A model for extending legacy applications by implementing new requirements as components

Nicomedes L. Cavalcanti Júnior  Roberto Souto Maior de Barros  Mlexener Bezerra Romeiro
Centro de Informática - CIn / UFPE, Av. Prof. Luiz Freire, s/n
Cidade Universitária, CEP: 50740-540, Recife-PE, Brasil
{nlcj,roberto,mbr}@cin.ufpe.br

Abstract

This paper presents a model for creating applications by joining components and sending data in XML format. The model focuses on legacy applications and ones that are hard to be maintained. In order to analyse it, an implementation of some of its elements has been carried out and used for two experiments.

1 Introduction

Back in 1968, at NATO’s Software Engineering Conference, McIlroy [15] advocated the software production “industrialization”. At that time, he had already perceived the necessity of creating software that is easily adaptable to various types of situations, such as pre-molded structures from the civil construction industry. This would allow the development of new systems since not everything would have to be created from scratch.

The software industry has advanced noticeably since that conference. However, software production by joining prefabricated bits is still not a common practice. Additionally, the advancements in the area could have been more effective if - in the last few years - more effort had been put in the production of new software, rather than keeping older versions.

The legacy software is at work in various places, being responsible for important activities. Although it is not meeting the current demands, such as integration to the Web or integration with other systems, one solution such as remaking this software in more modern languages is out of the question due to high costs involved. The principal cost in disposing of the legacy systems is the loss of knowledge (business rules, for example) and functionality. Such loss would take place because in many cases there is no source of information about the legacy systems other than the source code.

The changes in business as well as in information and communication technologies that can be used to support the business processes are occurring continuously. These changes cannot be seen separately since, in the case of e-commerce - started due to the technological development of the World Wide Web - one technological advancement made way for new forms of doing business. This way, companies must try to be flexible and integrate changes in their businesses and technologies. The legacy software was
not conceived to support the demands that the new forms of business and technology have brought about [12, 18]. Adapting legacy software to the context of business and technological requirements is a bottleneck.

McIlroy's idea, in this context, has taken form and has received attention under the name of components; Meyer [6] defends that development based on components is the ideal solution so that the software industry can continue delivering more and more complex software, as is expected of it. It is through components that new applications are being developed. Likewise, the techniques that aim at adapting legacy software are investing on components, to allow these to meet the new requirements.

This paper proposes one framework to allow creating and keeping component-based applications through data exchange in the XML format. The focus of this paper is legacy applications or applications to which one may want to add functionality with as little changes as possible in the main application. The main idea is to have the legacy as one component and provide implementation to the new requirements as components that interact by exchanging data. The assumption is that we can keep the legacy working in its usual environment and use it to provide others components with data. The advantage of this data is that it is a result of the business rules which the legacy system knows. Having as an input the data that just the legacy system can provide a new separated component can provide the new requirement that would be hard or even impossible to implement directly in the legacy system due to technological limitations for example.

The rest of the article is organized as follows: section 2 presents detailed background of the problem and of the state of the art; section 3 presents a model that will be the framework base; section 4 introduces a framework implementation; section 5 reports experiments carried out to verify the utility of the proposal; section 6 displays a discussion on the proposal, and section 7 presents the conclusion and ideas for future works.

2 Background

Even in the area called Component-based Software Engineering there is considerable variation as to the definitions, although these are not obvious. Szyperski [1] presents one discussion on these definitions. The definition used by Bachmann et. al. [3] is very consistent with the definitions proposed by Szyperski [1]. Thus it is the latter that will be dealt with here: "A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties."

Software component technologies in general reflect the project pattern under which the system is described at a high level of abstraction by the types of components in the system and their interaction patterns (something very close to an architectural style), which are shown graphically in Figure 1.

Figure 1 shows one reference model for component concepts. In this figure, number one (1) represents one component that represents one software implementation which can be executed in one physical or logical device. One component implements one or more interfaces (2) and this means that the component satisfies certain obligations which are named contracts (3). These contractual obligations guarantee that independently developed components obey certain rules and can be used in one construction environment and pattern execution (4). One component-based system is formed by a distinct set of components, all of which play a specialized role in the system (5) and are described by one interface (2). One component model (6) is the set of all types of components, their interfaces and, additionally, the pattern of interaction allowed among the types of components. One component framework (7) supplies one range of services in the time of execution (8) to support and reinforce the model of components [3].

Some of the advantages that component technology brings about to development are: to offer one repository of basic services such as transaction control, data persistence, safety; to allow the use of components made by third parties or "shelf"; to allow software updating optimization since, being non-monolithic, only the components that evolve will need to be delivered; to reduce the time of software development and at the same time to improve the software quality (components have been validated and tested) produced through reuse of existing components; to create a new market: components can be seen as a way of packaging value, since knowledge over some domain can be transformed in one specific component and sold to third parties to be used by them.

To form a system from components, these must interact. This requires a discipline that the components must follow so that they can communicate. In the reference model in Figure 1 two types of entities can be identified: components and frameworks. Literature defines three main types of interactions present in component-based systems: Component-Component (C-C) - this is the composition that allows interaction among components: application functionalities come from these interactions; Framework-Component (F-C) - way of composition which allows interaction between one framework and its components: this way of interaction allows the framework to manage the components’ resources; Framework-Framework (F-F) - way of composition that allows interaction between frameworks: this way of interaction allows composition of components that are implanted in heterogeneous frameworks.
One type of contract is associated with each of these forms of composition among entities. These are, respectively, application level contract, system level contract and interoperability contract [3].

2.1 Legacy application scenario

Modern technologies to develop components are already available, but they have drawbacks such as requiring a great quantity of computer resources, and the learning curve necessary to start using them.

As the legacy systems have been developed in simpler languages than the current ones, they perform satisfactorily in computers with little computational resources. Thus, although the use of components may enable the integration of legacy applications to new business requirements like e-commerce, utilizing more modern technologies may mean high expenses for the acquisition of new machinery and training.

The argument above that the legacy applications do not need - by nature - machines with many computational resources to perform satisfactorily brings up a question: would it be possible to build legacy applications with components in a simpler way, or with computer requirements similar to those available nowadays for these applications?

The aim of this paper is to present a framework that implements a simpler model for the creation of component-based applications without compromising its own advantages. The model proposed here aims at legacy applications and environments with computational power that is inferior to currently used machines.

3 Proposed model

The model described below deals with the interaction among components. In Figure 2 one generic component-assembled application is presented. The goal is to exemplify the elements of the model at use. The types of interaction that can be represented in this model are C-C and F-C both presented in section 2. The possible modeling way is the application’s component interaction mode when the components are in execution. In other approaches, the concern is with the modeling during project time, for example.

The components that form one application are organized in moments. A moment is one part of the application in the sense that it is responsible for performing a set of actions. They are organized in chronological order, that is, moment 1 is executed before moment 2, and so on. In Figure 2, one moment is represented by the Expression Moment 1, for instance, and by the line that separates one expression - as the one above - from the other. The main purpose of the moment concept is to show the chronological order in which actions must take place. For example, if a system for electronic commerce had to be built using this model a logging in component that would check whether or not a user would be an authorized one and if so would redirect him or her to a proper interface according to the user preferences would be a component placed in this system model first moment. The components in the second moment, for instance, would be ones able to provide the different user interfaces this system would like to offer.

Each moment can have several components. The component is represented in figure 2 by the shadowed square with a text in the center which is formed by letter C and one more number that serves as a unique identifier.

The components that are placed in the same moment are considered parallel, which can be interpreted as absence of dependence between them. There can be dependence (connection) among the components in different moments; this is expressed as a line joining one component to another in figure 2. The line represents that the components will exchange information through XML files, typically one’s output will become the other’s input.

When components exchange information as described in the paragraph above, it is possible to insert some transformation between the information origin component and the destination component. The goal of the transformation is to
make the information - that is being passed by the source - compatible with what the destination expects to receive. In figure 2, the transformation operator was represented by one little square over the line that joins the components.

In figure 2, the transformation operator was represented by one little square over the line that joins the components.

Figure 3. Proposed model description using UML class diagram

It is possible that there is more than one component at the output of one connection. It is also possible that only one of them be used at a time. In this case, a selection operator is introduced. According to the information that is being exchanged, the destination component should be selected. In Figure 2, the selection operator is represented by one triangle over the lines that join components in Moment N to components in some moment before that.

Finally, there is one operator to control the flow of the moment execution. As was said, the natural order is that in the flow of execution of moments after Moment N, one Moment N+1 be executed. However, making use of the flow control operator, the moment to be executed after the one in which the operator is can be different from the one defined by the standard order. In Figure 2, this operator is represented by a circle, which appears in Moment N, and is connected to the outlets of both components of this moment. This shows that the choice of the next moment can be based on execution data.

The structural aspects of the model are represented in Figure 3 below using the UML (Unified Modeling Language) notation [25]. The diagram used was the class diagram. Component and connection are shown with two different associations in order to represent two things: first that one component can be associated with no connection; second that when a connection occurs it has to involve at least two components one that is responsible for creating an output data and other one that takes this input data as its input data.

4 Framework implementation

In order to validate the idea in this article, a framework that permits executing other applications in parallel and/or in series and connecting them by means of XML files was created. The objective is to provide a helping mechanism to the search of better conclusions.

The framework implementation is called ConnectApps - Connecting Applications with XML, and was developed in Java, using the Jbuilder8 [14]. Its conception is based on the idea of project, in which the user must provide the necessary data about the various applications that he/she wishes to execute and interconnect. For example, using a friendly interface the user can set the applications he/she wants to execute at each moment, inform the transformation from an output XML file into an input XML file.

In the implementation of the ConnectApps DOM [7] trees were built to store the applications' data, making use of the JAXP [8]. Figure 4 presents the DOM parser’s work flowchart, using JAXP.

So as to make possible saving the data specified by the user in an XML file, as well as opening the saved project’s XML file, APIs TrAX [9] e XPath [10] were used respectively.

5 Experiments

In this section, two experiments, whose objective is to demonstrate the utility of the framework, will be presented. In both cases, a problem is posed by the need to implement a new requisite in an application. However, altering its source code is not a desired solution because its maintenance is difficult (one change can introduce errors in the other functionalities).

5.1 Commercial legacy system

Take a company’s sales and inventory control system, which has been operating for about twenty years and was written in an old language like COBOL or Clipper. The company desires to expand its business, getting closer to clients. For this reason, the system was changed some time ago in order to enter and store clients’ email addresses.
The legacy system, in this case, has been working satisfactorily. Its maintenance record shows that it normally breaks the usual stability of work, be it for introducing changes in the usual system behavior or even errors. Then, it is believed that changing something in this system is expensive because there are few specialized companies and it is time-consuming, due to the complexity and lack of system documentation.

Using the framework presented in this article, the implementation of this new requisite that can be called “intelligent electronic mailing” can be done in a way as to not modify the legacy system. It is only necessary to implement one component that will receive the data that the legacy system can supply and make it do what the company wants: send email to the clients suggesting products.

In figure 5 two components are presented, which were created to implement the functionality in question. One component, the one that appears in moment 1, is the encapsulated legacy system. By encapsulated we mean a component that uses the legacy system to extract the necessary information and puts it in an XML file. The component that appears in moment 2 processes this information and sends the mailing.

5.2 New system, but without documentation

In this experiment, the system in question is a data clustering application. To cluster data is to divide the data (also called individuals or patterns) into groups so that the most similar ones stay in the same group. Popular clustering algorithms are fuzzy clustering (each individual belongs to one cluster with a certain degree of pertinence) and hard clustering (each individual is in only one single cluster). In these two algorithms one central element is the distance measure to be used; there are various measures proposed in literature.

This way the tool in question has the following functionalities: it performs hard and fuzzy clustering on one database. It permits that one among various distance measures be selected as it clusters one base. These functionalities are useful for evaluating the performance of different algorithms and different distances on one base.

Another set of functionalities serves to measure the statistical performance of one distance measure. These functionalities are: generation of artificial bases following principles of a Monte Carlo experiment; cross-validation, which is repeating the execution of one algorithm several times, each time part of the data base is used. This way, it is intended to verify if the algorithm is reaching results influenced by the data in question. Finally, the tool generates a lot of statistical information about one algorithm performance.
This application was developed in C++ and does not have any documentation besides the source code. Thus, as the source code is modified to insert new functionality, it is easy to modify some other functionality by maintenance side effect. There are 43 files in all, around 30 classes and nearly 20,000 code lines.

Some of the distance measures that can be used for clustering have the characteristic of being applied to a database whose data type is only at intervals. One interval is considered something in the form \([a;b]\) in which \(a\) is less or equal to \(b\). By the definition of an interval, some punctual datum such as 10 could be transformed in intervals, so becoming \([10;10]\). What is desired is to add one functionality that permits using these distance measures with a numeric database that is not just at intervals.

Because of the properties of the tool, direct maintenance in its source code is not a simple task. In this case, the use of the proposed framework can help to solve the problem. For this, it must be considered that some numeric data can easily be transformed into intervals. The problem is then reduced to previously processing the base, and the data that are not interval-based must be converted into intervals. Every new functionality can be developed independently from the change in the current application; it is enough to implement one new component that will verify the base and will make the necessary conversions and, after that, pass the altered base on to the clustering application. In figure 6 the two components used to implement the functionalities in question can be seen. The component in moment 1 implements the base transformation. If there is some non-interval-based numeric datum, it will convert it to intervals, and this will open the door to another component in moment 2, which is the system that was already encapsulated as a component.

6 Discussion

In the experiments above there were systems that were working but which presented difficulties in being modified. Developing new functionalities in the form of components has proved to be an interesting alternative since it is possible to implement new functionalities without running the risk of altering the existing system or introducing errors.

The framework does not determine in which language a component must be implemented. The flexibility in the language choice is important because one legacy system can work in a machine that does not support the execution of an application developed in a more modern language.

The most outstanding use of the proposal presented here is for cases in which the changes can be made in a more local way, that is, changes that do not reflect on various points of one system. In the legacy system of the experiment, if a client’s shopping profile had to be used in several other points of the sales and inventory system, it would not be possible to implement it without altering the system itself. This question shows that the ideal situation to apply the presented approach is the one in which the existing system is stable, that is, it does not go through significant changes in relation to the business it serves.

The use of XML as a link between the components has advantages such as: having several technologies whose development is based on this pattern, integrating more easily to other systems since XML is increasingly adopted for data exchange. In the case of experiment 2, a lot of information is generated as a result of data clustering. This information comes basically in the form of indexes and their statistics. Then new typical functionalities in this system are supposed to modify the outlet report, giving more emphasis to some information. This type of functionality, provided that the outlet tool be an XML file, can be easily performed by another component applying only some other technology to XML, such as XSLT or XPath.

7 Conclusion and future works

In this paper, a new model of interaction among components, and the implementation of a framework as part of this model were presented. Components were used for they are considered to be a form of developing higher-quality applications due to some characteristics such as allowing reuse. XML was defined as the path of data exchanging because this representation pattern is being more and more used, and
also because of the technologies available for the manipulation of XML data. The focus of the paper was legacy applications or applications that are difficult to maintain. One important research activity is to use approaches to guarantee that this software lives long enough to continue meeting the new technological and business requirements. The main contribution of the present article is the proposal of a new model in order to use the concept of components together with legacy applications and thus achieving the goal of joining the legacy software and the new technological requirements to meet the market's current demands.

Points for future works are:

- Implementing the model as a new language. The implementation of the framework was graphic. There is an advantage if we choose to implement the model as a language, since several different graphic tools can be made to use it;

- The model gives little emphasis to the F-C type of interaction. For this reason little service is offered by the framework. One service that could be offered would be an XML parser. This way, in one application, all components could use and share the same implementation, reducing the code size of each component;

- About the XML parser implementation, an interesting area to explore would be the implementation of several types of parsers, that is, implementing parsers in the most common and oldest languages, implementing parsers in old computers that are still used in legacy systems;

- The model does not specifically deal with the fact that the components are distributed, but this is not forbidden either. It would be interesting to explore the explicit extension of the model for distributed components.

- Developing the contract-explicit concept using XML, for example, and this way proposing algorithms to validate during the project if one set of components can be connected. The validation can occur from the data compatibility check that one component can supply and the data that another can receive. The incompatibility check can serve for another paper: the proposition of automatic transformations to transform incompatible components into compatible ones.

References


