation level—but they represent inheritance relationships, and do not reflect an application's architecture. In contrast, interaction relationships, such as coordination and synchronization of a group of objects collaborating to achieve a task, manifest themselves as patterns of message exchanges. Such patterns of communication have a logical and conceptual identity at design level but this identity is lost when we move from design to implementation because the information about such collaborations is spread out among the participating objects and the method calls. The loss of this information makes the resulting application opaque with respect to its architecture, difficult to understand, to reuse and to reengineering. The first step toward more explicit application architecture at code level is to enable the location of information about interactions inside the application's code. In this paper we propose the following solution: enriching component interfaces with contracts. These contracts are formed by logical asserts and the behavior we want when the asserts are not satisfied. Besides this methodological proposal, we have developed a tool for generating automatically the connectors. These connectors represent the interaction relationships between components. The main contribution of this paper is to provide connectors at implementation level that are able to provide client requirements using contracts and able to solve run-time adaptation problem using semantic web techniques. Therefore, connectors will be run-time entities that not only describe, but also actually control inter-component communication and behavior against unexpected situations. The connectors adapt non-functional requirements, statically,

in the composition step when defining contracts and on-failure statements for connectors and carry out a dynamic adaptation when a service is invoked and it is not found in the requested component.

The State of the art

One characteristic of progress in programming languages and tools has been the regular increases in the level of abstraction or the conceptual size of software design building blocks. To obtain this into perspective, let us begin by looking at the historical development of abstraction techniques in computer science. The design and evolution of programming languages is one of the most important areas of computer science.

Programming languages define the manner in which we communicate with our machines. They give us layers of abstraction with which to work, so we can accomplish our tasks without reaching into the hardware. They also attempt to increase our efficiency by automating many hardware bound tasks. Anyone who has attempted to write a large project in an assembly language understands the necessity for higher level languages.

Over the last 50 years or so, languages have continued to evolve in order to support their ever increasing usage. Program design and maintenance became issues with the dawn of software engineering. People sought to formalize methods for constructing correct, efficient and easily modified programs. Languages evolved in order to support these new requirements. Block structure came into existence from a desire for modularity. Object-oriented programming (OOP) was created from a desire to have language constructs for modeling real-world objects and encouraging software reuse. OOP has been around for more than a decade now, and has become the default paradigm in many peoples' minds.

Object-orientation still presents some problems. Although it encourages software reuse, practical experience has shown that OOP does not handle this as effectively as people originally thought. Prepackaged software often does not suit the programmer's needs, and ill-constructed interfaces make using these packages difficult. Furthermore, OOP should enhance maintainability by causing redesign to affect as few modules/classes as possible. However, as our programs continue to get larger and larger it becomes increasingly difficult to cleanly separate concerns into modules. The treatment of these situations has brought about the necessity of new models because of the traditional programming has been unable to treat them in a natural way. This way, OOP has been the base of software engineering for non-open systems. However, OOP has not been useful enough when designing and developing open systems. OOP does not allow users to express clearly the distinction among computational or functional aspects and compositional and non-functional aspects in a software application. Besides, doing the concept of object to prevail over the concept of component. In addition, OOP does not keep in mind the market factors such as time to market or the change of the requirements that are highly necessary in the real world, as well as the distributed features of the real applications and the possibility of getting component software and putting it into our system. Aspect-oriented programming (AOP) has been first introduced by Kickzales [3] in 1997. Like structured programming and OOP introduced a new approach to design programs and a set of guidelines to make more readable and reusable code, AOP is a philosophy that is related to the style of programming. It addresses issues that can be solved in other approaches, but in a more elegant way. To understand the spirit of AOP, you must understand the following issues. (i) Separation of concerns: referring the Law of Demeter, a project's efficiency increases if all the concerns are well modularized and if you only have to speak to your direct friends to make a modification of the program (this is a very old principle and the OOP gives some answers). (ii) Crosscutting: in a complex system, there are always some concerns that are not easily modularized, especially common-interest concerns that, by essence, are used by several modules (i.e. several modules use/ share a well-known service of a module—such as a logging or a persistence service). (iii) Dependencies inversion: the best way to avoid crosscutting is to NOT use the well-known services that crosscut. This is possible by reversing the dependencies (i.e. the well-known service shall use the other modules instead of the contrary). This dependency inversion is implemented by aspects. AOP furnishes a set of programming concepts, which allows the user to implement the dependency inversion in a clean way. Model-driven architecture (MDA) [4] is a framework for software development, made by the object management group. Key to MDA is the importance of models in the software development process. Within MDA, the software development process is driven by the activity of modeling your software system. The MDA process defines the following three steps. (i) Build a model with a high level of abstraction, that is independent of any implementation technology. This is called a platform-independent model (PIM). (ii) The PIM is transformed into one or more platformspecific

models (PSMs). A PSM is tailored to specify your system in terms of the implementation constructs that are available in one specific implementation technology, for example, a database model, an EJB model. (iii) Transform a PSM to code. Because a PSM fits its technology very closely, this transformation is rather trivial. The complex step is the one in which a PIM is transformed to a PSM.

Component-oriented programming was born, starting from those ideas, as a natural extension of the object orientation for open systems. This new programming paradigm proposes the development and utilization of reusable components inside a new environment such as, for example, a global market of software components. However, having components is not enough unless we are able to reuse them. And, reusing a component does not only mean using it more than once, it means being able to use it in different contexts. In this way, the software engineer

community has always dreamt of having a global market of software components. To obtain a real global market of software components, it is necessary that components are packed in such a way as to allow an easy distribution and their composition with other components, especially, components developed by other parts, called commercial off-the-shelf (COTS). The concept of COTS is changing the traditional idea of software development where everything is private, to new software development methodologies where it is possible to take external elements and put them into our application. These new methodologies are changing the way we design and develop our systems. Allen and Garlan [1] propose a formal model for software design that makes the interaction relationships between components explicit using the abstraction of connector. Describing software architectures in terms of interaction relationships between components brings us closer to a compositional view, and hence a more flexible or open view of an application. In this scenario, connectors allow us to view the application's architecture as a composition of components. This paper presents one implementation and methodology for the Allen and Garlan software architecture model as well as a tool supporting the methodology proposed. This implementation has been improved with contracts, dynamical adaptation and connector subtyping. In addition, there is another way to deal with the problem. Besides adapting the components, we recommended adapting the connectors and developing new connectors as a subtype of a previous one. This feature opens the door to a new framework where a developer can find 'generic connectors' implementing classical coordination protocols, then using such connectors to coordinate their components or as a base for the development of new connectors. 2. THE NOTION OF CONNECTOR The CBSE is concerned with the development of systems from reusable parts (components), the development of components and also system maintenance and improvement by means of component replacement or customization. Building systems from components and building components for different systems requires established methodologies and processes not only in relation to development/maintenance phases, but also in relation to the entire component and system life-cycle including organizational, marketing, legal and other aspects. This work proposes an implementation of the [1] software architecture model. This implementation has been improved with contracts, dynamical adaptation and connector subtyping. The main idea of the approach presented lies in the separation of the roles. This way, the component designer will be the person taking care of the component functionality and the system designer will be the person in charge

of making the system work properly. The system designer (a client of the component) takes a set of components that have a well-known functionality and he/she builds the system interconnecting those components by means of connectors where he/she is able to express the semantics of the system in terms of a set of contracts. As it shown in Fig. 1, the system designer will define the contracts for the component behavior and could provide one ontology of the domain of the component. The ontology will be translated to prolog. The prolog will be translated to C and joined to the contracts to generate the connector. 2.1. Using web services as connectors Nowadays, Web services are an important emerging technology for software component development and integration. The paper [5] defines a Web service as a software system designed to support interoperable machine-to-machine interaction over the Internet. It has an interface described in a machine-processable format (e.g. WSDL). Other systems interact with the Web service in the way prescribed by its description using SOAP messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards. WSDL describes the static interface of a Web Service. It defines the protocol and the message features of end points. The goal of Web Services is to automate and to make business process collaborations easier both inside and outside enterprise boundaries. Useful business applications of Web services in business to business, business to consumer and enterprise application integration environments will require the ability to carry out complex and distributed Web service integrations and the ability to describe the relationships between the constituent low-level services. WSDL describes only the 'static interface' of a Web service.

So, several questions arise when using and reusing components implemented using Web services in applications: (i) How can you trust a component? (ii) What if the component behaves unexpectedly, either because it is faulty or simply because you misused it?

Therefore, it is necessary to determine whether a component can be used within a certain context. We would like a specification that tells us what the component does without entering into the details of how it does it. Furthermore, such a specification should provide parameters in order that the component may be verified and validated, and thus providing a kind of contract between the component and its users enriching the WSLD. Components are mainly for reuse. One of the essential tasks in the component-based software development process is finding the right component providing the functionality and interface expected by component clients. Once the right components are identified, a connector is used to adapt the behavior of components without modifying the components themselves. A connector is an external component that sits between the component client and the component server and mediates the mismatch between them. Hence, the connector mechanism does not try to change any internal part of the component. Therefore, a connector is a component for managing, connecting and adapting the behavior of one (or more) server component for any client. A client component wanting to get some services from a server component has two possibilities: it can either use a server using a given behavior only using a known connector, or it is possible to obtain a different behavior by developing a new connector. A connector tests the behavior of a server and schedules which requests will be served, when they will be served and which server can attend the reclaimed service (in the case of a connector having more than one server attached). All the constraints set for the server are established as contracts (preconditions, postconditions and invariant) and represented by logical expression. The connector tests the invariant and preconditions for each service before calling it and tests postconditions and the invariant when the server returns. The connector can also execute some defined actions when the contract fails. This way, a client is able to set how he/she wants the server to behave and he/she is also able to set what should be done when the requirements fail. Our proposal starts from the server features definition (its interface), which is extended to define server location, requirements and behaviors. Connectors have the following features. (i) Connectors are defined by clients. (ii) Connectors use contracts to express the properties expected by the client. (iii) Connectors are Web services that will be automatically generated. This is important because it allows a connector can be used by any client component running in any host. The tool COMPOSITOR (COMPOnent connectorS Ipso facto generaTOR) takes the extended component interface as the starting point for connector generation.

Connectors allow clients to work with one or more different servers having the same behavior. (v) Connectors allow clients to monopolize a server using the predefined predicates Available() or Available(name) as part of a precondition. (vi) Connectors are able to be enriched statically and dynamically with one ontology of the server domain (OWL syntax) in order to adapt requests on the domain context using semantic web techniques, Prolog and reection. This mechanism is called run-time adaptation (see Section 5). (vii) Connectors allow subtyping (see Section 4). As mention previously, connectors will be implemented as Web services. But, there is no supposition about the implementation of the rest of the components in the system (Fig. 2).

5.2 Context Aware Contract

5.2.1 Elements of Context Aware Contract

En este apartado, expondremos los diferentes tipos de componentes para capturar contexto que integran un modelo context-aware. Particularmente, mencionaremos uno de los más tradicionales expuesto en la tesis doctoral de Dey, A. (2000). La tabla 1 muestra una clasificación de arquitecturas en las que se resaltan las principales características.

Arquitectura Características Acceso directo a sensores: No extensible Componentes altamente acopladas Basado en Middleware: Permite la ocultación de detalles de censado de bajo nivel. Extensible. Servidor de contexto: Permite el acceso de múltiples clientes a los datos de forma remota. Libera a los clientes de recursos que demandan operaciones intensivas. Debe considerar el uso de protocolos apropiados, rendimiento de la red, calidad de lo parámetros de los servicios. Widgets: Encapsulamiento. Intercambiable. Controlado a través de un manejador de widget. En el caso que las componentes estén acopladas, incrementan la eficiencia al costo de perder robustez ante fallas de componentes. Networked services: Similar a la arquitectura servidor de contexto. (modelo orientado

a objeto) No tiene la eficiencia de la arquitectura widget debido a la complejidad de las componentes bases de red, pero provee robustez. Blackboard model: Procesos posmensajes para compartir medios, pizarrón (modelo basado en datos) Simplicidad en el agregado de nuevos recursos de contexto. Fácil configuración. Un servicio centralizado. Carece de eficiencia en la comunicación (son necesarios dos puntos de conexión por comunicación)

Llegados a este punto, nos introduciremos en una definición de la componente principal del modelo: el widget. Un widget es un componente de software que brinda una interfaz pública para algún tipo determinado de sensores (hardware) (Dey, A. K. y G. D. Abowd, 2000). Los widgets ocultan detalles de censado de bajo nivel y al mismo tiempo son componentes con alto grado de reusabilidad, facilitando el desarrollo de aplicaciones. El encapsulamiento en los widgets permite intercambiarlos entre aquellos que proveen el mismo tipo de datos (ejemplo: intercambiar un widget de radiofrecuencia por un widget de una cámara filmadora donde ambos recolectan datos de locación de individuos). Usualmente los widgets son controlados por alguna clase de administrador de widget. En el caso que las componentes sean acopladas incrementan la eficiencia, al costo de perder robustez ante fallas de componentes. Se identifican cincos categorías de componentes que implementan distintas funciones: Context Widgets: se encarga de la adquisición de información de contexto; Interpreters: cumplen la función de transformar y aumentar el grado de abstracción de la información de contexto, deben combinar varias piezas de contexto para producir información procesada de alto nivel; Aggregator: es un componente que junta información de contexto de una entidad para facilitar su acceso a las aplicaciones; Services: brindan servicios en un determinado ambiente adaptando su funcionalidad al tipo de información de contexto adquirida; Discoveres: permite que las aplicaciones (y otras entidades) optimicen su desempeño pudiendo determinar la característica del entorno (tipo de restricciones y dominio de aplicación). Entre estos componentes se pueden establecer un número limitado de relaciones. Widgets es consultada, o bien, notificada ante eventuales cambios en los clientes. Los clientes pueden ser aplicaciones, aggregators u otros widgets. A su vez, aggregators actúa como un puente entre widgets y aplicaciones. Un interpreters puede ser solicitado en un determinado estado por un widget, aggregator o aplicaciones. Los services son lanzados por las aplicaciones (también otros componentes pueden hacer uso de los servicios). Discoveres se comunica con todos los componentes, se adquiere desde los widget, interpreters y aggregators, y provee información a las aplicaciones por medio de notificaciones y consultas. La figura 4 expone una posible configuración con dos dispositivos de censado, dos widgets, un aggregator, dos interpreters, un servicio, un discoverer y dos aplicaciones. En el instante en que algún componente contexto se encuentre disponible, registra sus capacidades en un discoverer. Esto permite que los aggregators encuentren widgets e interpreters y, a las aplicaciones, encontrar aggregators, widget e interpreters. Un sensor provee datos para un context widget, el cual se

encarga de almacenar el contexto. Puede llamar a un interpreter para obtener un mayor nivel de abstracción de los datos y luego pone el contexto a disposición (para que pueda ser accedido) por otros componentes y aplicaciones. Un aggregator recolecta información de contexto de las entidades, representadas por los widgets. Finalmente, las aplicaciones pueden consultar o suscribirse a los agreegators (o directamente con los widgets, si se quiere) y llamar a interpreters (si el deseado nivel de abstracción no se encuentra disponible desde los widgets y aggregators). Estos componentes se ejecutan independientemente de las aplicaciones, asegurando una continua adquisición de información de contexto y el uso de múltiples aplicaciones. Además, todos los componentes y aplicaciones se comunican entre ellos automáticamente usando protocolos y lenguajes conocidos. Esto da lugar a que los programadores que implementan un particular componente o aplicación puedan comunicarse con otro componente sin tener conocimiento de los mecanismos usados para lograr la interacción.

5.2.2 Coordination of Contex Aware Contracts

3. LOS CONTRATOS CONTEXT-AWARE Elementos de la componente contrato La componente contrato es la información que se tiene de una componente. El contrato puede ser configurado por medio de diferentes mecanismos, desde el lenguaje cotidiano hasta un lenguaje de especificación formal y un lenguaje basado en XML2 para los casos en que sean necesarias especificaciones que puedan ser procesadas por máquinas. El tipo de tecnología y forma de implementación de los contratos es transparente para los objetos que consumen los servicios en donde se encuentran involucrados. La

configuración de un elemento contrato que forma parte de las componentes de un servicio representa la información necesaria del mismo para ser utilizado por el invocador, sin necesidad de que el objeto invocador tenga detalles de la ejecución. El contrato representa una enriquecida y efectiva interfaz de construcción que contiene toda la información sobre las componentes de los servicios y deberá tener información sobre algún tipo de información de contexto para su utilización. En este caso, el concepto de interfaz tiene mayor significación que un simple acceso a un contrato de reglas entre el objeto proveedor del servicio y el objeto consumidor.

En la figura 2 podemos observar los elementos conceptuales básicos de esta componente a través de una serie de elementos en relación con el contrato. Este metamodelo tiene las características conceptuales y operativas que se fundamentan en "Obra abierta". Para una mejor comprensión de las componentes del modelo explicitaremos a continuación su caracterización y funciones particulares. Identificador: una componente servicio es identificada para un determinado contexto por un único nombre en el espacio de identificación. Comportamiento: de acuerdo con los roles asignados en un determinado contexto, una componente servicio expone comportamientos correspondientes a provisión y pedido de operaciones, y/o publicaciones y recepción desde/hacia cada contexto. Las operaciones pueden ser definidas de dos tipos: operaciones que ejecutan cómputos o transformaciones (tipo update) y operaciones que proveen algún tipo de información sobre consultas (tipo query). Éstas se encuentran enteramente especificadas en base a un contrato, con el uso de precondición, poscondición y condicionales para lograr la coordinación entre contratos. En las condiciones de coordinación se especifican cómo requerir y proveer operaciones, así como también los eventos publicados y recibidos son coordinados en los momentos adecuados. Para lograr una comunicación precisa con una componente servicio, no sólo se tiene en cuenta qué operación fue provista o requerida y cómo el ejecutor ha lanzado el evento apropiado, sino también cómo todas esas actividades están mutuamente relacionadas para ser aprovechadas por el objeto consumidor. Un evento del contexto que lanza una operación dada puede ser parte de un conjunto de precondición, mientras que un evento emitido a través de una exitosa operación puede ser parte de una poscondición.

Las operaciones provistas y requeridas por la componente de servicio deben estar asociadas, a fin de determinar las operaciones que deben ser completadas antes de la activación de un servicio (qué es posible ejecutar en paralelo o ser sincronizado por otro camino). Por ejemplo, en el caso de una componente de servicio ManejadorOrden, la operación HacerOrden no puede ser invocada hasta que el servicio que la consume no esté correctamente autorizado por el componente de servicio AdministradorRegistros, o la operación DeleteOrder no puede ser invocada si la operación HacerOrden con el mismo parámetro OrdenID no fue previamente completada. Tipos de información: una componente de servicio debe manejar, usar, crear o tener cierta información de recursos con el propósito de proveer servicios adecuadamente. Este elemento del contrato define el tipo de información relevante para las componentes asociadas al contrato, así como también restricciones y reglas sobre instancias de esos tipos. Esto representa un modelo de información lógica de una componente de servicio. Formalmente, esta información de tipos puede ser considerada como definiciones de tipos de los parámetros de las operaciones o tipos relacionados a ellos. Configuración de parámetros context-aware: una componente servicio depende del contexto de su actual entorno. La misma, para utilizarse en diferentes contextos logrando la adaptación a eventuales cambios, debe tener definido un conjunto de parámetros de configuración. Ejemplos de estos parámetros pueden ser: Contexto del Usuario (CU), en un sentido similar a lo definido en el capítulo anterior, locación en tiempo y espacio de los servicios consumidos y suministrados. Estos parámetros pueden ser enviados dentro de las invocaciones de las operaciones de los servicios o por medio de otros caminos, mediante componentes de servicios que pueden adaptar su comportamiento ante el cambio de contexto en una determinada situación. La configuración de parámetros está directamente asociada a las relaciones de las operaciones de los servicios para lograr una mejor adaptación a la medida de las circunstancias brindada por la información relevada del contexto. El concepto de la configuración de los parámetros context-aware es un paso muy importante hacia la concepción de servicios automatizados y autoadaptables (tomando el sentido paradigmático de los teóricos de la inteligencia artificial). Parámetros no funcionales: una componente servicio puede definir un conjunto de los llamados parámetros no funcionales que caracterizan la "calidad" de sus prestaciones dentro de un determinado contexto. Estos parámetros son

elementos para los consumidores de los servicios que permiten optar por el uso de un determinado servicio, o buscar otro con el mismo o similar contrato. Como ejemplo de parámetros no funcionales podemos mencionar: performance, fiabilidad, tolerancia a fallos, costos, prioridad y seguridad.

Coordinación de contratos En términos generales, la coordinación de contratos es una conexión

establecida entre un grupo de objetos (en nuestras consideraciones los participantes serían un objeto cliente y un determinado servicio), donde reglas, normas y restricciones (RNR) son superpuestas entre los actores participantes, estableciendo con un determinado grado de control las formas de interrelación (o interacción). El tipo de interacciones establecidas entre las partes es más satisfactoria que las que se pueden lograr con UML o lenguajes similares (orientados a objetos) debido a que éstas contienen un mecanismo de superposición donde se toman como argumento los contextos. Cuando un objeto cliente efectúa una llamada a un objeto suministro, el contrato "intercepta" la llamada y establece una nueva relación teniendo en cuenta el contexto del objeto cliente, el del objeto servidor e información relevante (respecto de la relación) adquirida y representada como contexto del entorno. Como condición necesaria, la implementación de los contratos no debe alterar el diseño y funcionalidad en la implementación de los objetos.

5.3 Especial Conditionals in Contracts Rules

En las implementaciones de campo de cursos de nivel superior realizadas por el equipo de "Obra abierta", uno de los recursos primarios que los educadores utilizaban era el registro de actividades de la plataforma. Podemos ahora considerar una información de contexto, a un valor cuantitativo que resignifique una característica del comportamiento de un usuario, tal como lo expusimos en el capítulo V. Consultar los registros de actividades es una práctica muy usual para obtener información sobre el "grado" de interactividad que los usuarios tienen en el marco del dispositivo hipermedial. En los casos más básicos, existe una estructura de "planilla" (filas y columnas) en las que se detallan los registros con los siguientes campos: fecha, hora, usuario, tareas realizadas. Las tareas pueden estar significadas con valores literales como: "visitas al foro", "visualización de perfiles de usuarios", "obtención de recursos", "modificación datos de la wiki", etcétera.

El "grado" de interactividad como información de contexto fue observado específicamente durante la experiencia de taller expuesta en el capítulo IV bajo el título "Un diseño de taller físico-virtual-interactivo-comunicacional". Retomando esta temática y atendiendo a la posibilidad de usar la información de interactividad como parámetro context-aware de los contratos según lo referido en "Elementos de la componente contrato", de este mismo capítulo, podremos establecer un lazo de retroalimentación entre las prácticas efectuadas en la plataforma Moodle, establecidas en el registro de actividades y las acciones que devengan de los contratos. En esta sección utilizaremos un modelo particular de métrica y una propuesta de integración de dicho modelo al framework del dispositivo hipermedial dinámico a través de los contratos context-aware.

5.3.1 MConditionals

Para atender a los requerimientos anteriores, en primer lugar contamos con la aplicación Moodle con el agregado de la componente adicional contrato. El contrato fue incorporado a la herramienta foro de Moodle, mediante un prototipo experimental siguiendo el modelo arquitectónico desarrollado en el capítulo V. Cabe destacar que, si bien lo fundamentado sobre el agregado de la componente contrato a la arquitectura original de la plataforma está referido al proyecto Sakai, la misma es similar a la que proponemos para Moodle. En segundo lugar, diseñamos un mecanismo de métrica que fue usado como parte esencial de las reglas de los contratos, tomando como referencia un modelo de estandarización bien definido. Particularmente, la métrica formará parte del condicional de las reglas del contrato. A estos tipos de condicionales los denominaremos: "Mcondicional". Conceptualmente no existe diferencia entre un Mcondicional y una condición común, los valores de verdad de ambos pueden depender de la invocación de métodos o funciones programadas para el sistema y colaborarán en la articulación de la ejecución de acciones (servicios y procesos computacionales). Al incorporar las métricas en el modelo original de contratos contextaware, proponemos que el sistema sea más adaptable a nuevos cambios del contexto de usuario (por ejemplo, en concordancia con el tipo de interacción de la herramienta foro). Esto implica que las reglas que se pueden establecer para el contrato se formulan en el marco de la complejidad de los procesos didácticos (o investigativos y/o productivos), entendiendo que los mismos siempre exceden la posibilidad de ser absolutamente reglados. Por lo tanto, las mencionadas reglas estarán singularizadas en función de la dinámica de dichos procesos, debiendo ser explícitas y de clara implementación y por sobre todo, adaptables para los contextos específicos de los usuarios (en este caso, estudiantes a nivel de grado). El primer paso es lograr la explicitación de

las reglas del contrato y que los condicionales representen criterios de decisiones sobre aspectos relevantes del proceso didáctico (investigativo o productivo). Por ejemplo, un estudiante puede adquirir un servicio determinado de una herramienta, previo a la evaluación de una condición representada como condicional de una regla. En general, a estas reglas debemos diseñarlas con el cuidado de no incorporar redundancias, ambigüedades o incoherencias, tanto entre las propias reglas de un contrato como con otras reglas implícitas que se desprenden de los servicios. De esta manera, definimos las reglas del contrato como un conjunto de condiciones, acciones y prioridades. La condición es un expresión Boolean sobre relaciones (mayor, menor, igual, distinto, etc.) entre parámetros y valores concretos. Las acciones conforman un conjunto de asignaciones de valor a otros parámetros también definidos por el tipo de regla. Algunos de los parámetros de las acciones deben ser "métodos de cálculo" que permitan cambios en el comportamiento de los servicios en los cuales estas reglas son aplicadas. La prioridad permite simplificar la cantidad de reglas que se deben escribir: en lugar de la escritura de una regla para cada combinación de posibilidades de los valores de los parámetros, se asegura que dos reglas no puedan ser ejecutadas simultáneamente. El usuario podría escribir una prioridad baja para todas las reglas y luego, con prioridades altas, ir identificando las excepciones para el caso configurado inicialmente. En síntesis, las reglas son ejecutadas mediante un orden de prioridades. En consecuencia, en cada tipo de regla, cada una tiene sólo una prioridad. Para ejemplificar este concepto, consideremos una regla donde, dependiendo del tipo y cantidad de interacciones de un sujeto en formación (por ejemplo, Juan Pérez), el servicio de la herramienta foro, es readaptado. Cabe aclarar que en este caso, la acción del contrato establece el valor de calificación de la intervención del usuario Juan Pérez en los temas tratados en los foros. Entonces, el valor de verdad (Boolean) del condicional (donde parte de la expresión está representada por el predicado: getForum$_T heme >$ $Minimus_F orum_T heme) dependedelaejecuciónde unamétrica, debidoaqueelmétodogetForum_t hemelanzaelproceso$

la integración del subsistema de métrica como si fuera un sistema externo conectado por la componente contrato. If (getStudent = "Juan") and (getForum$_t heme > Minimus_F orum_T heme) ThensetPermissionQual$
$Reglasdecontratosconcondicionales$

Modelo de integración

Para implementar la invocación de métricas mediante métodos correctos, propusimos desde la perspectiva del rediseño e implementación computacional, un modelo de integración de muy bajo costo, sin cambios sustanciales ni en la arquitectura original ni en el código de la implementación. El modelo conceptual de métrica pertenece al Modelo INCAMI (Information Need, Concept model, Attribute, Metric and Indicador: Información relevante, Modelo Conceptual, Atributos, Métricas e Indicadores) (Olsina, L., G. Lafuente y G. Rossi, 2000). INCAMI es un framework organizacional, orientado a la medición y evaluación que permite economizar consistentemente, no sólo metadata de métricas e indicadores, sino también valores mensurables en contextos físicos. Por medio de un diagrama UML, se representa un modelo global de integración, teniendo en cuenta experiencias vinculadas al agregado de nuevas componentes en determinadas implementaciones resueltas para sistemas elearning similares Sakai. La integración contempla, por un lado, el modelo de coordinación de contrato (enmarcado en el área de contrato, y por el otro, el modelo de métrica propuesto por Olsina (Olsina, L. y G. Rossi, 2002). En la figura 3 podemos observar lo correspondiente a cada una de las áreas mencionadas. La figura 3 también nos muestra que la principal componente para lograr la integración está representada por la incorporación de una relación de agregación entre la componente contrato, representada por una clase en el modelo UML y la entidad método. Los condicionales de las reglas de los contratos son invocados (mediante un método explícito relacionado con la noción de los Mcondicionales, por ejemplo, getForum$_t heme) pormediodeunmecanismodecallbackquepermitelacorrectainvocacióndelamétrica. Elmodelodemé$

suplir los requerimientos de interacción referidos al registro de actividades. Si tenemos en cuenta el proceso de definición de métricas, debemos crear métricas por cada atributo. Definido en una estructura denominada árbol de requerimiento (tema que desarrollaremos más adelante), cada atributo puede ser cuantificado por más de una métrica. Cabe aclarar que este proceso de confección de métricas, podemos considerarlo como una metodología propia con fines específicos de esta área disciplinar de la ciencia. Las métricas contienen la definición de métodos y escalas para un determinado tipo de medición y/o cálculo. Dada una métrica m representa una relación (mapeo) m: A X, donde A es un atributo empírico de una entidad (el mundo empírico), X es la variable en la cual se asignarán (o pueden ser asignados) valores categóricos y numéricos (el mundo formal), y la flecha denota la relación de mapeo (mapping). Para lograr el mapping, es necesaria una correcta y precisa definición de actividades de medición por

medio de una especificación explícita de los métodos de las métricas y las escalas tal como podemos ver en la figura 3. Para las métricas directas, podemos aplicar métodos de mediciones objetivos y subjetivos, en cambio las métricas indirectas sólo pueden ser calculadas mediante funciones, con la intervención de fórmulas lógicas/matemáticas.

5.3.2 DMConditionals

Como ya observamos, existen diferentes alternativas para lograr la integración de modelos por medio de un contrato. En este sentido, planteamos en la sección anterior un modelo de integración donde, por medio de los condicionales de las reglas, se invocaban métodos de un modelo externo. Ahora proponemos una nueva integración de un modelo externo que permitirá, aplicando la minería de datos, enriquecer aún más la semántica de los contratos. Estado actual de la aplicación de la minería de datos a los sistemas de enseñanza mediados por la web En este apartado nos introduciremos en aspectos relevantes del estado del arte en la investigación sobre la aplicación específica de técnicas de minería de datos a los sistemas de enseñanza mediados por la web o sistemas e-learning. Tanto la minería de datos como dichos sistemas son áreas que muestran importantes desarrollos, y la vinculación de las mismas está suscitando un creciente interés por parte de investigadores y empresas

Entre los diferentes métodos y técnicas existentes de minería de datos, nos hemos centrado principalmente en la minería de utilización web, concretamente en la clasificación y agrupamiento, descubrimiento de reglas de asociación y secuencias de patrones, ya que son técnicas motivadas por los mismos requerimientos planteados en este capítulo y fundamentados en la perspectiva de "Obra abierta". Asumimos que en los sistemas e-learning el principal objetivo de la utilización de técnicas de data mining (minería de datos) es proveer mecanismos que permitan guiar a los estudiantes durante su aprendizaje, con el propósito de incrementar las facilidades y servicios para la obtención de información y conocimiento calificado. Las técnicas más utilizadas en la minería de datos aplicada a los sistemas de e-learning son: clasificación y agrupamiento, descubrimiento de reglas de asociación y análisis de secuencias. A continuación, vamos a exponer brevemente los principales trabajos de investigación agrupados dentro de estos tres tipos de técnicas, aunque algunos investigadores utilicen no una, sino varias. Clasificación y agrupamiento Las técnicas de clasificación y agrupamiento o clustering (Arabie, P., J. Hubert y G. de Soete, 1996) consisten en la habilidad intelectual para ordenar o dividir fenómenos complejos (descriptos por conjuntos de objetos con datos altamente dimensionales) en pequeños y comprensibles unidades o clases que permiten un mejor control o comprensión de la información. Su aplicación a sistemas e-learning permite agrupar a los usuarios por su comportamiento de navegación, agrupar las páginas por su contenido, tipo o acceso y agrupar similares comportamientos de navegación. A continuación describiremos algunos trabajos de aplicación de minería de datos en educación. La técnicas de agrupamiento son utilizadas por Gord McCalla y Tiffany Tang (Tang, T. y G. McCalla, 2004) para formar clusters o grupos de usuarios basándose en su comportamiento de navegación, aplicando un algoritmo de clustering basado en largas secuencias generalizadas. También proponen incluir un sistema recomendador inteligente dentro de un sistema de aprendizaje basado en web evolutivo capaz de adaptarse no sólo a sus usuarios, sino también a la web abierta. El sistema puede encontrar contenidos relevantes en la web pudiendo personalizar y adaptar sus contenidos, basándose en observaciones del sistema y por las propias valoraciones acumuladas dadas por los miembros en formación. Otro trabajo que también emplea agrupamiento es el realizado por Elena Gaudioso y Luis Talavera (Romero, C., 2003) en el que analizan los datos obte-

nidos de cursos basados en sistemas e-learning y utilizan técnicas de clustering similares al modelo probabilístico de Naive Bayes para descubrir patrones que reflejan comportamientos de los usuarios. El objetivo es utilizar la minería de datos para dar soporte a la tutoría en comunidades de aprendizaje virtual. La utilización conjunta de clustering con otras técnicas como secuenciación es realizada por Julia Miguillón y Enric Mor (Mor, E. y J. Minguillón, 2004) para analizar el comportamiento de navegación de los usuarios para la personalización del e-learning. Utilizan clustering de estudiantes para intentar extender las capacidades de secuenciación de algunos sistemas estándares de manejo de aprendizaje como SCORM para incluir el concepto de itinerario recomendado. Los autores Erkki Sutinen y otros (Sutinen, E., W. Hämäläinen, J. Suhonen y H. Toivonen, 2004) proponen un modelo híbrido que combina técnicas de minería de datos y de aprendizaje de máquinas para la construcción de una red bayesiana (http://es.wikipedia.org/wiki/Red$_bayesiana$)$quedescribeelprocesodeaprendizajedelosestudiantes.Suobjetivoesclasificara$

nan, se ha establecido una tipología informal para este tipo de estructuras. Así, se habla de reglas de decisión, asociación, clasificación, predicción, causalidad, optimización, etc. En el ámbito

de la extracción de conocimiento en bases de datos, las más estudiadas han sido las reglas de asociación, de clasificación y de predicción. Las reglas de clasificación tienen como objetivo almacenar conocimiento encaminado a la construcción de una clasificador preciso. En su antecedente contienen una serie de requisitos (en forma de condiciones) que debe cumplir un objeto determinado para que pueda considerarse perteneciente a la clase identificada con el consecuente de la regla. Desde el punto de vista sintáctico, la principal diferencia con las reglas de asociación es que presentan una sola condición en el consecuente, que además pertenece a un identificador de clase. La estructura de una regla puede ser caracterizada (en términos generales) mediante un conjunto de condicionales lógicos en los que se fija un criterio de ejecución de la acción. En este ejemplo, se verifica en el usuario "alumno", si su nivel de interacción en el foro pertenece a la clase de alto, y si la calidad de sus intervenciones se encuentran en un rango aceptable. Este tipo de información es obtenida del registro de actividades de la plataforma.4 Supongamos, ahora, que los valores de estos dos últimos condicionales se obtienen a partir de la ejecución de un algoritmo tipo TDIDT (Top-Down Induction of Decision Trees), entonces, debe ser invocado el algoritmo por medio de un método (en este caso, getCalidad$_intervencion)atravésdeunareglaexplícitaenelcontrato, delasiguientemanera$:

if (usuario = 'alumno') and getCalidad$_Intervencion = `aceptable')andthis.getCantidad_Interaccion =$ `alto')$thenherramienta.accion()$

Interesa destacar la estructura de la regla Si-entonces (If-Then) y los condicionales de los cuales se desprende la ejecución de un algoritmo de data mining. En este ejemplo, este tipo de condicional se muestra subrayado; y lo denominaremos MDcondicional (en lo conceptual, tenemos gran similitud con los Mcondicionales de la sección "Métricas de interacción para 'Obra abierta'"). De esta manera, por medio del MDcondicional, se capturan los resultados de la inferencia del árbol de decisión que se ilustra con la figura 4.

En las secciones posteriores nos introduciremos en las propuestas y justificaciones para la incorporación de las técnicas de minería de datos a través de los MDcondicional. Algoritmos de construcción de los MDcondicionales Una de las formas más simples de representar conocimiento es mediante árboles de decisión. Dicha estructura de representación del conocimiento está formada por una serie de nodos, donde: cada nodo interno es etiquetado con el nombre de uno de los atributos predictores; las ramas que salen de un nodo interno son etiquetadas con los valores del atributo de ese nodo; cada nodo terminal, o nodo hoja, es etiquetado con el valor del atributo objetivo. Un ejemplo de árbol de decisión se muestra en la figura 4, donde el atributo objetivo es el atributo nivel de aceptación y los atributos predictores son lo atributos: interacción, respuestas, evaluación, consultas. Una característica muy interesante de los árboles de decisión es que, a partir de ellos, es muy fácil derivar un conjunto de reglas de producción del tipo Si-entonces (If-Then) completamente equivalente al árbol original. El algoritmo que nos permite realizar este cambio de modelo de representación es casi trivial y convierte cada camino que va desde el nodo raíz hasta una hoja, en una regla de la siguiente forma: los nodos internos y sus correspondientes ramas de salida son convertidos en las condiciones del antecedente de

la regla; los nodos hojas se convierten en el consecuente de la regla. Al convertir el árbol de decisión en una colección de reglas se obtiene una regla por hoja del árbol. Dichas reglas serán, por tanto, mutuamente excluyentes. El algoritmo de construcción de árboles de decisión más popular es el algoritmo ID3 (Quinlan, J. R., 1987). Este algoritmo está basado en la división sucesiva y construye el árbol de decisión desde la raíz hacia las hojas, incrementando en cada paso la complejidad del árbol. Inicializar T con el conjunto de todas las instancias Si (todas las instancias en el conjunto T satisfacen el criterio de parada) Entonces Crear un nodo hoja etiquetado con el nombre de la clase y parar. Sino Seleccionar un atributo para utilizar como atributo de particionado. Crear un nodo etiquetado con el nombre del atributo y crear una rama por cada valor del atributo. Particionar T en subconjuntos tales que contengan las instancias que cumplan los valores del atributo. Aplicar este algoritmo recursivamente para cada subconjunto. Fin Si Recorrer el árbol y formar las reglas.

Detalles del modelo de integración

Para lograr la incorporación de la componente data mining, volvemos a proponer una nueva integración a muy bajo costo, desde el punto de vista del rediseño e implementación con el propósito de facilitar la misma y el impacto en la modificación de los módulos estándares de la aplicación e-learning. En particular, Sakai, a diferencia de Moodle, provee una arquitectura en la que es posible trabajar bajo

el paradigma orientado a objetos (POO), con toda la potencia y versatilidad de la tecnología Java, motivo por el cual se logra una mejor adaptación del modelo de la componente conceptual data mining mediante un correcto diseño orientado a objeto (DOO) (Grady, B., 1982). En la figura 5, representamos el contrato con un diagrama de clase. Las reglas, normas y restricciones (RNR) están representadas con tres métodos independientes en el diagrama de la clase contrato. Teniendo en cuenta la analogía planteada entre las reglas del contrato y la estructura de los árboles que determina el algoritmo TDIDT, es posible plantear un nuevo mecanismo total o parcial de construcción de reglas dentro de los contratos. Esta nueva perspectiva puede ser conceptualizada de las siguientes

formas: 1) el uso de técnicas de data mining permite una construcción automática de reglas. Bajo los contratos (representado en la figura 5 como una relación de agregación en UML) se realiza dicha construcción en tiempo de ejecución. En este caso se aporta automatización, adaptabilidad y dinamismo; 2) proporciona una manera indirecta de recolección de información de contexto, debido a la naturaleza de este tipo de técnicas (minería de datos) donde la fuente de información procesada reviste a datos empíricos y valores que denotan tipos de relaciones. Con esta información de contexto, los contratos adquieren mayores niveles expresivos en cuanto a su sensibilidad al contexto, similar a los sistemas context-aware. Cuando, dentro en una NRR se utiliza un método para implementar algunas de las técnicas de data mining (en este caso, la construcción de un árbol de decisión por medio de un algoritmo TDIDT), queda conformada una regla donde el valor de su condicional (el cual fue denominado MDcondicional) responde –conceptualmente– a una estructura, en este caso: un árbol. El vínculo entre las reglas del contrato y la efectiva representación del TDIDT se concreta a través de una relación de agregación entre los parámetros del contrato (representado por la clase "parámetros context-aware", que se encuentra dentro del subsistema que engloba el contexto, similar a la propuesta de Schmidt, A., 2005) y la clase DMcondicionales (representada como una generalización de subclases donde se distinguen las posibles técnicas de data mining). En la figura 5, las componentes significativas para la integración de los modelos se encuentran enmarcadas en color gris. La primera pertenece al área de coordinación de contrato y mantiene la estructura original del modelo pro-

puesto en el proyecto "Obra abierta". La segunda componente se encuentra representada por una clase conceptual que simboliza las características y funcionalidades de minería de datos, articuladas para cumplir los objetivos y requerimientos propuestos.

Mecanismo de comunicación La figura 5 es una reelaboración significativa de integración arquitectónica entre dos modelos (el framework e-learning y el modelo de minería de datos) donde se muestran, además, relaciones salientes de dependencias entre objetos de las distintas áreas. Por ejemplo, cuando un objeto perteneciente a una herramienta5 de la aplicación atiende un pedido de un usuario, se comunica con otro objeto (en este caso un objeto contrato, perteneciente al área de coordinación de contrato) con el propósito de verificar la existencia de una regla que determine cuáles son las acciones que se deben tomar, teniendo en cuenta determinadas condiciones. Si alguna de las condiciones está representada por un MDcondicional, entonces existirá una comunicación entre el objeto contrato (que contiene la regla) y un objeto del área de minería de datos. Esta cadena de eventos y relaciones entre áreas permite que un servicio ordinario del framework Sakai se pueda enriquecer con información de contexto recopilada con técnicas de minería de datos.

Caso de uso Retomamos un requerimiento similar al propuesto en la sección "Métricas de interacción para 'Obra abierta' ", y agregamos que el análisis de información almacenada es utilizada como recompilación de información (datos) de contexto. En este ejemplo, cambiaremos pequeños aspectos de los requerimientos originales para poder involucrar las técnicas de minería de datos como parte de la implementación, refiriendo el caso sobre el entorno Sakai. Supongamos ahora que volvemos a la hipótesis de contar con un dispositivo hipermedial conformado con la arquitectura del modelo de integración y además, la implementación del modelo de la figura 6. Sabemos que la herramienta se encuentra configurada con un contrato (del tipo de la sección "Los contratos context-aware") conformado, a su vez, con una regla similar a la presentada en la sección "Modelo de integración".

Entonces, cuando un usuario alumno ingrese al foro y ejecute cualquier tipo de servicios donde intervenga el contrato anterior, las prestaciones que recibirá estarán supeditadas a los valores que se obtengan de la ejecución del clasificador en base a la información obtenida a partir del registro de actividades de la plataforma Sakai. Cabe aclarar que Sakai contiene un registro de actividades al cual podemos acceder desde las tablas de las bases de datos. La implementación y operatoria que

implique la clasificación de los datos desde los MDcondicionales es responsabilidad del subsistema que lo implementa, respetando de esta manera la propiedad de independencia entre los modelos de la integración señalada en la sección "Detalles del modelo de integración". Si profundizamos en el tipo de relación que se establece a través de los MDcondicionales y tomamos como referencia el punto de vista del usuario, podemos caracterizar el flujo de ejecución de órdenes de la siguiente manera: en la figura 6, podríamos considerar un estado inicial, en el que el sistema ejecuta las reglas de un contrato que contiene una de ellas con un MDcondicional. Por medio de un método, se produce la invocación del método de clasificación que se encuentra implementado en el subsistema de minería de datos (referenciado con la entidad "modelo externo" en la sección "El contrato como conector"). Las transacciones y estados que se suscitan son luego transparentes para el modelo de integración y pueden resumirse en la ejecución de un algoritmo de data mining donde se provoca un consulta a la base de datos Sakai y la ejecución de métodos de clasificación (similares a los de la sección "Antecedentes de técnicas de minería de datos") para obtener el valor de verdad del MDcondicional, llegando, así, al estado final del ciclo de ejecución. Con este tipo de configuración, la herramienta foro de Sakai adquiere características context-aware y el agregado de la siguiente propiedad: los contratos context-aware pueden ser modificados en tiempo de ejecución. Esto quiere decir, que el mismo alumno, ante una nueva interacción con el foro y bajo las mismas condiciones, puede obtener resultados diferentes en el caso de que se hayan modificado las reglas del contrato. Además, la incorporación de un subsistema para la aplicación de técnicas de minería de datos ofrece mayor potencial expresivo para la construcción de reglas en los contratos. Finalmente, sobre esta temática consideramos que el uso de un correcto modelo para la implementación de técnicas de minería en la clasificación de valores de contextos permite potenciar las soluciones creadas para los dispositivos hipermediales dinámicos para educación e investigación. Es importante entonces, que el grupo de diseñadores, expertos y desarrolladores sean competentes en la utilización comprensiva de este tipo de técnicas.

Hacia la implementación

Con el objetivo de profundizar la visión sobre el diseño de dispositivos hipermediales dinámicos, abordamos en este capítulo especificaciones técnicas que justificaron la implementación y diseño de modelos de integración donde la componente contrato adoptaba el rol de conector. Observamos, entonces, que la base conceptual para que el contrato pueda ser visto y manipulado en su diseño e implementación como una pieza de software, se centra en el rol protagónico y responsable que debemos adoptar como expertos diseñadores de dispositivos hipermediales dinámicos en el ciclo de vida del mismo (diseño, implementación, uso). Es a partir de esta nueva toma de posición, que la etapa de diseño adquiere una dimensión activa centrada en la participación con la puesta en obra de la teoría de coordinación de contratos context-aware. Tomamos conciencia de que el camino hacia la implementación del marco teórico propuesto demanda detenimiento para su comprensión, integración de conocimientos de distintas disciplinas, diálogo interdisciplinar para poder alcanzar claros criterios en la utilización del contrato y en la explicitación de las reglas singulares que lo componen, configuradas en función de los procesos educativos, investigativos o de producción que las susciten. Teniendo en cuenta su poder expresivo, podremos desarrollar adecuadas aplicaciones contextualizadas que nos abran la posibilidad de desarrollar nuestra formación continua y creatividad, construyendo polifónicamente una singular mesa de arena.

....

CHAPTER 6

Framewrok to Implemet Context Aware Contract in E-learning Environment

6.1 Introducción

A medida que el avance en la investigación y desarrollo de plataformas e-learning brindan mejoras e innovaciones en herramientas (videoconferencias, porfolios, wikis, workshops, etc.) y sus respectivos servicios, crece la cantidad de posibles configuraciones de los espacios e-learning. Estas configuraciones abarcan diferentes tipos de requerimientos pertenecientes a las etapas de diseño, desarrollo e incluso exigen que el espacio e-learning se adapte en tiempo de ejecución. A partir de estos requerimientos se definen los procesos e-learning (que nosotros denominamos Pe-lrn) (7) de manera semejante a procesos de negocio en otro dominio de aplicación. Al igual que los procesos de negocios en una Aplicación Web convencional, los Pe-lrn están compuestos por transacciones Web (). En este contexto, una transacción (o transacción e-learning) es definida como una secuencia de actividades que un usuario ejecuta a través de una Aplicación e-learning con el propósito de efectuar una tarea o concretar un objetivo, donde el conjunto de actividades, sus propiedades y las reglas que controlan sus ejecuciones dependen del Pe-lrn que la Aplicación debe brindar. Un ejemplo de estrategia didáctica es la posibilidad de que un alumno acceda a determinado tipo de material (vídeos, archivos, etc.) dependiendo de sus intervenciones en los Foros. Estos tipos de requerimientos resultan difíciles de implementar con las actuales aplicaciones e-learning de extendido uso a nivel global.

Las características funcionales de las actuales aplicaciones Web e-learning (AWe-lrn), como Sakai[a], se basan en brindar navegación entre páginas a través de links y ejecución de transacciones e-learning por desde las herramientas (ej., Foros, Anuncios, Exámenes, Blogs, etc.) que utilizan los servicios de la plataforma (ej., edición, manejo de audio y vídeo, consultas, navegación, etc.).

En el marco de los análisis efectuados podemos sostener que el proyecto Sakai brinda una de las propuestas más consolidadas de diseño y desarrollo de entornos colaborativos e-learning para educación, orientado a herramientas que se implementan a través de servicios comunes (servicios bases). Por ejemplo, el servicio de edición de mensajes es utilizado en las herramientas Foro, Anuncio, Blog, PorFolio, etc. Más aun, otras de las características salientes de Sakai es la versatilidad para su extensión y/o configuración. En efecto, Sakai permite alterar ciertas configuraciones en tiempo de ejecución, por ejemplo, instrumentar una nueva funcionalidad en un servicio base de Sakai.

Sin embargo, estas soluciones no pueden resolver aquellos Pe-lrn que involucren cambios en el comportamiento de la relaciones entre un componente (cliente) que ocasiona, a través de un pedido, la ejecución de un componente servidor (proveedor). Estos tipos de cambios refieren a la capacidad de adaptación dinámica del sistema (14) extendiendo, personalizando y mejorando los servicios sin la

[a]SaKai: Entorno colaborativo y de aprendizaje para enseñar. Es de código abierto y está resuelto con tecnología Java. Véase detalles en wwww.sakaiprojet.org)

necesidad de recompilar y/o reiniciar el sistema. En este trabajo se presenta un propuesta para la incorporación (agregado) de propiedades de adaptación dinámicas a los servicios bases del framework Sakai, especialmente diseñadas para implementar Pe-lrn definidos en el marco de (9) donde se requieren nuevos aspectos de adaptación (1) en la perspectiva de los Dispositivos Hipermediales Dinámicos en el campo del e-learning (12) con característica *context-aware* (2; 5). Se entiende por DHD *"a una red social mediada por las TIC en un contexto físico-virtual-presencial, donde los sujetos investigan, enseñan, aprenden, dialogan, confrontan, evalúan, producen y realizan responsablemente procesos de transformación sobre objetos, regulados según el caso, por una coordinación de contratos integrados a la modalidad participativa del Taller"*. (San Martín et al, 2008)

El resto de este trabajo se encuentra organizado de la siguiente manera: Tras esta introducción se presenta una referencia conceptual del framework contratos con características "context-aware", utilizados en el proyecto Obra Abierta: Dispositivos Hipermediales Dinámicos para educar e investigar (Dir. Dra. Patricia S. San Martín (Sección 6.2). Luego se describe la macro-arquitectura de integración entre los frameworks de Sakai, la teoría de coordinación de contratos sensible al contexto (TCCs-c), un modelo de contratos context-aware y aplicaciones externas (sección 6.3). Siguiendo, se presenta un modelo de implementación de la TCCs-c a través del de patrones de diseño (sección 6.4). En la sección 6.5 se propone un caso de estudio concreto. Finalizando con una breve conclusión.

6.2 Contractos con características sensibles al contexto

Nuestra propuesta de solución a los requerimientos mencionados sobre adaptación dinámica comienza con la construcción de un modelo de contrato orientado a la implementación de servicios sensibles al contexto. El uso de contratos parte de la noción de Programación por Contrato ("Programing by Contract") de Meyer (11) basada en la metáfora de que un elemento de un sistema de software colabora con otro, manteniendo obligaciones y beneficios mutuos. En nuestro dominio de aplicación consideraremos que un objeto cliente y un objeto servidor "acuerdan" a través de un contrato (representado con un nuevo objeto) que el objeto servidor satisfaga el pedido del cliente, y al mismo tiempo el cliente cumpla con las condiciones impuestas por el proveedor. Como ejemplo de la aplicación de la idea de Meyer en nuestro dominio de sistemas e-learning planteamos la situación en que un usuario (cliente) utiliza un servicio de edición de mensajes (servidor) a través de un contrato que garantizará las siguiente condiciones: el usuario debe poder editar aquellos mensajes que tiene autorización según su perfil (obligación del proveedor y beneficio del cliente); el provedor debe tener acceso a la información del perfil del usuario (obligación del cliente y beneficio del proveedor).

A partir de la conceptualización de contratos según Meyer se propone una extensión por medio del agregado de nuevas componentes para instrumentar mecanismos que permitan ejecutar acciones dependiendo del contexto.

En aplicaciones sensibles al contexto (6), el contexto (o información de contexto) es definido como la información que puede ser usada para caracterizar la situación de una entidad más allá de los atributos que la definen. En nuestro caso, una entidad es un usuario (alumno, docente, etc.), lugar (aula, biblioteca, sala de consulta, etc.), recurso (impresora, fax, etc), u objeto (examen, trabajo práctico, etc.) que se comunica con otra entidad a través del contrato. En (2) se propone una especificación del concepto de contexto partiendo de las consideraciones de Dourish (20) y adaptadas al dominio e-learning, que será la que consideraremos en este trabajo. Contexto es todo tipo de información que pueda ser censada y procesada, a través de la aplicación e-learning, que caracterizan a un usuario o entorno, por ejemplo: intervenciones en los foros, promedios de notas, habilidades, niveles de conocimientos, máquinas (direcciones ip) conectas, nivel de intervención en los foros, cantidad de usuarios conectados, fechas y horarios, estadísticas sobre cursos, etc.

En términos generales, la coordinación de contratos es una conexión establecida entre un grupo de objetos aunque en este trabajo se consideran sólo dos objetos: un cliente y un servidor. Cuando un objeto, cliente efectúa una llamada a un objeto servidor (ej., el servicio de edición de la herramienta Foro), el contrato "intercepta" la llamada y establece una nueva relación teniendo en cuenta el contexto del objeto cliente, el del objeto servidor, e información relevante adquirida y representada como contexto del entorno (20). Como condición necesaria, el uso de contratos no debe alterar la funcionalidad de la implementación de los objetos participantes, aunque sí se espera que altere la funcionalidad del sistema.

A continuación se presenta un modelo conceptual resumido de contratos sensibles al contexto. Se brindarán detalles sobre algunas de los componentes y relaciones esenciales para la integración de este

modelo con el framework Sakai y con los módulos que instrumentan la coordinación de contratos 6.3.

6.2.1 Elementos de los contratos sensibles al contexto

Un contrato que siga las ideas de Meyer contiene contiene toda la información sobre los servicios que utilizarán los clientes. Para incorporar sensibilidad al contexto nuestros contratos deberán tener referencias sobre algún tipo de información de contexto para su utilización.

En el diagrama de relaciones entre entidades mostrado en la Figura 6.1 se describen los elementos que componen el concepto de contrato sensible al contexto, especialmente para Pe-lrn.

Figure 6.1: *Modelo de un contrato context-aware*

El propósito de la presentación de este modelo de contrato sensible al contexto es identificar cuales son los componentes que lo componen y que serán referenciados en la siguiente sección para la implementación de un modelo de integración con el framework Sakai.

La figura comienza con la representación de un contrato según Meyer donde se caracterizan los principales elementos que lo componen (pre-condiciones, acciones, pos-condiciones). La flechas salientes de la zona gris indican los dos tipos de relaciones que se debe instrumentar para incorporar un mecanismo que provea a los contratos la característica de sensibilidad al contexto. La primera de estas relaciones indica que desde las acciones del contrato se invocan a los servicios que provee un objeto servidor. La segunda relación se da debido los invariantes pueden depender de información de contexto. En la porción derecha de la Figura 6.1 aparecen las entidades necesarias para obtener contratos sensibles al contexto. Las explicamos a continuación.

Servicios: En esta componente se representan los elementos necesarios para la identificación de un servicios y clasificación de los servicios que pueden formar parte de las acciones de los contratos. Por ejemplo, nombre del servicio, identificadores, alcance, propósito, etc. Para más detalles consultar (5). El comportamiento funcional de cada servicio se expone a través de la componente **Comportamiento**.

Comportamiento: El comportamiento de un servicio se logra a partir de combinar operaciones y eventos que son representadas con el las componente **Operaciones** y **Eventos**. De la misma manera el servicio puede ser implementado a través del uso de eventos, representados con el componente **Eventos**, que puede lanzado operaciones del componente **Operaciones**. Por ejemplo, de acuerdo con los roles (ej., alumno, instructor, docente, etc.) asignados a un usuario de una herramienta involucrado en un Pe-lrn, en un determinado contexto del entorno (ej., si está en un espacio Foro) y del usuario (ej., si tiene permiso de moderador), la componente **servicio** brindan distintas funcionalidades (ej., editar un mensaje), que son instrumentadas por medio de operaciones concretas (ej., guardar un mensaje en una tabla) y/o a través de la publicación o subscripción de eventos. Las operaciones pueden ser de dos tipos: operaciones que cambian el estado del sistema (tipo "update") y/o operaciones que proveen algún tipo de información sobre consultas (tipo "query"). Estas operaciones pueden ser creadas creadas por el programador o utilizadas a partir de las operaciones provista por el núcleo del framework sakai (ej., adquirir el id de un canal de mensajes) usadas en los servicios bases (ej., el servicio de edición de mensajes).

Parámetros Context-Aware: Se denomina **parámetros context-aware** a la representación de la información de contexto que forma parte de los parámetros de entrada de las funciones y métodos exportados por los servicios, estableciendo de esta manera una relación entre el componente **Servicios** y el componente **Parámetros contex-aware**. La influencia de estos parámetros en el comportamiento funcional de los servicios es representada a través de la relación entre los componentes **Parámetros context-aware** y **Comportamiento**.

Contexto: Este componente representa el contexto o información de contexto definida anteriormente en esta sección. Para nuestro modelo este tipo de información es utilizada de dos maneras diferentes: en primer lugar para la asignación de los valores que toman los **Parámetros context-aware**; en segundo lugar esta información puede ser utilizada para definir los invariantes que se representan en los contratos.

6.3 Modelo de integración de Sakai con contratos

En esta sección se muestra un esbozo de la propuesta para incorporar a Sakai un mecanismo de coordinación de contratos sensibles al contexto, a través de la presentación de una diseño que describe la comunicación entre módulos (15) y sus dependencias. En la figura 6.2 los módulos son representados con paquetes UML y las relaciones entre ellos por medio de flechas que indican comunicación y dependencias. A su vez, en un segundo plano se muestra cuáles son las clases específicas de cada módulo y cómo se implementa la integración a través de estas.

El framework Sakai, representado en la figura 6.2 por el módulo Sakai, está diseñado según una arquitectura de cuatro capas (10): La capa de **agregación** se encarga completamente de la implementación de la interfaz con el usuario al estilo de las implementaciones de portales Web. La segunda capa denominada **presentación** tiene la responsabilidad de permitir la reutilización de los componentes Web (ej., "widget" que proveen calendarios, editores "WSYWIG", etc.). La tercera corresponde a la capa de **herramientas** donde reside la lógica de negocio de las herramientas Sakai que interactúan con el usuario (ej., Foro, Wiki, etc.). Por último la capa de **servicio** implementa los servicios Sakai bajo una Arquitectura Orientada a Servicios que serán utilizados por varias herramientas a través de API.

Para nuestra propuesta de integración tendremos en cuenta únicamente servicios que componen laa capa de **servicios** pertenecientes a los servicios bases Sakai. Los servicios del núcleo Sakai serán envueltos mediante la coordinación de contratos, lo que se representa en la Figura 6.2 con las referencias del **Módulo Sakai** hacia el **Módulo de Coordinación de Contractos**.

De esta forma, se logra controlar las invocaciones a los servicios del núcleo Sakai a través del **módulo Coordinación de Contratos**, quedando establecida una primer relación de integración. Esta misma relación es implementada a nivel de clases como una relación de agregación entre la clase *Servicios Sakai* (correspondiente al módulo Sakai) y la clase *Conector* correspondiente al framework de coordinación que será tratado en la sección 6.4 a través del uso de patrones de diseño.

Figure 6.2: *Diseño de la integración de Sakai con Contratos*

La última relación de integración se produce entre el módulo de **Coordination de Contratos** y el módulo de **Contratos Sensible al contexto** el cual fue desarrollado en la sección 6.2. En este caso, la figura 6.2 muestra cómo desde el framework de coordinación se estable una relación de composición entre la clase *Conector* y la clase *Contrato C-A* que implementa el elemento Contrato representado en la Figura 6.1 coloreada en gris.

De esta forma, instanciando dinámicamente diferentes contratos, Sakai presentará una flexibilidad en tiempo de ejecución que actualmente no posee y que es la que requieren los DHD.

6.4 Implementación de contratos en Sakai

En las secciones anteriores se mencionó al contrato como un agente activo que se encarga de la coordinación de las componentes que lo integran. En esta sección se describe cómo este mecanismo fue implementado a través de una herramienta que permite modificar implementaciones de clases Java de los servicios Sakai para incorporarle mecanismos de coordinación de contratos sensibles al contexto, según el diseño mostrado en la Figura 6.2. Sin embargo creemos que antes es conveniente resumir brevemente otros intentos de proveer la flexibilidad requerida por los DHD.

Lograr este grado de flexibilidad es absolutamente necesario teniendo en cuenta que la implementación de estas clases, para los frameworks colaborativos e-learning como Sakai, no pueden ser modificadas (rediseñadas, reemplazadas). Para estos casos, la implementación de la TCCs-c proporciona un mecanismo que se distingue de propuestas existentes para el modelado de interacción entre objetos.

6.4.1 Niveles de flexibilidad de Sakai

Diferentes estándares y frameworks de desarrollo de software basados en componentes e inyección de servicios fueron propuestos para mejorar la flexibilidad y adaptabilidad[b] de los sistemas e-learning Web, los más importantes son: CORBA, JavaBeans, Hibernate, Spring, JSF, RSF, etc. Sin embargo, ninguno de estos estándares proveen un mecanismo convenientemente implementativo y abstracto en donde se permita representar a las relaciones entre usuarios y servicios como un objeto de primera clase (11). En este sentido, se implementa una solución utilizandos patrones de diseño que implemente características deseadas de la TCCs-c sobre la capa de servicios base de framework Sakai.

Desde un punto de vista implementativo, esta propuesta cambia la filosofía Java2EE-Sakai[c] donde la incorporación de nuevas prestaciones de versatilidad para la configuración e implementación se logran a través de extensiones de clases, relaciones a liberías, técnicas de reflexión ("Reflection"), etc.

En el siguiente diagrama de clases UML se muestra el diseño propuesto para incorporar coordinación de contratos sensibles al contexto en Sakai. A continuación explicamos cada uno de los componentes y sus relaciones.

6.4.2 Patrones de diseño para la coordinación de contratos sensible al contexto

Figure 6.3: *Diseño del modelo de integración*

MétodoHerramineta_Interfaz: Como su nombre lo indica, esta es una clase abstracta que define una interfaz común de los servicios de una Herramienta provisto por **MétodoHerramienta_Proxy** y **MétodoToProxy_Adaptador**

MétodoToProxy_Adaptador: Esta es una clase concreta que implementa a un intermediario manteniendo la referencia a la clase *Método_Proxy* para encargarse de los pedidos recibidos. En tiempo de ejecución, esta entidad puede indicar a **MétodoServiceComponente** que no hay contratos involucrados o instrumentar una conexión entre **ConectorMétodosService** para que conecte la clase real **MétodoServiceComponente** con el contrato **ContractMétodo** en el que se encuentran las reglas de coordinación.

MétodoHerramienta_Proxy: Es una clase abstracta que define la interfaz que tienen en común **MétodoHerramientaToProxy_Adaptador** y **ConectorMétodoService**. La interfaz es heredada de **MétodoHerramienta_Interfaz** con el propósito de garantizar que ofrezca la misma interfaz de **MétodoHerramientaToProxy_Adaptador** con la cual un cliente real debe interactuar (ej., un objeto que invoque el servicio de edición de un Foro).

EditServiceComponente: Es la clase concreta donde se encuentra la lógica de la edición del Foro y la implementación concreta de los servicios originales de Sakai.

ConectorMétodoService: Efectúa la conexión entre el contrato y los objetos reales (en este caso **MétodoServiceComponente**) involucrados como partes del contrato. De esta manera, no se requiere la creación o instanciación de nuevos objetos para coordinar el mismo objeto real agregando o sacando otros contratos. Esto es posible solamente con una nueva asociación a un nuevo contrato y con la instanciación de una conexión con una instancia existente de **ConectorMétodoService**. Esto significa que hay una sola instancia de esta clase asociada con una instancia de **MétodoServiceComponente**.

ContractMétodo: Es la clase que cuya instancia genera un objeto de coordinación que al ser notificado toma decisiones siempre que un pedido es invocado a través de un objeto real. Para los casos en

[b] La adaptabilidad en el sentido que se proponen en los sistemas hipermediales que permiten personalizarse (adaptarse) en función de los usuarios individuales (Henze, 2000).

[c] Sakai mantiene la filosofía de desarrollo de Java2, de aplicaciones orientadas a servicios ("service-oriented) que pueden ser escalables, fiable, interoperable y extensible.

que no haya contratos coordinando un objeto real, el diseño de la figura 6.3 puede ser simplificado y sólo las clases y relaciones que no pasen por la zona del rectángulo gris son necesarias. La introducción de un contrato implica la creación de las clases y relaciones enmarcadas en la zona gris. La clase que implementa al contrato, llamada **ContractMétodo**, se encuentra representada dentro de la figura de un paquete UML, llamado **Contratos Sensible al Contexto**, indicando su pertenencia a un modelo subyacente descripto en la sección 6.2.

Este mecanismo de intercepción permite imponer otra funcionalidad en la interacción entre componentes del llamante (por medio de su pedido) y del llamado (a través de su respuesta). De esta manera, desde el aspecto tecnológico se instrumenta la posibilidad de brindar un nuevo "grado de libertad" a los servicios de Sakai, permitiendo establecer configuraciones propias de un DHD en tiempo de ejecución.

Por lo expuesto, es válido reforzar la argumentación sobre las ventajas de esta propuesta en comparación con las actuales soluciones tecnológicas basadas en componentes y frameworks para la inyección y representación de servicios. A su vez, se observa que el camino de su instrumentación es complejo debido a que se deben generar (automáticamente) porciones de código y creación de nuevos archivos código Java, a partir de los originales de la plataforma Sakai, que deberán ser compilados y puestos en servicio para su operatividad. Cabe destacar que los procesos de compilación y puesta en servicio de Sakai es efecutada una sola vez, cualquier tipos de cambios y configuración serán impuestas a través los contratos sensibles al contexto.

La implementación de la coordinación de contratos en Sakai requiere del uso de una herramienta específica para el dominio de aplicación (en este caso aplicaciones e-learning) que modifica los archivos de código fuente Java que implementan los servicios bases perteneciente al framework núcleo (similares a los representados en esta sección), con el fin de incorporar la infraestructura para trabajar con contratos de forma más o menos automática.

Para este propósito, se diseñó e implementó un prototipo experimental de una herramienta llamada SwC ("Sakai with Contract") y un entorno de desarrollo que permite instrumentar una metodología (3; 4) para la inclusión de contratos en Sakai bajo la perspectiva de un DHD. El desarrollo de la herramienta SwC está basada sobre el proyecto CED (Coordination Development Environment)[d] que implementa la coordinación de contrato a través de un lenguaje llamado Oblog (13). En este artículo no se describen detalles técnicos de cómo la herramienta lleva a cabo la modificación del código Java dentro de los distintos componentes de la arquitectura Sakai y cuáles fueron las modificaciones puntuales efectuadas a partir de CED.

6.5 Caso de estudio para la implementación de contratos

En esta sección se presentará un caso de uso para ejemplificar las diferentes etapas que se deben cumplimentar para lograr la inclusión de contratos en Sakai bajo la perspectiva de los DHD, retomando la idea de incluir un contrato en los servicios de edición de la herramienta Foro de Sakai.

Si bien consideramos importantes las observaciones realizadas sobre el por qué de la necesidad de utilizar una metodología concreta para el diseño del contrato referidas al lugar que ocupa dentro del flujo de ejecución de un Pe-lrn, qué tipo de requerimientos satisface, cómo fueron relevados dichos requerimientos y una especificación (en lo posible semi-formal) para su implementación, un desarrollo más exhaustivo consta en otra publicación. Dicha metodología -llamada UWATc- fue desarrollada en (4) para este mismo caso de estudio, y en (3) se amplían los detalles tecnológicos involucrados en su diseño. En esta instancia UWATc brindará información sobre: cuáles son los servicios, en qué clase (archivo Java) se encuentran representados (teniendo en cuenta el flujo de ejecución) y cómo es configurado el contrato.

Nos centramos ahora en la última etapa del ciclo de vida de la configuración de un espacio e-learning Web perteneciente al tiempo de compilación. En efecto, en esta instancia el ingeniero de software cuenta con la información necesaria para localizar los archivos de código fuente de la aplicación Sakai, los archivos de configuración de los frameworks adicionales y demás configuraciones propias de la herramienta para la inyección de los contratos.

Siguiendo con el caso de estudio de las etapas anteriores, a continuación se muestra una porción de código Java correspondiente al código fuente original de Sakai, en donde se propone un método llamado

[d] CED: es el primer prototipo de una herramienta que implementa el uso de la coordinación de contrato en aplicaciones Java. La herramienta pertenece a ATX Software (www.atxsoftware.com);fue desarrollada en Java y es de código abierto.

editMenssage correspondiente a la implementación del servicio edición de la herramienta Foro. El siguiente fragmento de código fuente Java implementa una herencia de la clase *BaseDiscussionService* donde se encuentran las interfaces de los servicios de edición correspondiente al núcleo del framework Sakai.

```
import org.sakaiproject.discussion.api.DiscussionMessage;
public class DiscussionService extends BaseDiscussionService{
public MessageEdit editMessage(MessageChannel channel, String id){
return (MessageEdit) super.editResource(channel, id);}
```

Luego, a través de la herramienta SwC se crean archivos XML (similar a la herramienta CED) para especificar las reglas del contrato que involucrarán servicios implicados en el método *editMensage* (Paso1). A continuación, se ejecuta la generación automática de código para crear dos tipos de archivos Java diferentes (Paso2). El primero (archivo 1) implementa las funcionalidades que permitan las conexión con el Proxy (ej., *MétodoHerramienta_Proxy* para nuestro caso de estudio); el segundo (archivo 2) implementa el conector *ConectorEditService* representado en la figura 6.3.

A continuación se muestran fragmentos de código que permiten una mejor ilustración de las características adquiridas a partir de las modificaciones del código fuente de la aplicación Sakai a través de la herramienta.

Fragmentos del primer archivo - (archivo 1)

1. Se importan los paquetes correspondiente al framework de coordinación de contratos (figura 6.3) y el framework context-aware (figura 6.1)

```
package org.sakaiproject.discussion.impl; import java.util.*;
import cde.runtime.*; import obab.ca.*; // Framework context
public abstract class DiscussionService extends
BaseMessageService implements
DiscussionService,ContextObserver,EntityTransferrer,ForoInterface
```

2. Métodos agregados por la herramienta para identificar las clases que van a ser interceptadas por el contrato.

```
protected CrdIProxy _proxy;
private static Class _classId= Sakai.Discussion.class;
public static Class GetClassId() {return _classId;}
public CrdIProxy GetProxy() { return _proxy; }
public void SetProxy(Object p){if(p instanceof CrdIProxy && p instanceof
DiscussionInterface) _proxy = (CrdIProxy)p; }
public void SetProxy(CrdIProxy p)  { _proxy = p; }
AccountInterface  GetProxy_Account(){if ( _proxy == null ) return null;
return (DiscussionInterface) _proxy.GetProxy(_classId);}
```

3. Método que implementa la llamada del objeto cliente al Proxy

```
public long _getNumber(){new ComponentOperationEvent(this , "getNumber")_
.fireEvent(); return number;}
public messageEdit editMessage(MessageChannel channel,String id){
new ComponentOperationEvent(this,"Edit").fireEvent();
return (MessageEdit) super.editResource(channel, id);}
```

Fragmentos del segundo archivo - (archivo 2)

1. Porción de código donde se importan las componentes de los frameworks, se heredan las clases abstracta del los conectores y proxys. La clase del objeto real *MétodoService_Componente*, según la figura 6.3, se representa a través del atributo subject.

```
package org.sakaiproject; import java.util.*;
import cde.runtime.*; import obab.ca.*;
public abstract class IDiscussionPartner extends
CrdContractPartner implements CrdIProxy, DiscussiontInterface {
protected  Discussion subject;
```

2. Definición de los métodos abstractos para la conexión (*ConnectorMétodoService*, representada en la figura 6.3), del contrato con el servicio.

```
public void SetProxy(Object p) {subject.SetProxy(p);}
protected Object GetSubject_Object() {return subject;}
public void ResetProxy() { subject.SetProxy(null);}
```

3. Métodos que permiten el acceso a los métodos que definen los servicios (ej., métodos de la clase $MétodoService_{Componente}$)

```
protected Discussion GetSubjectDiscussion(){return (Discussion) subject;}
protected IDiscussionPartner GetNextPartner_Discussion(){
return (IDiscussionPartner)GetNextPartner(Discussion.GetClassId());}
```

4. Implementación por defecto de métodos definidos en la interfaces de los servicios. A través del métodos *GetSubjectDiscussion()*, detallado en el punto anterior, se accede a los métodos creados por la herramienta en el primer archivo.

```
public void messageEdit (double amount,Customer c)_
throws DiscussionException { IDiscussionPartner
next = GetNextPartner_Discussion()
if (next != null) next.editMessage(amount,c);
else GetSubjectDiscussion()._editMessage(amount,c);}
```

5. Implementaciones de los condicionales de la reglas de los contratos que se encuentran en los archivos XML para configurarlos.

```
public CrdPartnerRules  messageEdit_rules(string texto,Student c) throws
DiscussionException, CrdExFailure { return new CrdPartnerRules (this);}
```

La generación de código automático a través de la herramienta presupone tener ciertos niveles de conocimientos sobre: el lenguaje Java, aspectos de la implementación del framework base Sakai y los frameworks que los componen; a igual que experiencias en la compilación a través de Maven [e] de Sakai y las clases modificadas-agregadas para el uso de contratos.

6.6 Conclusión

Fundamentados en el marco conceptual de los Dispositivos Hipermediales Dinámicos para educación e investigación y en las actuales limitaciones observadas en las plataformas e-learning sobre el grado de flexibilidad adaptativa en el sentido de (21) y (14)), que pueda ser inducida por usuarios expertos del dominio (ej., docentes) para el desarrollo de estrategias didácticas efectivas para el logro de objetivos pedagógicos o investigativos planteados y, constando que dicha adaptabilidad no pueda ser enteramente resulta por sistemas adaptativos inteligentes (ej., agentes, hipermedia adaptativa, sistemas expertos, etc.), es que en este trabajo propusimos una implementación de la teoría de coordinación de contrato en un framework específico sobre sistemas colaborativos Web e-learning (correspondiente al proyecto Sakai) brindando un modelo evolutivo conceptual y tecnológico. Partimos entonces, de las implementaciones de servicios y herramientas, consideramos el agregado de componentes para la representación de cierta información de contexto con aspectos context-aware para finalizar con la incorporación de contratos para la coordinación de las interconexiones de los servicios bases del framework original e-learning. De

[e] Proyecto maven: http://maven.apache.org/ref/2.0.4/maven-project/

esta manera, el contrato es una nueva primitiva prevista como una extensión de la noción de contratos de B. Meyer (11) a la que se le adjudica el rol de coordinación de las interacciones de los objetos.

El fin último de esta propuesta en desarrollo, se inscribe en brindar respuestas tecnológicas efectivas a la necesidad de promover procesos educativos e investigativos de interacción responsable entre los sujetos intervinientes en la red sociotécnica del DHD, implementando la solución en los servicios cuya prestaciones dependen -fuertemente- de reglas con estructuras volátiles cuyos condicionales son influidos por el contexto (en el sentido de "Identificación del contexto" desarrollado en el capítulo 5 (1) y manteniendo las lineamientos sobre contexto fundados por P. Dourish (20)).

Bibliography

[1] San Martín, P., Sartorio, A., Guarnieri, G., Rodriguez, G.: Hacia un dispositivo hipermedial dinámico. Educación e Investigación para el campo audiovisual interactivo. Universidad Nacional de Quilmes (UNQ). ISBN: 978-987-558-134-0. (2008)

[2] Sartorio, A., San Martín, P.: Sistemas Context-Aware en dispositivos hipermediales dinámicos para educación e investigación. En San Martín P., Sartorio A., Guarnieri G., Rodriguez G. Hacia un dispositivo hipermedial dinámico. Educación e Investigación para el campo audiovisual interactivo. Universidad Nacional de Quilmes (UNQ). ISBN: 978-987-558-134-0. (2008)

[3] Sartorio, A.:Un modelo comprensivo para el diseño de procesos en una Aplicación E-Learning. XIII Congreso Argentino de Ciecncias de la Computación. CACIC 2007. ISBN 978-950-656-109-3

[4] Sartorio, A.: Un comprensivo modelo de diseño para la integración de procesos de aprendizajes e investigación en una aplicación e-learning. Edutec2007. Inclusión Digital en la Educación Superior Desafíos y oportunidades en la Sociedad de la Información. ISBN 978-950-42-0088-8. (2007)

[5] Sartorio A. Los contratos context-aware en aplicaciones para educación e investigación. En San Martín, P., Sartorio, A., Guarnieri, G., Rodriguez, G.: Hacia un dispositivo hipermedial dinámico. Educación e Investigación para el campo audiovisual interactivo. Universidad Nacional de Quilmes (UNQ). ISBN: 978-987-558-134-0. (2008)

[6] Dey, A.K., Salber, D., Abowd, G.: A Conceptual Framework and a Toolkit for Supporting the Rapid Prototyping of Context-Aware Applications, anchor article of a special issue on Context-Aware Computing. Human-Computer Interaction (HCI) Journal, Vol. 16 (2-4), pp. 97-166. (2001)

[7] Brambilla, M., Ceri, S., Fraternali, P., Manolescu I.: Process modeling in web applications. ACM Transactions on Software Engineering and Methodology (TOSEM), in print.

[8] Andrade, L., Fiadeiro, J.L.: Interconnecting Objects via Contracts. In UML'99 – Beyond the Standard, R.France and B.Rumpe (eds), LNCS 1723, Springer Verlag (1999)

[9] Obra Abierta: Proyecto de I&D (CONICET-CIFASIS), que se centra en el desarrollo e implementación de Dispositivos Hipermediales context-aware Dinámico para investigar y aprender en contextos físicosvirtuales de educación superior. Directora: Patricia San Martín

[10] Severance, C: Sakai Project Report: The Evolution of the Sakai Architecture, disponible en: "$http://elearning.surf.nl/docs/sakai_nl/2006_07_arch_history_v01.doc$" 25-05-2008

[11] Meyer, B.:, Applying Design by Contract, IEEE Computer, 40-51. (1992)

[12] Proyecto de investigación CIFASIS-UNR llamado: "Técnicas de Ingeniería de software aplicadas al Dispositivo hipermedial dinámico. Director: Mg. Maximiliano Cristiá". Dicha propuesta consiste en la incorporación a Sakai de un framework para la coordinación de contratos (8; 19)

[13] The Oblog Corporation. The Oblog Specification Language, disponible en: "$http$: $//www.oblog.com/tech/spec.html$"

[14] Dowling J, Cahill, V.: Dynamic software evolution and the k-component model. In: Proc. of the Workshop on Software Evolution, OOPSLA. (2001)

[15] Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M.: Pattern- Oriented Software Architecture. John Wiley. (1996)

[16] Gelernter, D., Carriero, N.: Coordination Languages and their Significance. Communications ACM 35, 2, pp. 97- 107, (1992)

[17] Sartorio, A., Guarnieri, G., San Martín, P.: Students' interaction in an e-learning contract context-aware application with associated metric", Actas del INTED2007, International Technology, Education and Development Conference, IATED, Valencia, España. (2007).

[18] Gamma, E., Helm R., Johnson R., Vlissides, J.: Design Patterns: Elements of Reusable Object Oriented Software, Addison-Wesley (1995)

[19] Davy, A., Jennings, B.: Coordinating Adaptation of Composite Services. Proceedings of the Second International Workshop on Coordination and Adaptation Techniques for Software Entities. WCAT'05 Glasgow , Scotland (2005)

[20] Dourish, P.: What we talk about when we talk about context. Personal and Ubiquitous Computing, vol. 8, N° 1, Roma, 2004, pp. 19-30, disponible en http://www.springerlink.com/content/y8h8l9me8yabycl3/

[21] Houben, G.: Adaptation Control in Adaptive Hypermedia Systems. en Adaptive Hypermedia Conference. AH2000. Trento, Italia. Agosto. LNCS, vol. 1892. Springer-Verlag, pp. 250-259. (2000)

6.7 Introductions

6.8 E-Learning Frameworks

6.9 Context Aware Contract Frameworks

6.9.1 Coordination Contract Framework

6.9.2 DHD Framework

CHAPTER 7

Design Frameworks for Context Aware Contract Web E-learning Application

7.1 Introduction

Web services are standards for describing, publishing, discovering and binding application interfaces. The set of standards include a standard specification for public registries known as Universal Description Discovery and Integration (UDDI), a description language, namely Web Services Description Language (WSDL), a distributed object communication protocol called Simple Object Access Protocol (SOAP), and a dynamic, self-defining information-specification Language with semantic support known as eXtended Markup Language (XML) [4].

As a web application, portal provides a secure, single point of interaction with diverse information and business processes that is personalized to a user's needs and responsibilities. There is a good analogy between what portals add to Web applications and what window managers (like Microsoft Windows) add to operating systems (like DOS): both provide a consistent and uniform way to interact with applications. [5]

In this section, we design and model a framework that is able to comply with most of the system features outlined in Section 2. Figure 3-1 illustrates a block diagram for a multiple-tier e-learning Web services-oriented portal framework running in distributed environments. This framework includes two parts: portal framework and Web services framework: AUTHENTICATIONPortal API (java)User BeanPortal RegistryAccess ControlSOAP/XML–RPCService connectivitiesJCA ConnectorsEJBJMS(MQ)Java MailJDBCJ2EE Platform.NET PlatformOther Platform Figure 3-1 E-learning Web services portal framework

3.1 Portal Framework

7.2 Conceptual Design

7.3 Logical Design

7.4 Model-Driver development of DHD

7.4.1 Dynamic Model Driver

7.4.2 UWATc: A comprehensive design model for Web E-learning Transactions

CHAPTER 8

A Case Study of DHD Framework Implementation

8.1 Aplication Domain

8.1.1 Techologial Aspect

8.1.2 Functional Aspect

8.2 Applying The DHD Framework

8.3 Using Coordination Context Aware Contract: Evidence for use en e-learning environment

8.3.1 Learning To Contract

While the quotation from Williamson (1996) above mentions learning, transaction cost theory has not systematically incorporated learning into its framework (Williamson 1999). Indeed, the theory's equilibrium assumptions can be interpreted to imply that learning is relatively quick and thorough. While there been little study of learning to contract per se, there is a large literature suggesting that learning in general within and between organizations is an important phenomenon (e.g., Lieberman 1984; Lyles 1988; Darr, Argote Epple 1995; Argote 1999), and that it tends to be quite gradual and incremental. According to this literature, this importance is because agents' rationality is quite limited, and learning tends to occur through a relatively slow process of environmental selection of faster learners (Alchian 1950), and/or "error detection and correction in theories-in-use" (Argyris Schon 1978). This limitation implies that periods of disequilibrium, during which underperforming firms can survive in the market, can be fairly long (Winter 1988). Moreover, individuals and organizations tend to learn through the repeated practice of routines, which gradually come to embody the fruits from prior learning, but also constrain the range of new learning. As a result, learning tends to be incremental and "local", in that it tends to occur in areas "close to" areas of previous knowledge or experience (Nelson Winter 1982; Cohen Levinthal 1990; Levinthal March 1993). Several empirical studies have found evidence for this local character of organizational learning (e.g., Helfat 1994; Podolny Stuart 1996; Martin Mitchell 1998). Scholars have also noted that organizations often learn how to collaborate with each other over time (Doz 1996; Anand Khanna 2000a). For example, Child (2001) writes that one kind of learning in a strategic alliance involves "...the accumulation of mutual experience with and knowledge about how to

Numerous employees from Softstar and HW Inc. were interviewed. Interviewees included engineering project managers, engineering managers (up to the director level), programmers, marketing personnel, and quality assurance personnel. Many of these managers had extensive experience in the computer

industry. For example, the Softstar project manager in charge of working with HW Inc. had 25 years of experience as an engineer and manager at IBM. The interviews were semi-structured, with broad, general questions oriented toward understanding the role of contracts in the evolution of Softstar's relationships with its customers, especially HW Inc. As will be evident from the narrative below, we consistently asked subjects to describe the chronology and details of events that led to contract changes, rather than relying on unsupported or idiosyncratic interpretations they might offer. We also triangulated these descriptions with other interviewees (Jick 1979). With regard to the Softstar-HW Inc. relationship in particular, the project manager in charge was interviewed twelve times over a one-year period. Twelve additional personnel from Softstar and HW Inc. were interviewed from one to four times each. The combination of archival data (the contracts dating back to the beginning of the relationship) and extensive interviews provided a comprehensive picture of the relationship between Softstar and HW Inc, and a good picture of a typical relationship between Softstar and its customers in the 1990's.6 Evolution of the Softstar-HW Inc. Contractual Relationship Softstar's relationship with HW Inc. began in 1989, just one year after Softstar went public, but approximately 10 years after its founding. According to managers, during most of Softstar's history, the company was preoccupied with completing its complex software development projects, securing financing, and managing the kinds of day-to-day operational issues that are often especially pressing on small, resource-constrained firms facing tight product release deadlines. As a result, little time and attention was devoted to documenting and systematizing contracting processes. One Softstar engineering manager described Softstar's operations as having been "ad hoc" (Interview, 5/29/96). Therefore, in initiating its relationship with HW Inc., Softstar relied more on the general experience of its managers than on specific contracting experiences in its own history. However, the new project manager hired in 1994 took a very different approach to the management of the contracting process. She took on several initiatives designed to foster communication among the project managers and get them to share what they

had learned in dealing with their respective customers. The new manager adapted her previous experience to the way business was done at Softstar with a clear goal of enhancing knowledge sharing in the area of crafting and managing contracts. Softstar maintained a two-part contractual structure for all the HW Inc contracts and indeed with each of its customers during the period in question. The first part was an overall contract, negotiated by lawyers, which specified prices such as labor cost for software code development and royalty rates. While the prices stipulated in this part of the contract changed somewhat over the period, its general structure did not. This first part of the contract did not, however, obligate the parties to undertake any particular project. Once the overall contract was in place, engineering project managers from Softstar and its customers negotiated the terms and conditions of individual projects. For each project, an addendum called a Statement of Work (SOW) was added to overall contract that contained the project-specific terms. Lawyers played no direct role in the negotiation of the SOW. The changes in the structure of contracts with HW were concentrated in the SOWs. While each SOW was a legally valid part of the contract, the Softstar project manager noted that because it often took "too long to get an SOW signed, work is done prior to having a signed SOW" (Interview, 4/15/97). According to managers, this two-part structure was adopted because it was impossible in this setting to negotiate a single long term contract for supplying HW Inc.'s future embedded software needs, because HW Inc.'s future products were not yet developed, and the details of its future technical needs were therefore unknown. In addition, there was concern at Softstar that given the lack of a market price for such customized software, using completely independent contracts for each project would lead to excessive bargaining over prices, royalties, etc. The two-part contract structure helped to mitigate these problems. The first software product Softstar developed for HW Inc. in 1989 was fairly advanced for its time, but involved a very sparse SOW. This project involved the development of system software (specifically, BIOS) for HW Inc.'s computer product that incorporated some innovative features, but would not radically differentiate HW Inc.'s computer from those of competitors. The purpose of the first SOW, which was written by an engineering project manager, was clearly to define the technological aspects of the project. It was largely silent regarding management of the project, because the parties felt that once the technical details were specified, the execution of the project would go smoothly. This first

contract was only eleven pages long, seven of which were devoted to describing the product features required. Other than product features, this first SOW contained a listing of reference documents, an estimate of development time (measured in engineering days) and associated non-recurring engineering development charges, and items that HW Inc. was required to provide to Softstar to facilitate development. No delivery schedules were included. In addition, interviewees indicated that neither HW Inc.

nor Softstar contributed much knowledge from previous contracts with other suppliers in devising the contract structure (Interview, 2/4/97). The second SOW, like the first, was focused almost exclusively on technical detail. Since the first two projects occurred virtually simultaneously, there was no real chance to incorporate lessons learned from the first project into the second. Detailed measurement of the performance of the two projects is unavailable from Softstar. Testimony passed down from employees who were involved in the early projects, however, indicates that the relationship was not as smooth as either party had hoped. However, HW Inc. was looking for a supplier with whom it could build a long-term relationship, and Softstar wanted a long-term relationship in order to lock-in an important customer. The parties demonstrated their goodwill by beginning work on projects before their SOWs were signed. The first two projects were both over budget, by modest amounts, and included more features than were in the SOW. In response to these problems, the parties attempted to resolve their differences and improve their working relationship. These efforts resulted in a series of important modifications to contract structures, whose aims can be classified into four categories: (1) enhancing the communication between personnel from the two firms, (2) clarifying responsibilities and expectations of both parties, (3) planning for contingencies, and (4) format modifications. Most of the changes to the SOWs can be traced to problems encountered during the execution of prior SOWs. The details of each of these types of changes are described in detail below. These details are summarized in Table 1. Enhancing Communication Crafting effective processes for interfirm communication became an issue that Softstar and HW Inc. had to address because it was disrupting the working relationship between the two parties. The first problem the parties encountered during the first two projects was that HW Inc. was making changes based on oral authorization, and a variety of HW Inc. personnel were giving Softstar engineers different instructions. This caused confusion at Softstar and led to delays as requirements and tradeoffs were

gradually clarified. The parties introduced a clause into the third SOW that required change requests to be made in writing, in an attempt to resolve the issue by making HW Inc. personnel more accountable for changes they requested. Requiring changes in writing helped the problem of conflicting directives, but it resulted in an unanticipated side effect—delays in notifying Softstar of important changes. Forcing HW Inc. to make changes in writing made HW Inc. personnel reluctant to make themselves accountable by documenting their change requests. This reluctance resulted in changes being made at the last minute, which often forced Softstar to scramble to meet delivery dates. To address this problem, a clause was added to the sixth SOW that required HW Inc. to notify Softstar of changes "...in a timely manner." While this clause did not completely solve the problem, it did serve to highlight the issue, which facilitated earlier communication to Softstar—typically via e-mail. Requiring that requests be made in writing, however, did not eliminate conflicting requests from customer personnel. Softstar managers therefore negotiated the addition of a problem reporting procedure in the eighth SOW, which helped define how information on technical problems or changes should be communicated. While in practice Softstar sometimes accepted changes outside this process, it did not hesitate to use these concessions as leverage in negotiations over changes to cost and schedule. Such give and take typically occurred within the same project, but the project manager at Softstar indicated a couple of situations in which significant allowances in the preceding project were used as a negotiating point in the following project. The parties never actually went back and amended the present SOW to reflect changes—the changes were too frequent. They did, however, try to include processes in future contracts that would limit the potential for future disputes. Clarifying Responsibilities and Expectations A number of major disagreements and conflicts arose in the first several projects concerning the roles and responsibilities of each of the parties. The first disagreements concerned the level of detail required by Softstar in order to design and deliver the products to HW Inc. The issue here was that HW Inc. wanted to include specification development as part of the deliverable, whereas Softstar's understanding of the deal was that it would provide software code written to a specification provided by HW Inc. The Softstar project manager believed that HW Inc. was attempting to "push" specification development onto Softstar, and to otherwise "get us to do more than just write the code" (Interview,

5/28/96). To clarify this issue, clauses were added to the sixth SOW that called for HW Inc. to create a full hardware specification that spelled out minimum requirements. In the seventh SOW, a section was added that spelled out in detail the nature of the deliverables that Softstar would provide to HW Inc., including information on deliveries of source code, release notes, etc.7 Conflicts over the project release schedule, which had always been problematic, became increasingly contentious. These conflicts eventually had to be "escalated to the Director level" at both Softstar and HW Inc. (Interview, 11/4/96). The schedule referred to the release dates for the alpha, beta, and final versions for each

project.8 The first attempt to deal with scheduling was to include a clause in the seventh SOW that directed the parties to negotiate a schedule immediately after the SOW was approved. This clause proved ineffective because it simply documented what the parties had already been doing, which had led to numerous conflicts over deadlines. Such a process was problematic in part because it required the parties to negotiate well into project execution, during which time managerial focus shifted away from negotiating contractual details, and onto project completion. Therefore, negotiations were little different than before the clause was included. Realizing that a different approach was needed, the parties, beginning with SOW 8, decided to negotiate the schedule upfront, and document it in each SOW. This structure forced the parties to determine mutually agreeable delivery dates early in the coding process, before managerial attention on scheduling was lost. The new structure also led to the inclusion of a schedule for HW Inc. to deliver test platforms to Softstar. Having established the project release schedule, the firms then disagreed over the exact content of each release. Each successive release included additional features until the final release, which included everything. In order to meet the schedule, Softstar wanted to stagger the features and include about 50

This enabled Softstar to have earlier access to fully functional hardware platforms, so that it could ensure the completed system achieved the required functionality. Planning for Contingencies A key issue that arose during the early projects was the desire of HW Inc. for functionality that required Softstar to push the frontiers of certain technologies. When delays occurred during the projects as a result of technological problems related to these high-risk areas, HW Inc. expressed strong dissatisfaction with Softstar's performance. In an attempt to highlight the key technological challenges facing the project, the Softstar project manager inserted a "Risks and Concerns" section into the fifth SOW. The idea was to create a section of the SOW that listed major project risks, in order to force the participants to think through the project and attempt to identify potential risks in advance. Another problem that caused disputes was that HW Inc. would make frequent changes to project requirements that impacted the project cost and schedule. HW Inc. assumed that "minor" changes should not impact price or schedule. Softstar, however, did not always agree with the designation of a change as minor and often believed that HW Inc. should share in the additional costs created by changes made after the SOW was approved. The short development times (generally about six months) made lengthy reviews of changes infeasible. To deal with this issue, a clause was added to the eighth SOW stipulating that project cost and schedule were subject to re-negotiation whenever HW Inc. made a hardware change, problems were experienced with the performance of HW Inc.'s its hardware platform, or HW Inc. requested additional functionality. This clause, however, was not particularly successful because it did not stipulate any guidelines for providing a new schedule or price. So the ninth SOW introduced an engineering change procedure for changes required by HW Inc. after the SOW was signed. The engineering change process included an impact analysis that addressed cost and schedule impacts. As with the problem reporting process, the engineering change process was not always followed, but it provided a basis for negotiation between the parties. Modifying the Format The final series of changes were designed to facilitate review of the SOW prior to approval. HW Inc. had expressed frustration in the first two projects over the length of time it took Softstar to complete what HW Inc. perceived to be minor changes. In order to highlight key interdependencies and technological interfaces, Softstar added a system architecture section to the third SOW. This section

allowed both firms to better understand how the entire product fit together and the impact to Softstar if HW Inc. made a late hardware change. The system architecture in turn allowed the firms to better deal with key technological issues ex ante, rather than waiting until Softstar discovered the problem midway through the project.10 As with many of the other changes to the SOWs mentioned above (e.g., adding a risk section, adding a schedule) the addition of the system architecture section was aimed at pushing the decision-making process toward the front-end of the project, and at requiring HW Inc. to more carefully define their requirements earlier in the process. Another issue related to decision-making was the review of the SOW. As the relationship developed, more personnel from each firm became involved in reviewing and approving the projects. With the increase in the number of people who were peripherally involved came an increase in the review loop for both HW Inc. and Softstar. To facilitate review, the Softstar project manager added a revision history section to the fifth SOW so that all communications referencing the SOW could include the current revision level. This section prevented managers from unwittingly approving an outdated version of the document. Other Characteristics of the Relationship Managers and engineers reported that the relationship between Softstar and HW Inc. generally improved over time, and was driven more by its own internal dynamic than by influences outside the relationship. The parties rarely referred to the contents of the SOWs

in day-to-day operations. The Softstar project managers did not report a sense of operating by rules in which each firm would jump to point out deviations from the procedures set forth in the SOWs. Rather than using the clauses in a legalistic sense, the parties tended to refer to them only when they needed to provide some give and take. For example, in one instance HW Inc. needed more time to provide an engineering platform to Softstar, but wanted Softstar to keep its final delivery date. When a clause from the SOW was mentioned, it was usually by the person who was breaking it. For example, in the case of the late engineering platform, the HW Inc. engineer called the Softstar project manager, acknowledged the agreed-upon due date, and

apologized for the fact that it would be late. He then asked Softstar to work overtime to complete its portion of the task according to the regular schedule. The Softstar project manager eventually agreed, but the with a clear understanding that Softstar was doing HW Inc. a favor that it might redeem from HW Inc. at a later date. This kind of give-and-take became easier as the additional clarity in the SOWs made it clear who was asking for special consideration. In earlier projects, the parties often disagreed over whether something was standard or required special consideration. Another important feature of the Softstar-HW Inc. relationships was that the trust between the two parties appeared to grow over time. As noted earlier, work on a new project often began before the new contract was signed. Moreover, the Softstar manager in charge of the HW Inc. relationship stated in 1997 that he trusted the HW Inc. managers "more with each passing project"; that he "trusts them to pay", "trusts their intentions", and feels that they "operate in good faith"(Interview, 6/19/97). Other managers and engineers shared this view, and explained that it was easier to build trust when the SOWs were more detailed, because misunderstandings were avoided, and agreement on expectations, roles and responsibilities of the parties was facilitated (Interview, 6/19/97). Finally, as the contracts' administrative structure developed, there appeared to be very little reference by either party to other contracts each had with other parties, or to any influences from experiences gained from those other relationships. In part this may have been due to a lack of explicit processes and incentives for searching for and capturing any such knowledge. For example, there was no explicit component of project managers' salaries or bonuses that had any relation to sharing or acquiring knowledge from other project managers. One project manager mentioned that he had too much to do to be trying to figure out what was happening with other project managers (Interview, 5/29/96). Summary Data Figure 1 graphically depicts changes in a several variables related to the contracts. The horizontal axis shows the SOWs over time, from the first SOW in 1989, to the eleventh SOW in 1997. For each SOW, the vertical axis measures project costs, and the total number of SOW pages, and the pages devoted to administrative and technical matters, which is a useful indicator of the attention each category is receiving in deliberations and negotiations between the parties.11

Unfortunately neither Softstar nor HW Inc. kept detailed measures of the performance of each project. We did, however, ask the project managers that we interviewed to assess the performance of the projects in which they participated qualitatively. Their assessments clearly indicated that performance increased significantly over time. Many problems were encountered during early projects, but both parties were more satisfied with the performance of the later projects (Interview, 6/19/97). The Contracting Environment While the PC industry as a whole was experiencing the rising power of Microsoft and Intel, and computing efficiency continued to increase dramatically, there were no major shocks or trends that substantially influenced the contracts between Softstar and HW Inc. Softstar personnel indicated that events taking place in the broader industry had little or no effect on its bargaining power with HW Inc., or on other factors that could influence the terms of their SOWs with HW Inc. (Interview, 2/4/97). There were also no major changes regarding competitors, technological standards, or coding technology that had much of an influence on the contracts during this period. There were likewise relatively few changes within Softstar that influenced the contracts with HW Inc. Softstar was growing and adding new clients throughout the sample period. There was some turnover in personnel, although no more than the typical Silicon Valley firm experienced during this period. Three different project managers worked with HW over the sample period. Interviews reported relatively smooth transitions between project managers. The engineering manager who was in charge of the project managers was relatively "hands-off" during most of the sample period. New Policies As noted above, Softstar did not attempt to apply any lessons from its own contracts with other partners to its contracts with HW Inc. At the very end of the period a new, more hands-on engineering manager was hired, who began to make changes that were beginning to be phased in as our data were being collected. Some of these changes were informed by the new manager's experiences in her previous job at another firm, and were aimed at creating more formal processes for sharing experiences within and between projects (Interview, 5/8/97). For example, she began calling regular meetings of the various

project managers, using them as forums to share status reports on customer relationships. In addition, she

instituted a policy of conducting more detailed post mortem analyses of completed projects and codifying and distributing the results of these analyses.

8.3.2 Theoretical Interpretations

Learning to Contract The contractual relationship between Softstar and HW Inc. clearly falls into the category of bilateral governance (Williamson 1985). Both parties desired an ongoing supply relationship, and were willing to make sustained efforts to maintain the relationship in the face of disturbances to it. Moreover, it is clear that Softstar was making significant idiosyncratic investments in the relationship. All software development for HW Inc. was completely customized to their requirements and hardware platform. Had any of the projects been cancelled before completion, Softstar's investments up to the point of cancellation would have been at risk of essentially total loss. The contracts were not, however, "life-or-death" deals for Softstar. Transaction cost theory would suggest, therefore, that the degree of asset specificity was high enough to require complex contracting, but not high enough to require vertical integration (Williamson 1991). It is less clear from the data whether HW Inc. was making relationship-specific investments, but HW Inc. would have experienced costs associated with delays involved in switching to another supplier. The interview data show how the structures of the contracts changed over time as the parties gained experience working together. In particular, as disagreements and problems arose in the context of one project, adjustments would be made in contracts negotiated for subsequent projects. Thus the primary driver for change was not potential problems, but actual problems that the parties encountered during projects. The changes did not always follow in the next SOW, but severe and/or persistent problems were eventually incorporated into the SOWs for future projects. These adjustments were made with the precise aim of preventing similar disputes from arising in the future, and/or to provide a contractual basis for resolving such disputes. The interview data thus strongly suggest a pattern of learning to contract. The interview data also suggest that the Softstar-HW Inc. relationship involved different but related types of learning beyond learning how to write effective contracts per se. Indeed, one reason why the changes to the contracts are so theoretically interesting is because they reflect the learning that was

going on the broader relationship between the two firms. For example, as the relationship progressed, the two firms gradually learned how each other operated (their internal organization structures, decision-making styles, etc). This knowledge enabled them to eventually incorporate contract terms that took such factors into account. The single-point-of-contact clause added in SOW 9 would be an example of this type of learning. Other added clauses reflected learning about how to work together that was relatively independent of the internal decision-making structures of each partner. Some of the project scheduling clauses added in SOWs 7 and 8 might reflect this type of learning. It seems clear, therefore, that the relationship between learning to contract and learning how to collaborate was very close in this case. The firms thus could not learn to contract with each other without also learning how to work with each other. The character and speed of the learning-how-to-contract is also of particular theoretical interest. As the relationship proceeded over the period, the parties did not make serious efforts to anticipate how future contingencies that could disturb the relationship, or at least failed to gauge the probability and/or severity of the disturbances that did eventually arise. For example, the early contracts did not include terms aimed at mitigating problematic consequences of such disturbances. These contracts did not provide for any dispute resolution procedures, or even much basis or guidance for negotiating such resolutions, despite the fact that at least one party (Softstar) was about to make non-trivial relationship-specific investments. Even more significantly, however, the major disputes over specification development responsibility during SOWs 1 and 2, though they had important ramifications for the distribution of project costs between the parties, did not lead them to immediately develop something close to the "optimal contract" to address a wider range of such contingencies (say, in SOW 3). It would appear that the early disputes did not stimulate contemplation of, or at least serious efforts to deal with, a broader range of possible future contingencies, and incentive misalignments to which they might give rise. The parties did not, for example, make provision for disturbances over release scheduling and content, change definitions, etc. that arose in later contracts. If such disturbances were considered at all, the probability of their occurrence, and/or the severity of their effects, was underestimated. And once again, as these later disputes arose, subsequent contract terms were devised to address them only ex post. At no time during the design of each of the 11 contracts did the parties

attempt to arrive at a relatively inclusive and stable set of administrative procedures for governing similar projects into the future.

Moreover, even after adding contract terms aimed at better incentive alignment, dispute prevention and dispute resolution, the parties were sometimes unable to anticipate problematic side effects of these terms, and had to further adjust them in subsequent contracts. For example, disputes led to the addition of simple terms calling for renegotiation when engineering changes are required. But these terms proved inadequate, and explicit engineering change procedures had to be added, though these did not emerge in full form until SOW 9. In essence, then, the parties proceeded quite gradually and incrementally during the contracting sequence, responding to disputes one-by-one. Though each new contract negotiation provided an opportunity to attempt a more inclusive contracting approach, only after actually experiencing a dispute did the parties attempt to address it in later contracts. Indeed, major modifications to the governance features of the contracts were being made as many as 9 years after the relationship began. This slow, incremental learning process thus bears a closer resemblance to processes described in evolutionary theories of organizational learning (Nelson Winter 1982) and behavioral theories (Cyert March 1963) than to the kind of foresighted contracting contemplated in transaction cost theory. Indeed, the learning process in the Softstar-HW Inc. relationship seems well described by Cyert March's (1963) notion of "problemistic search", in which learning is motivated by the search for solutions to immediate problems, rather than by long-term planning needs. Thus, organizations would tend to limit search to "the neighborhood of the problem symptom" (p. 122) as "they move from one crisis to another" (p. 102). One reason for the incremental nature of the learning may have been the perceived success of the projects. Managers explained that despite the problems that arose during the execution of the projects, most projects were considered to have been moderately successful. Starbuck Hedberg (2001) argue that successful outcomes often cause managers to be reluctant to embrace significant changes, for fear of creating a worse outcome. While moderate levels of failure can draw attention to potential problems (Sitkin 1992), they may not generate a sufficient level of urgency to overcome organizational inertia, which acts to delay or even counteract problem solving (Hedberg 1981). A second explanation for the incremental nature of learning offered by interviewees is that the engineers who managed these contractual relationships for Softstar faced a variety of challenges in their daily jobs that prevented them from taking the time to plan for the future. In particular, the engineers were under strong time pressure to meet the technical challenges involved in the projects, and to complete the 21

development work in a timely way. Projects were occurring in quick succession, as HW Inc. raced to introduce new computer products into a market in which product lifecycles were very short. As a result, engineers reported allocating less of their attention to the organizational and contracting issues. In addition, the Softstar project manager in charge of the HW Inc. relationship indicated that Softstar was usually understaffed due to strong growth. Engineers worked very hard to keep things going on a daily basis, and had therefore little time to reflect about future contract contingencies or which of the possible problems were most likely to occur. This implies that the farsightedness assumption that transaction cost economics employs may not always hold for resource-constrained firms operating in high-velocity environments.

Lack of Knowledge Spillovers The interview data suggest that the source of the learning was almost entirely internal to the contractual relationship. Managers did not report, for example, that either Softstar or HW Inc. learned about potential contingencies and/or their management from their contractual relationships with other contractual partners. The Softstar project manager clearly indicated, for example, that HW Inc.'s role in bringing new knowledge to the contracting process from experience with other suppliers was virtually nonexistent. Even more surprising, it appears from the interview data that, until the very end of the period, project managers within Softstar were not sharing their contracting experiences with each other, at least not in very extensive, formal ways. The data indicate that at the end of period, the new engineering manager was making efforts aimed specifically at promoting more intrafirm learning of this kind. Thus, the interview data suggest that in the absence of explicit policies aimed at stimulating managers and engineers to learn from other contracting experiences, it may not happen at all. Finally, managers and engineers also reported having little time to attend industry association meetings in which contracting knowledge might be shared across firms. This finding regarding the lack of effort to capture knowledge spillovers might be somewhat surprising. Managers explained this lack of effort as resulting from the extreme time pressures they faced in striving to complete complex software development projects on time and within budget. Another possibility is that the parties expected the learning to be mostly local and partner-specific, and therefore not very applicable across

relationships. Zollo, Reuer and Singh (2002) found, for example, that more partner-specific experience was associated with better alliance performance in their sample. Similarly,

Gulati (1995) argued that more partner-specific experience increases partner-specific trust, which again would limit the utility of learning from different contractual partners. Moreover, the fact that software development remained relatively ad hoc at Softstar (and at most other firms: see Cusumano 1992), with few standardized procedures, probably made wide applicability of contracting experiences more difficult. We discuss other, related effects of this feature of software development on learning to contract in the next section. Learning: Communication, Codification and Governance As suggested in the interview data, provisions were added to the contracts between Softstar and HW Inc. for purposes of communication, codification and governance. With regard to communication, provisions requiring written notification of engineering changes, project scheduling, and disclosure of information about system interactions, and the like. These additions were aimed at achieving better information flow between the parties, so as to avoid coordination failures and other honest mistakes due to miscommunication. With regard to codification, the SOWs served to record lessons from previous experiences in the relationship, and did so in a single document that was the guiding reference for dealing with uncertainties or disputes between the parties. Such codification helped avoid ambiguity, and prevented "organizational forgetting" (Argote 1999, Ch. 2). With regard to governance, several provisions were added that were clearly aimed at preventing behavior that was self-interested, if not opportunistic. For example, provisions requiring complete specification were clearly added to deal with what were perceived by Softstar managers to be self-interested attempts by HW Inc, to shift project costs onto Softstar. Similarly, after the written notification provisions were added in SOW 3, further timeliness requirements were added in SOW 6 to respond to delays caused by a perceived lack of effort by HW Inc. Many provisions no doubt served governance, production and communication purposes simultaneously, such as those concerning project scheduling (aimed at both avoiding hold-up and communicating plans), and those defining "major" and "minor" engineering changes (aimed at reaching common understanding, and avoiding self-serving definitions). While it is difficult to separate the relative importance of governance, communication, and codification in the Softstar-HW Inc. contracts empirically, it appears, consistent with transaction cost theory, that the governance functions were quite important. We draw this conclusion because it would seem that that the information and administrative procedures that were written into the contracts could

easily have been communicated and codified in other ways. For example, the parties could have communicated face-to-face more frequently or for longer periods. They also could have developed a single, non-legally binding document file to which each could refer when questions of procedure arose. The fact that the parties insisted on incorporating all the administrative procedures into a legally binding contract suggests that the parties were quite concerned with defining the transaction carefully in order to reduce the potential for opportunistic interpretation of the contract.

Alternative Explanations

Transaction cost theory might suggest an alternative explanation for the pattern of contracting observed – one that does not rely on learning effects at all. For example, it is clear from Figure 1 that the contracts struck between Softstar and HW, Inc. tended to become longer over time, as contract terms and conditions were gradually added. This lengthening tendency was not a smooth one, but was established clearly by SOW 7. Figure 1 also shows that total project cost began to increase monotonically by SOW 7. Recall that because each software development project undertaken for HW Inc. was completely customized, total project cost is highly correlated with the degree of asset specificity in the transaction. Therefore, a plausible alternative explanation for the expansion of contract terms aimed at preventing and resolving disputes is that Softstar (and possibly HW Inc. as well) had an increasing amount of relationship-specific investment (i.e., the development cost of the project) at risk of expropriation through hold-up. Under this explanation, the contracts were lengthened only when the level of relationship-specific investment reached a minimum threshold, such that the parties became willing to spend more time and attention on contract design.12 This alternative explanation, however, cannot explain away the learning effects completely. First, as Figure 1 shows, the close association between total project cost and total number of SOW pages covering administrative procedures only begins at SOW 7. For SOW 's 1-6, this association does not exist. Therefore, the importance of learning effects during at least the first three years of the contractual relationship cannot be ruled out with this alternative explanation, although it may carry explanatory value

for later contracts. Secondly, this explanation does not take into account the interview data, which indicate a learning process through to SOW 11. Even if engineers only turned their attention to contract development when they thought it worthwhile, recall that they still often failed to identify a satisfactory remedy for a problem the first time. Indeed, they occasionally failed multiple times before landing on an acceptable solution. A related explanation might focus on specific investments related to how to work together, rather than to the product-related investment. Early in the relationship, little was at risk but the current project. However, once the parties invested in processes to work together effectively – processes that may have been partly partner-specific – a disturbance leading to the termination of the relationship would have carried a much higher cost. This is because the investments in the relationship itself would have been lost. While this consideration may have been a factor, it does not receive strong support from the interview data, nor can it explain the link between problems in one SOW and clauses to deal with that specific issue in later SOWs. This alternative explanation implies that the firms were anticipating contingencies in the new contracts, whereas the interview data suggests that the parties were actually reacting to past contingencies in writing those contracts. Moreover, these reactions continued to be incremental, and sometimes ineffective, even late in the relationship. It is also important to note that communication considerations alone do not explain the increases in contract length observed. Such considerations might suggest that longer, more detailed contracts were caused by increases in the complexity of the projects, since more complex projects require more communication procedures in order to solve more difficult coordination problems. However, while it is the case that the parties were undertaking increasingly larger projects that required more product features, the fact that the administrative sections of the SOWs, which dealt with processes and responsibilities, were growing faster than the sections of the SOW that dealt with technical requirements is evidence that learning and governance considerations were important as well. In addition, if complexity alone was driving the changes to the SOWs, then the addition of new contractual clauses would have been driven by communication challenges in contemporaneous transactions, and perhaps by the anticipation of increasing complexity in future transactions. The interview data demonstrate, however, that new clauses were responses to problems in earlier contracts. While it is likely that complexity played a role in affecting the evolution of contractual provisions

(especially in SOWs 8–11) learning and governance considerations were evidently important driving forces in this evolution.

8.3.3 Implications For Theory Development

Contracts as Knowledge Repositories One of the key findings from our study of the Softstar-HW Inc. relationship is that over time the contracts between them came to serve as repositories of knowledge about how to efficiently work with each other. As the parties worked together over the years, they encountered a series of problems and challenges that were not addressed in the contract that prevailed at the time. As these collaboration problems were gradually resolved, the solutions were gradually incorporated into later contracts, such that these later contracts eventually came to codify the parties' knowledge about efficient ways to collaborate. Thus, the written contracts enabled later projects to benefit from experiences gained in earlier collaborations, thereby improving the performance of those later collaborations. Traditional views about the role of contracts in the economic, sociological, and legal literatures have not paid much attention to this possible role of contracts as knowledge repositories. As mentioned above, economic theories of contracting such as transaction cost economics have paid scant attention to learning in general. Sociology-based views of contracting have also been focused elsewhere. As noted earlier, many organizational scholars have tended to downplay the importance of contracts and related formal structures in facilitating economic exchanges, emphasizing instead the roles of trust and social ties in exchange processes (e.g., Macaulay 1963; Larson 1992; Ghoshal Moran 1996; Uzzi 1997). As a result, the attention of organizational scholars has not been focused on contracting processes in general, and therefore not on processes of learning to contract. Traditional legal scholarship on contracting has also not focused on processes of learning to contract. Such scholarship has been highly influenced by Llewellyn's (1931) classic work, in which he advanced the concept of a contract as a flexible framework of "yielding rules", rather than "iron rules" for managing commercial relationships. This view stood in contrast to previous emphases on the legalistic contract forms and "black letter" contract law. Llewellyn explained that many key provisions of business contracts are relatively open-ended, vague and/or of dubious legal enforceability in order to allow parties the flexibility they need to "get the job done" with as little recourse to the courts as possible (so-called

"private ordering"). Macneil (1974, 1978) also took this view of contract as framework, developing the distinction between transactional and relational modes of contracting. This tradition of legal scholarship emphasizes the role of contracts as frameworks whose provisions are flexible to changes in circumstances as the underlying relationship moves forward in time. While this approach certainly appears applicable to the contracting relationship studied in this paper, it does not address the key phenomenon of interest here; namely, the addition of entirely new provisions, and the transformation of older ones, as a commercial relationship develops through a series of relatively short-term contracts – where these changes in rules are aimed at preventing conflict in the exchange process.13 Our findings therefore imply that it may be fruitful for organizational scholars to broaden our views of the role of contracts as flexible frameworks or governance structures, to include a notion of contracts as knowledge repositories – in at least some sets of circumstances. We begin to describe some of these circumstances below. The role of contracts as repositories of organizational knowledge has also not been fully appreciated in evolutionary economics, although it is quite consistent with theories in that tradition. Evolutionary economics has tended to focus on intra-organizational routines as the main locus of firm capabilities (Nelson and Winter 1982), although scholars have recently emphasized inter-organizational routines as another possible locus of organizational knowledge. Dyer and Singh (1998) argue that knowledge-sharing routines are one form of relationship-specific capital on which parties can earn "relational rents" from an alliance. Contracting processes, our evidence suggests, might in some instances serve to help develop and codify such knowledge-sharing routines. Moreover, it is well established that transferring knowledge within or between organizations often requires at least some codification (e.g., Nonaka Takeuchi 1995). By providing a means for this to occur, contracting processes could facilitate the development of such relational capital, in the form of a "collaboration capability". Zollo and Winter (2002) recently discussed the codification of organizational knowledge as a critical step in the creation of firm-level capabilities in general. Kale, Dyer and Singh (2002) found that firms possessing an "alliance capability" had better success with alliances than firms that did not.

When Contracts May Become Knowledge Repositories

When are contracts most likely to play this role as knowledge repositories? It is still early in this line of research to attempt to state a definitive set of conditions, but the fieldwork does suggest some possibilities. One of these is technological uncertainty. It is clear from the descriptions above that the managers and engineers in question were dealing with very high levels of technological uncertainty. One source of such uncertainty involved the management/technical processes. During much of the period in question, there were still were few techniques, routines, and guidelines established for managing software development projects in general (Cusumano 1992). The production process for software remained highly individualistic, which frustrated attempts to capture economies of scale. The difficulty Softstar was having capturing such economies by developing code-reuse techniques illustrates the relatively immature state of software development management practices. Moreover, the category of customized software products produced by Softstar was less than 10 years old. This uncertainty about management/technical processes made it especially difficult for managers and engineers to anticipate the kinds of problems and disagreements that would come up during development processes, and to identify procedures that would be effective in preventing them. Even if an array of potential problems could be foreseen, identifying those that were most likely to occur was problematic. This suggests that the role of contracts as knowledge repositories will be a more important one for transactions involving complex and innovative goods or services that are customized, than for transactions involving simpler goods or services whose underlying technology is well understood. A second possible condition for contracts to serve as knowledge repositories is when there is an important interaction between technological uncertainty and technological interdependence. In the early personal computer industry, there was significant uncertainty as to the kinds of innovative code that would control client hardware in desired ways; that is, uncertainties regarding the technical relationships between the software and the hardware. These uncertainties regarding compatibility were exacerbated by the fact that HW Inc. was innovating its hardware systems. Therefore, it was especially difficult for the parties to anticipate the kinds of engineering changes that might be required as projects proceeded, and to develop effective engineering change procedures and specification requirements descriptions, until some experience had been acquired. Thus, the need for precise compatibility with the buyer's product, and the innovative nature of that product, combined to make learning to contract especially important. This

8.4 Applying The DHD Mothologogy

8.5 DHD: Experimental Prototipe

CHAPTER 9

Conclusions

CHAPTER 10

Further Works

Bibliography