JOURNAL OF OBJECT TECHNOLOGY

Online at www.jot.fm. Published by ETH Zurich, Chair of Software Engineering ©**JOT, 2004**

Vol. 3, No. 4 (April 2004)
Special issue: TOOLS USA 2003

Orion – A Component-Based Software Engineering Environment

Daniel Lucrédio, Calebe de Paula Bianchini, Antonio Francisco do Prado, Luis Carlos Trevelin, Federal University of São Carlos, Brazil
Eduardo Santana de Almeida, Federal University of Pernambuco – Recife Center for Advanced Studies and Systems, Brazil

Abstract

Software Engineering Environments (SEEs) have been extensively studied, aiming to provide help in software development. Component-Based Software Engineering (CBSE) arises as an approach for constructing software through reusable components, reducing development costs and time, among other benefits. This paper identifies the main requirements for Component-Based Software Engineering Environments (CBSEEs), and presents Orion, a CBSEE which integrates different works from our research group, involving development tools, a repository, a process model and a middleware platform.

1 INTRODUCTION

CASE tools have existed since the first days of computing. Text editors and command-line compilers are among the first examples of tools designed to help programmers. Later, they evolved to more complex environments, automating tasks such as compiling and debugging. However, these first environments, called PSEs (Programming Support Environments), supported only coding activities [Harrison 2000]. The next tool generation came to support other activities, such as analysis and design. Today's environments, called SEEs (Software Engineering Environments) cover a wider range of the software life cycle [Harrison 2000, Sommerville 2000].

Meanwhile, Component-Based Software Engineering (CBSE) is a continuously growing field. However, there is a lack for good environment support for the CBSE activities, mainly because most of the environments do not offer comprehensive, integrated support for the full range of CBSE activities [Lüer 2001]. The construction of this kind of environments involves general environment issues, such as tool integration, and specific CBSE issues, such as domain engineering and application composition.

This paper presents a Component-Based Software Engineering Environment (CBSEE), called Orion, which integrates different works from our research group, including an UML modeling tool, a Java programming tool, a network tool, a middleware platform and a process model. Although not originally designed to work together, they could be integrated into an environment that fulfills most of the requirements for CBSE.

The paper is organized as follows: Section 2 presents the identified requirements for CBSEEs. Section 3 presents the different works from our research group that compose the Orion environment. Section 4 presents Orion, which is briefly evaluated in section 5. Related works are presented in section 6. Section 7 presents some concluding remarks.

2 REQUIREMENTS FOR COMPONENT-BASED SOFTWARE ENGINEERING ENVIRONMENTS

Authors such as [Silveira 2002] state that requirements for component-based software systems are difficult to identify, due to the complexity of the involved factors. However, several researches found in the literature attempt to identify requirements related to Component-Based Development [Pressman 2001, Silveira 2002] and CASE Environments [Almeida 2002b, Lüer 2001, Pressman 2001, Sommerville 2000]. Next the main requirements identified for CBSEEs and their rationale are presented.

Tool integration

CBSEEs are composed by a set of tools that must interoperate in order to help the Software Engineer through the development process. This interoperation can be achieved in different levels [Sommerville 2000].

Platform integration: The tools that compose a CBSEE must run on the same platform, so they can benefit from the same platform services.

Data integration: The tools within a CBSEE must share information along the software process, producing and using information that is common to the environment.

Presentation integration: When using a CBSEE, the Software Engineer must use different tools. It is easier for one to learn how to use a tool if it has the same appearance than the tools which he is already familiar with.

Control integration: A tool can control the operation of other tools.

Process integration: There is a process model to guide the Software Engineer through the usage of the tools.

Support for Component-Based Software Engineering activities

A CBSEE must give support to two major activities: **Domain Engineering**, which produces components for future reuse, and **Component-Based Development**, which produces applications that reuse existing components [Szyperski 1998].

Domain Engineering, also called development "for reuse", deals with components construction. Based on problem domain requirements, components are constructed for reuse. In order to aid in Domain Engineering, a CBSEE must support domain analysis, components design, implementation, packaging and publishing.

Component-Based Development, also called development "with reuse", deals with the applications composition, reusing existing components. If needed, new components can be developed, or even acquired from third party. To be effective, a CBSEE must provide means to find, connect and adapt existing components. It must also support the addition of components that are developed or acquired during this phase.

Reusability

In order to achieve effective reuse, the Software Engineer must be able to avoid all types of effort duplication. Not just code, but every kind of software artifact should be reused [D'Souza 1998]. An example is design patterns application [Gamma 1995]. Each time a pattern is applied, the same structure must be built. If this structure could be reused, the Software Engineer would only need to build it once. In order to support reusability, a CBSEE must offer ways to recover and reuse resources at any time or abstraction level.

Referential Integrity

In CBSE, reuse usually causes several components to reference each other. However, due to the evolutionary nature of software systems, components are constantly added, updated and removed from their location. Thus, a CBSEE must guarantee that every reference has its integrity assured. This is similar to the referential integrity present in most DataBase Management Systems, with the difference that in a CBSEE these references can involve different kinds of data, from simple text artifacts to complex high-level models.

Software Configuration Management (SCM)

Software Configuration Management can be defined as the ability to control the changes that naturally occur during the software process. The objective is to assure that the software evolves in an ordered way, reducing the confusion between the Software Engineers. This is an extensive subject, involving many tasks, such as versioning and modification control. In CBSE, these tasks must be performed either to control the changes in the components as the changes in the applications.

Multiple Views of Information

The information within the environment must be semantically rich, so that all tools involved can read and understand it in its own way. This is specially true in CBSE, where a component can be viewed internally (functionality), or externally (architectural).

Security

Security is a major issue in any software engineering approach, and so it is in CBSE. To be reliable, some information must be proven secure. Otherwise, it could risk the whole project. The idea is that only authorized developers should be able to access and modify the information, avoiding damage to the project.

Technology and Language Independence

Although extensively studied, software components are still evolving, and there is no definitive solution. New component-based technologies are constantly arising, such as EJB [Sun 2002], DCOM [Microsoft 1996], CORBA [OMG 2002b] and JavaBeans [Sun 1997]. A CBSEE must maintain a basic conceptual core, independent to any technology or language, responding to the novelty without being dependant on it.

3 COMPONENTS OF THE ORION ENVIRONMENT

Our research group has proposed several works in the software engineering area, including CASE tools, a software artifacts repository, a middleware platform, and a software process model. Next, these works are briefly discussed.

MVCASE

The MVCASE (Multiple View CASE) [Almeida 2002b] is a modeling tool that provides graphical and textual techniques based on UML notation [Rumbaugh 1999]. Being fully implemented in Java, it is platform-independent. MVCASE allows the Software Engineer to specify systems in different abstraction levels and four views: Use Case View, Logical View, Component View and Deployment View.

To persist the UML specifications, MVCASE uses the OMG's XMI (XML Metadata Interchange) standard [OMG 2002a]. The XMI is a XML-based format used to represent UML descriptions. Based on the specifications, MVCASE generates XMI descriptions. This standard allows MVCASE to share models with other tools.

In order to aid in the implementation activities, MVCASE provides wizards to automate tasks such as code generation, packaging and component publishing. Until this moment, these wizards support the following technologies: Java 2, EJB [Sun 2002], CORBA [OMG 2002b] and Java Servlets [Sun 2003b]. MVCASE is currently being used in several Brazilian institutions, helping in other researches and teaching.

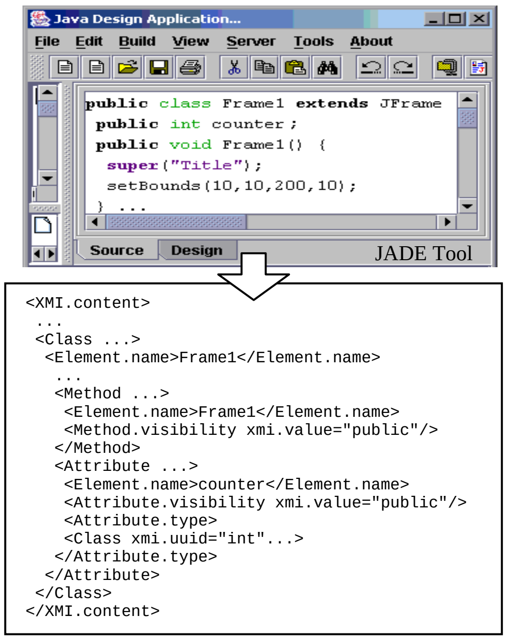
JADE

JADE (JAVA DESIGNER), is a Java-based, object-oriented programming tool with features that help the Software Engineer in the coding activities. Similar to most of the commercial programming tools, such as JBuilder [Borland 2003a] and Delphi [Borland

2003b], JADE has support for coding, compiling, executing and debugging applications. It has also support for Graphical User Interface (GUI) design.

The major different between JADE and the commercial tools is its ability to persist the edited classes in XMI format. JADE has a Java parser which retrieves the information from the code, identifying data such as attributes names and operation signatures. Based on these elements, XMI descriptions are generated. To describe GUI-related information, the XMI standard was extended. Figure 1 shows an example of JAVA code described in XMI.

This allows the information recovered from the code to be viewed by modeling tools that can read XMI, such as MVCASE. JADE is currently a prototype under development, being part of an ongoing MSc. dissertation.


Java Design Application...
File Edit Build View Server Tools About
Source
Design
JADE Tool

```
public class Frame1 extends JFrame
 public int counter ;
 public void Frame1() {
  super("Title");
  setBounds(10,10,200,10);
 } ...
```

```
<XMI.content>
 ...
 <Class ...>
  <Element.name>Frame1</Element.name>
   ...
   <Method ...>
    <Element.name>Frame1</Element.name>
    <Method.visibility xmi.value="public"/>
   </Method>
   <Attribute ...>
    <Element.name>counter</Element.name>
    <Attribute.visibility xmi.value="public"/>
    <Attribute.type>
    <Class xmi.uuid="int"...>
   </Attribute.type>
  </Attribute>
 </Class>
</XMI.content>
```

Figure 1: Code information described in XMI

MoDPAI

MoDPAI [Bianchini 2002] is a tool to monitor devices using pervasive computing issues and intelligent agents. Pervasive computing [Hansmann 2001] is the technology that allows the access to the Internet through mobile devices, such as digital cellular phones. MoDPAI uses software agents [Franklin 1996, Lucena 2002] to help the network administrator in his decision-making, speeding up the monitoring and devices control. The exchange of information between the administrator and agents can occur directly in the tool or through mobile devices via Internet.

MoDPAI meets the specifications described by the SNMP (Simple Network Management Protocol) [Stallings 1999], a network management standard based on TCP/IP [Tanenbaum 1996]. By meeting the SNMP standard, MoDPAI tool can monitor a network to collect information about distributed devices. This information is analyzed by the software agents that are programmed in a knowledge base. According to the intelligence degree, which is predefined by means of clauses written in the KB language [Lumina 2000], the agents make decisions that change the devices' configurations.

In order to be able to monitor a network, the administrator must identify and register all the devices to be monitored. MoDPAI provides mechanisms to automatically search the network for devices. This search is performed either by looking up devices that have the SNMP management software installed, or by looking up devices that respond to PING requests [Stallings 1999]. After identifying the devices, MoDPAI exports the information to XML. Figure 2 shows this process. This ability to identify the network topology and export it to XML makes MoDPAI an important part of Orion, as will be explained later.

MoDPAI tool was the subject of a MSc. dissertation [Bianchini 2002], and has several other functionalities, related to network monitoring, but that do not relate to the subject of this paper. Further information may be found in [Bianchini 2002].

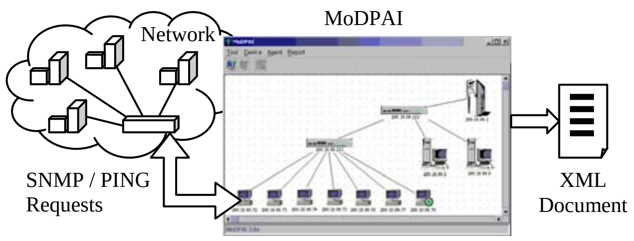


Figure 2: Automatic devices identification

JAMP Platform

JAMP (Java Architecture for Media Processing) is an environment for the development of cooperative multimedia applications [Guimarães 2000, Souza 2001]. It is composed by several servers, a broker (a specialized server that servers as a bridge between different clients and servers) and a framework set that is used by the multimedia applications. Based on the object-oriented paradigm, JAMP uses the Java language, the RMI [Sun 2003a] and CORBA models for the construction of cooperative multimedia applications.

The JAMP platform architecture has three layers. In the first, called **Application Layer**, there are several cooperative multimedia applications (chat, whiteboard, etc). These applications use the platform services through access frameworks, which define an API that allows the interaction between the applications and the platform services.

The second layer, **Services Layer**, comprises all the multimedia and distributed middleware services, being responsible for remote method invocation on distributed objects and media processing. One interesting service available on JAMP is the Load Balancing service, which allows method invocations to be redirected to different objects, to reduce of the communication overhead that occurs when a single object is excessively accessed. In this layer is also located JBroker, which allows clients to invoke methods on remote objects without knowing details of their location or implementation.

The third layer, called **Infrastructure Layer**, provides communication transparency for the distributed applications and the available services. Java/RMI and CORBA are examples of the models supported by this layer.

The JAMP platform is used according to the following process: The process starts with the server (an object that provides a service) registering itself in JBroker. When a client needs a service, it accesses JBroker to search for it. Next, the JBroker returns to the client a reference for an object that implements the service. The client can then invoke the service directly on the server. This process is known as Trading Process.

If a single server is excessively accessed, there may be a communication overhead. JAMP solves this problem through its Load Balancing Service. This service allows different copies of a server to be registered in JBroker. When a client needs a service, only one copy is selected, according to some policy or algorithm, reducing the overhead. Examples of such policies include: round-robin selection, random selection, among others. JAMP is the result of several MSc. Dissertations, including [Guimarães 2000, Souza 2001], and it is being currently used and improved with new functionalities.

Repository

The software development process involves great amount and variety of information. These include requirements, graphical analysis and design models, annotations, drafts, and executable code. These information constitute an important knowledge base of the software development process, influencing directly in its quality.

In CBSE, the management of this information is even more important, requiring efficient storage and search mechanisms to promote reuse. For this reason, a repository is being developed, with mechanisms to remotely store, search and recover software artifacts that are produced during development.

The repository is based on the XMI standard. Since the scope of the information that can be represented in XMI is not fixed, the repository extends it to represent other kinds of information, such as project execution information and version control. Figure 3 shows the architecture of the software artifacts repository. The repository has mechanisms for version control, which means it can control different versions of a given artifact. It has also security and transaction mechanisms, to assure the consistency and reliability of the artifacts.

The services for storing and searching software artifacts are available through a middleware. By doing so, these services can be accessed via a network. The artifacts are transferred in XMI, being physically stored in this format. By doing so, any tool that is able to import and export XMI may use the repository, through the middleware layer. Currently, this middleware is JAMP, which is compatible with Java/RMI and CORBA.

Using the repository, the information produced during the software development process can be stored and managed, becoming available for later reuse. The repository is currently under development, using multi-agents systems technologies and search engines, being part of an ongoing MSc. dissertation.

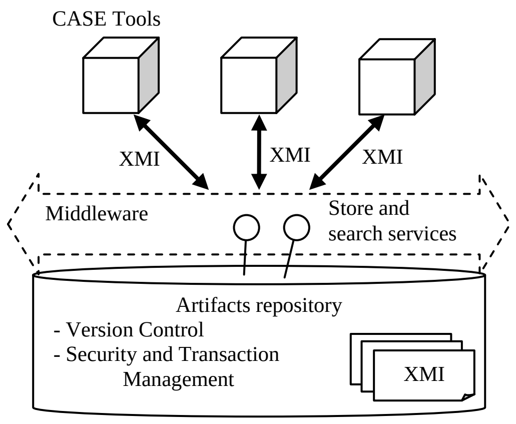


Figure 3: Software artifacts repository architecture

Incremental Process Model (IPM)

One of the most compelling reasons for adopting component-based approaches to software development is the premise of reuse. The idea is to build software primarily by assembling and replacing interoperable parts. The time reduction and improved product quality achieved make this approach very attractive [D'Souza 1998].

In order to make reuse effective, it must be considered in all phases of the software development process [Heineman 2001, Jacobson 1997, Rumbaugh 1999, Szyperski 1998]. Therefore, the Component-Based Development (CBD) must offer methods, techniques and tools that support components identification and specification, in a problem domain level, and their design and implementation in a component-oriented language. Besides, CBD must use interrelations among existent components, which have been previously tested, aiming to reduce the complexity and the development costs.

Current CBD methods and approaches, such as the ones discussed in [Jacobson 2001, Perspective 2000, Rumbaugh 1999], do not include full support for the concept of a component. The result is that components are handled mainly at the implementation and deployment phases. The methods are significantly influenced by their Object-Oriented origins, while trying to introduce the CBD concepts mainly through the use of standard Unified Modeling Language (UML) [Boertin 2001, Rumbaugh 1999, Stojanovic 2001].

In this context, motivated by ideas of reuse, component-based development and distribution, an Incremental Process Model (IPM) [Almeida 2002a, Almeida 2003a] was developed, as the subject of a MSc. dissertation [Almeida 2003b], to support the Distributed Component-Based Software Development (DCBD).

IPM was divided into two stages. In the first stage, the process starts from the requirements of a problem domain and produces implemented components in an object-oriented language (development "for reuse"). Once implemented, these components are stored into a repository. In the next stage, using the search mechanism offered by the repository, the Software Engineer consults the available components of a given problem domain, and develop the applications that reuse them (development “with reuse”). Figure 4 illustrates IPM using SADT notation [Ross 1997]. An experiment was conducted, to compare IPM with an ad-hoc approach. The results, as shown in [Almeida 2003b], indicate that some gains are achieved with IPM, such as reduced development time and increased number of documentation artifacts.

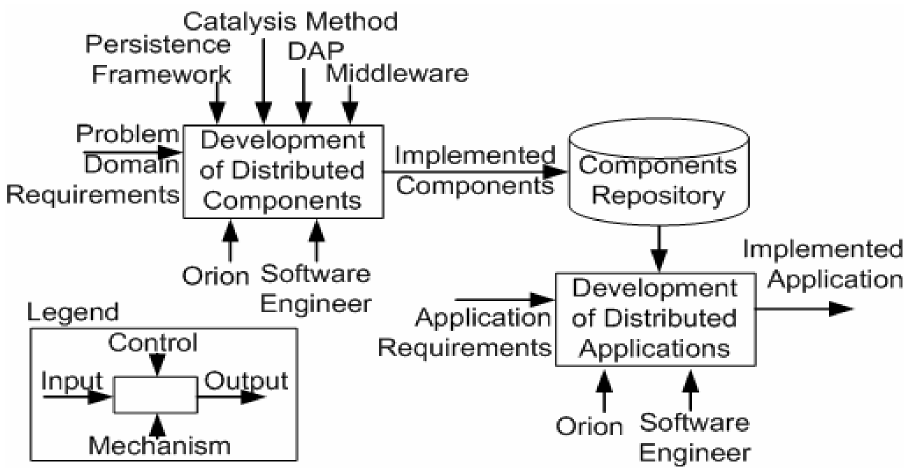


Figure 4: Process model for DCBD

4 THE ORION ENVIRONMENT

Orion covers a substantial part of the software lifecycle, with tools that aid in the specification, design, implementation and deployment tasks. Figure 5 shows Orion’s architecture.

Guided by IPM, the Software Engineers use the tools in different development tasks. The artifacts produced by the tools during the process are stored in the repository. The tools and the repository are integrated through JAMP platform, thus operating over a network.

This architecture allows cooperative work, since multiple Software Engineers can work together in a same project, using different workstations and sharing information through the repository. The version control and security mechanisms provided by the repository helps to maintain the information consistent and reliable.

Each tool of the environment has specific features, and contributes with the Software Engineer in different ways. However, the most important component of the environment is the IPM, which guides the Software Engineer during the usage of the environment, providing a well-defined approach for constructing component-based software. For this reason, the steps of IPM will be followed to explain how Orion works.

Following is a detailed presentation of each stage of IPM and how Orion is used in each one, using the Service Order problem domain of a computer company as an example. The Service Order (SO) domain applications are divided into three big modules. The first one, Customers, is responsible for registering and notifying customers of a certain service order. The second one, Employees, is responsible for registering employees and controlling service order tasks. The third one, Reports, is responsible for emitting several reports to the manager.

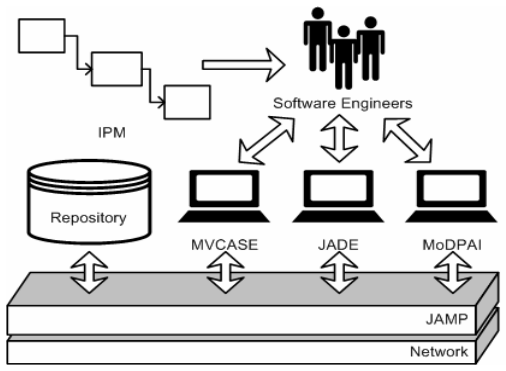


Figure 5: Orion Architecture

Development of Distributed Components

In this stage, components of a problem domain are built in four steps: Define Problem, Specify Components, Design Components and Implement Components.

In the first step, **Define Problem**, emphasis is placed on understanding the problem and specifying "what" the components must do to solve the problem. Initially, the domain requirements are identified, using techniques such as storyboards and mind-maps [D'Souza 1998] to represent the different situations and problem domain scenarios. Next, the requirements are specified in Collaboration Models [D'Souza 1998, Rumbaugh 1999], representing action collections and the participant objects. Finally, the collaboration models are refined into Use Cases Model [D'Souza 1998], aiming to reduce complexity and improve the problem domain understanding. MVCASE tool is used in this step. Through its graphical and textual capabilities, the Software Engineer constructs the mind-maps, collaboration and use cases models. The first step is summarized in Figure 6, where MVCASE screenshots show *mind-maps*, defined in the Service Order domain requirements identification, a *Collaboration Model*, and the *Use Cases Model*.

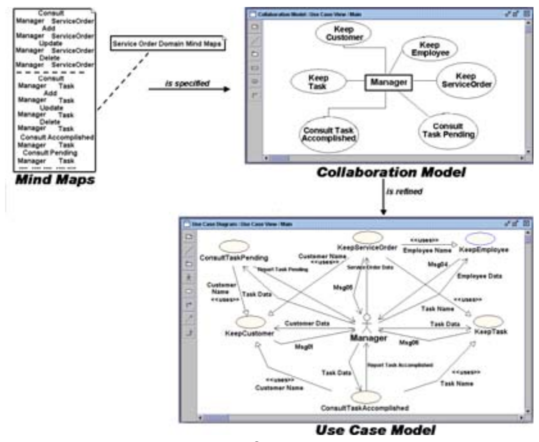
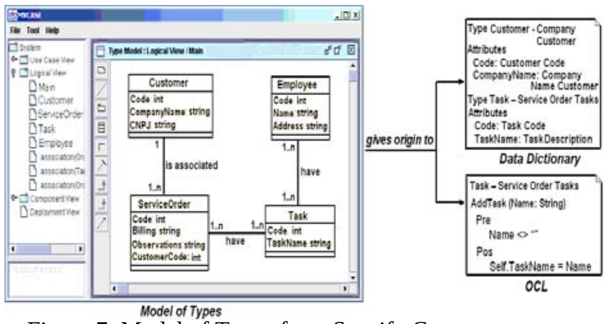


Figure 6: Define Problem step

In the second step, **Specify Components**, the system's external behavior is described in a non-ambiguous way. Again, MVCASE is used, to refine the previous specifications, obtaining the components specifications. The *Model of Types* is specified, according to Figure 7. Object types and attributes are defined, without worrying with implementation. Still in this step, the data dictionary can be used to specify each type, and the *Object Constraint Language* (OCL) [D'Souza 1998] to detail the objects behavior.



```
Type Customer - Company
                Customer
Attributes
 Code: Customer Code
 CompanyName: Company
              Name Customer
Type Task – Service Order Tasks
Attributes
 Code: Task Code
 TaskName: TaskDescription
```

```
Task – Service Order Tasks
AddTask (Name: String)
 Pre
   Name <> ""
 Pos
   Self.TaskName = Name
```

Figure 7: Model of Types from Specify Components step

Once identified and specified, the types are put together into *Model Frameworks*. Model Frameworks are designed at a higher abstraction level, establishing a generic scheme that can be imported, at the design level, with substitutions and extensions in order to generate specific applications [D'Souza 1998]. The fact that the Model Framework is small, thus narrowly focused, increases its reuse potential in a well-defined application domain. In addition, a Model Framework is a reusable asset at the design level. As a design represents much of the major decisions that go into finished code, it can specify frameworks at design level and offer a process to refine them down to the code level. Figure 8 shows a MVCASE screenshot with this model being edited. It is then stored in the repository, becoming available for future reuse.

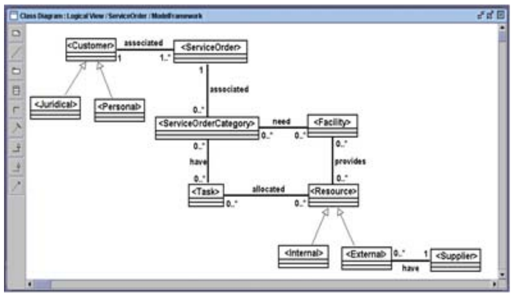


Figure 8: Service Order Model Framework

The types with names written between brackets are defined as *placeholders* [D'Souza 1998]. These types can be substituted in the specific application. The concept is similar to the extensibility of classes of the object-oriented paradigm. In Orion, this is achieved by retrieving the framework from the repository and using MVCASE to apply it to an application domain, as shows Figure 9. When the framework is applied, the placeholders are substituted by their respective types.

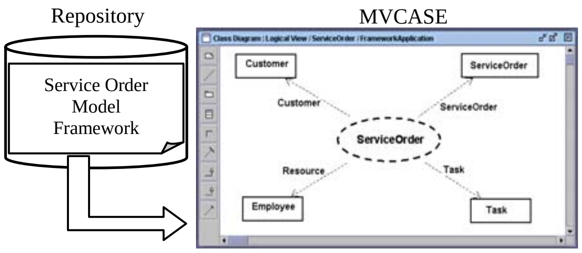


Figure 9: Service Order Framework Application

Still in this step, the *Use Cases Models* from the last step are refined through *Interaction Models* represented by sequence diagrams [Rumbaugh 1999], to detail the utility scenarios of components in different problem domain applications.

In the third step, **Design Components**, the Software Engineer performs the components inner design and specifies non-functional requirements, such as: distributed architecture, fault tolerance, caching, and persistence. First, the *Models of Types* are refined into *Classes Models* [Rumbaugh 1999], where the classes are modeled with their relationships, considering the components definitions and their interfaces. The previous Interactions Models are also refined to show details of method behavior in each class.

Next, the non-functional requirements are incrementally specified. For the distribution non-funcional requirement, IPM uses of the *Distributed Adapters Pattern* (DAP) [Alves 2001]. DAP is a combination of the Facade, the Adapter, and the Factory design patterns [Gamma 1995]. DAP introduces a pair of object adapters [Buschmann 1996] to achieve better decoupling of components in distributed architectures. The adapters encapsulate the API for allowing distributed or remote access of business objects. In this way, the business layer becomes independent from the distribution layer, so that changes in the latter do not impact on the former [Alves 2001].

To apply DAP, the Software Engineer uses both the repository and MVCASE. The repository stores the DAP structure, which is imported into MVCASE and adapted. Figure 10 shows an example, with DAP being applied to the *Customer* component of the Service Order Domain.

The *Source* and *Customer Target* components abstract the domain business rules. *ICustomer TargetInterface* abstracts *Customer Target's* behavior in a distributed scenario. At this interface, the components *Source* and *Customer Target* do not have communication code either. These elements form a distribution-independent layer. *CustomerSourceAdapter* and *CustomerTargetAdapter* are connected to a specific distribution API to encapsulate the communication details. *CustomerSourceAdapter* isolates the *Source* component from distributed code. It is located in the same machine that *Source* and works as a proxy to *CustomerTargetAdapter*. *CustomerTargetAdapter* is located in another machine, isolating *CustomerTarget* from distribution-related code.

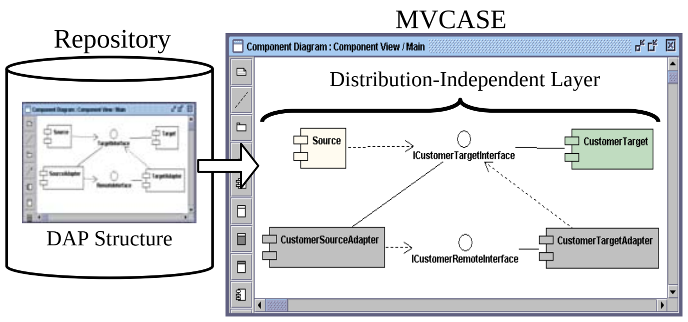


Figure 10: DAP application using the repository and MVCASE

Once the distribution requirement is specified through DAP, other non-functional requirements can be specified. In this example, database persistence will be specified, through the *Persistence* framework [Yoder 1998], shown in Figure 11. The *ConnectionPool* component, through *IConnectionPool* interface, manages the database connection. The *DriversUtil* component, based on XML, has information about the supported database drivers. The *TableManager* component manages the mapping between objects and tables. The persistent component of the *FacadePersistent* structure, through its *IPersistentObject* interface, is responsible for treating incoming requests.

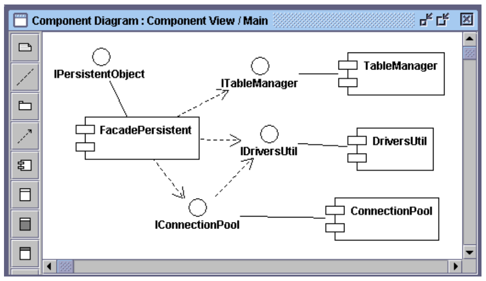
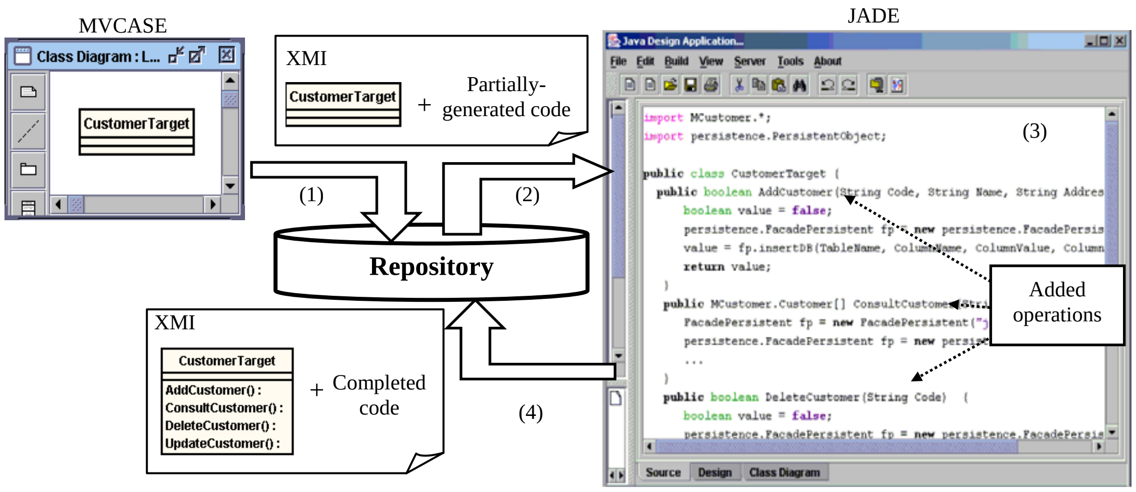


Figure 11: Persistence Framework

In the last step of the first stage, **Implement Components**, the Software Engineer defines the distribution technology. In this example, CORBA was chosen. Next, he uses a code generator from MVCASE to generate part of the executable code, such as the classes hierarchy and structure, with attributes, operation signatures and references. The components design and the generated code are then stored in the repository, in XMI.

To complement the partially-generated code of the components, the Software Engineer uses JADE tool, which has features to compile, execute and debug executable code. As mentioned earlier, the major advantage of using JADE instead of the similar commercial tools is that it can write high-level XMI descriptions, which can then be read in a modeling tool, helping to maintain the consistency between models and code.

Figure 12 illustrates this process. First, the Software Engineer uses MVCASE to generate part of the component code, storing it together with the component design in the repository (1), in XMI. Next, he imports the code into JADE (2), and completes it (3). Note that in this example some operations were added into the code. JADE generates XMI descriptions of the modified code, which are then stored back in the repository (4). Note that the operations added in (3) are now present in the class description (4).



```
import MCustomer.*;
import persistence.PersistentObject;

public class CustomerTarget {
  public boolean AddCustomer(String Code, String Name, String Addres
      boolean value = false;
      persistence.FacadePersistent fp = new persistence.FacadePersis
      value = fp.insertDB(TableName, ColumnName, ColumnValue, Column
      return value;
   }
   public MCustomer.Customer[] ConsultCustomer(Stri
      FacadePersistent fp = new FacadePersistent("j
      persistence.FacadePersistent fp = new persist
      ...
   }
   public boolean DeleteCustomer(String Code)  {
      boolean value = false;
      persistence.FacadePersistent fp = new persistence.FacadePersis
```

Figure 12: CustomerTarget Component being implemented. Since JADE is able to write in XMI, the changes are automatically reflected into the model

Once the components are stored in the repository, the Software Engineer goes to the second stage of IPM, where applications that reuse those components are developed.

Development of Distributed Applications

The application development starts with the application requirements identification and proceeds with the normal life cycle development, which includes: **Specify**, **Design**, **Implement**, and **Deploy Application**. For a better comprehension of these steps, an application to register a customer via web is used as an example. This application reuses the components of the Service Order (SO) domain, built in the previous stage.

The first step, **Specify Application**, starts with the problem understanding and the identification of the application requirements. Before the requirements specification starts at MVCASE, the Software Engineer imports the components of the related problem domain, in this case SO, available in the repository and that will be used in the application. Next, the requirements are specified in Use Cases and Sequence Diagrams.

In the second step, **Design Application**, the specifications from the previous step are refined by the Software Engineer to obtain the design. In this step, the non-functional requirements related to distributed architecture and data persistence are specified. Figure 13 shows a components diagram of the application, where the persistence framework is reused.

Once the application is designed, the Software Engineer goes to the next step, **Implement Application**, where he uses MVCASE to generate part of the application's code and complement it with JADE.

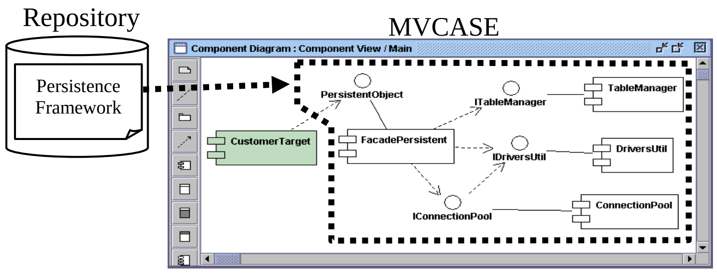


Figure 13: Application design with reuse of the Persistence Framework components

The implemented application can then be deployed, which is the final step, **Deploy Application**. In distributed component-based software, this involves choosing in which network computer each component will be deployed, aiming to reduce network traffic, achieve fault tolerance and load balancing, among others issues. Orion offers a novel approach to perform this task, as shown in Figure 14.

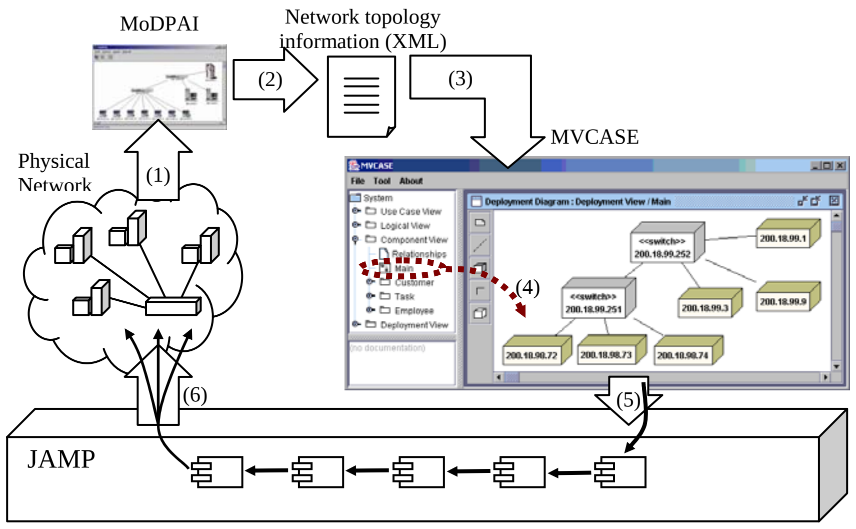


Figure 14: Application deployment with MoDPAI, MVCASE and JAMP

First, the Software Engineer retrieves the network topology, using MoDPAI tool (1). Next, a XML document containing the network topology description is generated (2). This document is then imported in MVCASE (3), becoming available in the form of the UML's deployment diagram. Next, the Software Engineer can analyze the best possibilities and, in a "drag and drop" process (4), configure in which computer each component will be deployed. Once the deployment configuration is defined, the Software Engineer can use the JAMP services (5) to remotely register each component in its corresponding computer (6).

The use of Orion in the deployment process gives the Software Engineer the ability to automatically configure the execution environment. He can change between different deployment configurations, and decide which one results in better performance, or reduced network traffic. Fault tolerance and Load Balancing can also be configured in the environment, by "dragging" the same component into more than one computer. JAMP's ability to register multiple copies of a same server can be used to implement these issues.

5 ORION EVALUATION

In order to evaluate Orion, a preliminary study was performed, with an Accountancy and Invoice domain. This section briefly describes the evaluation.

Methodology

The Accountancy and Invoice domain was developed, using Java as implementation language, HTML and Servlets for user interfaces, MySQL for persistent storage and JAMP as the distribution platform. The study was performed by two undergraduate students (S1 and S2) with two years of experience on these technologies.

First, based on the problem domain requirements, S1 and S2 used Orion to develop the components, during first stage of IPM. Table 1 summarizes this stage.

Table 1: Produced Artifacts and tools used in the first stage of IPM.

Step	Tools used	Artifact	Qt.
Define Problem	MVCASE	Mind-Map	1
Define Problem		Collaboration Model	1
Define Problem		Use Cases Model	1
Specify Components	MVCASE	Model of Types	1
Specify Components		Data Dictionary	1
Specify Components		Model Framework	1
Specify Components		Framework Application	1
Specify Components		Interactions Model	36
Design Components	MVCASE	Classes Model	11
Design Components		Interactions Model	36
Design Components		Components Model	11
Implement Components	MVCASE, JADE, Repository	Interfaces	22
Implement Components		Components	33

After the conclusion of the first stage, S1 and S2 began the next phases, completing the steps of the second stage of IPM. Table 2 summarizes this stage.

Table 2: Produced Artifacts and tools used in the second stage of IPM.

Step	Tools used	Artifact	Qt.
Specify Application	Repository, MVCASE, JADE	Use Case Model	8
Design Application		Components Model	11
Implement Application		Application	51
Deploy Application	MVCASE, MoDPAI, JAMP	Files	87

The construction of the domain and their applications resulted in a total of 33 components, 51 applications and 10263 lines of code developed, as measured by the Unix wc (word count) program, not counting blank lines, comments and libraries. Total time was 82 hours and 5 minutes.

Discussion

After this preliminary study, some key points could be identified. When comparing to an ad-hoc approach, with isolated tools, the following benefits are achieved with Orion:

Reuse: The distributed repository, with mechanisms to store, search and recover software artifacts, facilitates the reuse during the development process.

Maintainability: The ability to treat software artifacts in different abstraction levels facilitates changes to be performed. The support for code generation, components packaging and publishing helps in evaluating the impact of these changes.

Development time: The implementation and deployment tasks, which are partially automated in Orion, are executed in less time than if done manually. Other tasks, such as modeling, are facilitated by the graphical features of the environment, which also contributes to decrease development time.

A main disadvantage was observed. In order to fully take advantage of Orion, the Software Engineer is restricted to some technologies, such as UML and Java. Other technologies can be used, but without the complete support of the environment.

Summary

As mentioned before, although not originally idealized as such, Orion environment implements most of the identified requirements for Component-Based Software Engineering Environments:

Tool integration: Different types of integration between the tools are achieved. The use of JAMP platform as a common middleware layer between the tools, allows them to benefit from the same platform services. The common repository, which is capable of storing and searching XMI documents allows the tools to share their information. A tool can control the operation of other tools, as in the deployment activity, where MVCASE uses MoDPAI tool to recover network information. And a well-defined software process model guides the environment usage.

Support for Component-Based Software Engineering activities: The basic CBSE activities, development “for reuse” and development “with reuse”, are both supported by Orion. Components are constructed, stored in the repository, and then reused.

Reusability: The use of a repository, with services for storing and searching software artifacts, provides different levels of reuse, including design patterns, frameworks, components design and executable code.

Referential Integrity: The version control and search mechanisms of the repository help in avoiding wrong references, since the Software Engineer can search the repository for the referenced components. However, there are no mechanisms implemented yet to assure the integrity of the references, leaving this task for the Software Engineer.

Software Configuration Management: The version control mechanism of the repository helps to guarantee a minimal control of the changes. However, a more complete treatment for Software Configuration Management is required.

Multiple Views of Information: The use of the XMI standard allows the information to be exchanged between different tools. Each tool offers a different view of the information.

Security: The information produced is stored in the repository, which has a security mechanism, that assures minimal user access control.

Technology and Language Independence: Some tools of the environment are dependent on technology, such as JADE and JAMP, which are Java-based, and MVCASE, which uses UML as a modeling language. However, Orion supports the construction of distributed component-based software in any language or technology, demanding only more manual effort.

6 RELATED WORK

Harrison et al [Harrison 2000] offer a historical view of software engineering environments, covering the evolution of CASE tools from the first Programming Support Environments to complex Software Engineering Environments. Some future expectations are presented, describing current and future tendencies of CASE tools. Issues presented in this work, such as tool integration and multiple views of information are achieved by Orion.

Lüer et al. [Lüer 2001] proposes a Component-Based Development Environment (CBDE). They identified seven requirements for CBDEs adopted by industrial component models. However, these requirements do not cover tool integration issues. Since an environment is composed by two or more tools, this integration must be achieved in order to provide a quality support for the development process. They also utilize the concept of *requires* and *provides* ports. A *requires* port is a description of what is needed for some component to run. A *provides* port is a description of something that the component performs. A *requires* port of one component must be connected to a *provides* port of some other component. These concepts can be used to show an architectural view of the applications composed by components. They can also be used as a search criteria for finding components in a repository.

In his work, [Silveira 2002] introduces a new concept, the Spontaneous Software, which offers the Software Engineer the possibility to create software by dynamically installing the components as they are needed. The components are searched in repositories distributed over open networks. A framework was constructed in order to implement these ideas. Several advantages are achieved, such as reusability, evolvability, versioning, among many others. However, only executable code is treated. Orion assumes that these issues must be treated in different abstraction levels, not only code.

Ye & Fischer [Ye 2002] utilize the active repository concept, in order to achieve greater reuse. He states that reuse cannot be fully performed if the Software Engineer does not have a good knowledge of the component libraries, which is often true, since the libraries are usually extremely large. He proposes a process called information delivery, which consists in anticipating the Software Engineer's needs for components. This helps the Software Engineer in gaining more knowledge about the existing components, increasing the reuse. The *information delivery* process is performed by monitoring the activities of the Software Engineer, such as codification and code documentation, and automatically searching for the components. The search criteria is identified inside the code. For instance, a Javadoc comment, or a method call, can be used to formulate a search criteria. In order to increase the search result quality, a combination between searching and browsing is used, as in Orion. However, as in [Silveira 2002], only executable code is considered as software components.

One of the major differences between Orion and other works is that Orion provides practical tools and solutions to perform the CBSE activities. IPM gives the Software Engineer precise, comprehensive ways of using the different tools of the environment, benefiting from their features to develop quality component-based software. Another difference is that Orion works with artifacts in different abstraction levels, not only executable code. The features to recover network information, automatically deploying the components into the computers of the network, are also a differential between Orion and most of the CBSE environments found in the literature.

7 CONCLUDING REMARKS

This paper presented a Component-Based Software Engineering Environment. By integrating tools that were not originally designed to work together, with a software process model and a middleware platform, Orion implements most of the identified requirements, covering a substantial part of the component-based development process. Different technologies, such as middleware, XMI, frameworks, design patterns and CBD principles, were used to construct tools with features that offer an efficient, comprehensive way to develop component-based software. Together with the ability to work over a network, integrating several Software Engineers in the same project, these features make Orion an excellent choice for CBSE support.

Future works shall deal with the extension of Orion to cover a wider range of the development process, such as requirements engineering and testing. Other works shall deal with active repositories mechanisms, to improve reusability. Currently, adjustments and refinements are being performed. A download version is also being prepared, which will be soon available in our web site (http://www.recope.dc.ufscar.br). More details about this work may be found in the references.

ACKNOWLEDGEMENTS

The authors would like to thank to CAPES, CT-Info/CNPq, FAPESB and FAPESP, the brazilian institutions which help to support this work.

REFERENCES

[Almeida02a] Almeida, E, S.; Bianchini, C, P.; Prado, A, F.; Trevelin, L, C. "Distributed Component-Based Software Development Strategy". In: *The 12Th PhDOOS. In Conjunction With the 16Th ECOOP, Málaga - Espanha*. Lecture Notes in Computer Science (LNCS) Springer-Verlag, 2002.

[Almeida02b] Almeida, E, S.; Bianchini, C, P.; Prado, A, F.; Trevelin, L, C. "MVCase: An Integrating Technologies Tool for Distributed Component-Based Software Development". In: *The 6Th Asia - Pacific Network Operations and Management Symposium*, Proceedings of IEEE, Poster Session, Jeju Island – Korea, 2002.

[Almeida03a] Almeida, E, S.; Bianchini, C, P.; Prado, A, F.; Trevelin, L, C. "IPM: An Incremental Process Model for Distributed Component-Based Software Development". In: *The 5th International Conference On Enterprise Information Systems (ICEIS), Angers - França*. ACM Press, 2003.

[Almeida03b] Almeida, E.S. *An Approach for the Distributed Component-Based Software Development* (in portuguese), MSc. Dissertation, Federal University of São Carlos – Brazil, 2003.

[Alves01] Alves, V., Borba, P., 2001. "Distributed Adapters Pattern (DAP): A Design Pattern for Object-Oriented Distributed Applications". In *The First Latin American Conference on Pattern Languages of Programming* – Brazil, 2001.

[Bianchini02] Bianchini, C.P. *Devices Monitoring Tool using Pervasive Computing and Software Agents* (in portuguese), MSc. Dissertation - Federal University of São Carlos – Brazil, 2002.

[Boertin01] Boertin, N., Steen, M., Jonkers., H, "Evaluation of Component-Based Development Methods". In *EMMSAD'2001*, Sixth CAiSE/IFIP8.1, 2001.

[Borland03a] Borland Software Corporation. "JBuilder". Available at URL: http://www.borland.com/jbuilder/index.html - Consulted in April, 2003.

[Borland03b] Borland Software Corporation. "Delphi Studio". Available at URL: http://www.borland.com/delphi/index.html - Consulted in April, 2003.

[Buschmann96] Buschmann, F., et al, *Pattern Oriented Software Architecture: A System of Patterns*. John Wiley & Sons, 1996.

[D'Souza98] D'Souza, D. F. and Wills, A. C. *Objects, Components and Frameworks with UML: The Catalysis Approach*. Addison Wesley. 1998.

[Franklin96] Franklin, S., Graesser, A. "Is it an Agent or just a Program? : A Taxonomy for Autonomous Agents". *Proceedings of the Third International Workshop on Agent Theories, Architectures, And Languages*. Springer-Verlag, 1996.

[Gamma95] Gamma, E., Helm, R., Johnson, R., Vlissides, J.M., *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[Guimarães00] Guimarães, M.P. *Implementation Design for Cooperative Work Support in JAMP Platform* (in portuguese), MSc. Dissertation - Federal University of São Carlos, 2000.

[Hansmann01] Hansmann, U.; … [et al]. *Pervasive Computing Handbook*. Springer-Verlag, 2001. 409p.

[Harrison00] Harrison, W., Ossher, H., and Tarr, P. "Software Engineering Tools and Environments: A Roadmap". In *The Future of Software Engineering*. ACM, New York, 2000, 261-277.

[Heineman01] Heineman, G., T., Councill, W., T. *Component-Based Software Engineering, Putting the Pieces Together*, Addison-Wesley, 2001.

[Jacobson97] Jacobson, I., Griss, M., Jonsson, P. *Software Reuse: Architecture, Process and Organization for Business Sucess*, Addison-Wesley. Longman, 1997.

[Jacobson01] Jacobson, I., et al., 2001. *The Unified Software Development Process, 4th edition*. Addison-Wesley, 2001.

[Lucena02] Lucena, C.; … [et al]. "Software Engineering for Large-Scale Multi-Agent Systems". *SELMAS'2002. Proceedings of 24th International Conference on Software Engineering*. P 653-654. Orlando, Florida, USA, 2002.

[Lüer01] Lüer, C. and Rosenblum, D. S. "WREN: An Environment for Component-Based Development". *ACM SIGSOFT Software Engineering Notes*. Volume 26. Number 5. September 2001. pp. 207-217.

[Lumina00] Lumina Corporate Solution; Moura, L.M. "KB Implementation. Internal Technical Report" (in portuguese). Lumina Corporate Solution, 2000. 88p.

[Microsoft96] Microsoft Corporation. "DCOM Technical Overview". November 1996. http://msdn.microsoft.com/library/backgrnd/html/msdn_dcomtec.htm

[OMG02a] Object Management Group. *XML Metadata Interchange (XMI) – Version 1.2*, January, 2002.

[OMG02b] Object Management Group. *The Common Object Request Broker Architecture: Core Specification, Version 3.0.2*. December, 2002.

[Perspective00] Perspective Select Perspective. *Princeton Softech's practical methodology for delivering next generation applications, The Active Archive Solutions Company*, 2000. http://www.princetonsoftech.com. Consulted June, 2002

[Pressman01] Pressman, R. S. *Software Engineering: A Practitioner's Approach*, McGraw-Hill, 2001.

[Ross97] Ross., D., T. "Structured Analysis (SA): A language for communicating Ideas". IEEE Transaction on Software Engineering, 1997.

[Rumbaugh99] Rumbaugh, J., Jacobson, I., Booch, G. *The Unified Modeling Language Reference Manual*, Addison-Wesley, 1999.

[Silveira02] Silveira, G.E. *SOS - A Framework for Distribution, Management and Evolution of Component-Based Software Systems over Open Networks*. PhD thesis, Informatics Department, Federal University of Pernambuco - Brazil, 2002.

[Sommerville00] Sommerville, I. *Software Engineering (6th Edition)*. Pearson Education, August 2000.

[Souza01] Souza, L.F.H., *Middleware Service Models Study and JAMP Platform Extensions Proposal* (in portuguese), MSc. Dissertation - Federal Universisty of São Carlos – Brazil, 2001.

[Stallings99] Stallings, W. *SNMP, SNMPv2, SNMPv3, and RMON 1,2 and 3. ed.* Addison-Wesley, 1999. 619p.

[Stojanovic01] Stojanovic, Z., Dahanayake, A., Sol., H. "A Methodology Framework for Component-Based System Development Support". In *EMMSAD'2001, Sixth CAiSE/IFIP8.1*, 2001.

[Sun97] Sun Microsystems. *JavaBeans API Specification*. Version 1.01. July 1997.

[Sun02] Sun Microsystems. *Enterprise Java Beans Specification V. 2.1*. August, 2002.

[Sun03a] Sun Microsystems. *Java RMI: Remote Method Invocation Specification*. URL: http://java.sun.com/j2se/1.4.1/docs/guide/rmi/spec/rmiTOC.html. Consulted 04/2003.

[Sun03b] Sun Microsystems. *Java Servlet Technology*. URL: http://java.sun.com/products/servlet/ - Consulted in February, 2003.

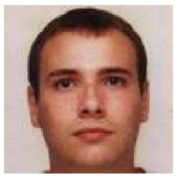
[Szyperski98] Szyperski, C. *Component Software – Beyond Object-Oriented Programming*. Addison-Wesley and ACM Press, 1998.

[Tanenbaum96] Tanenbaum, A.S. *Computer Networks. 3 ed*. Prentice Hall, 1996. 848p.

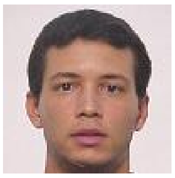
[Ye02] Ye, Y, Fischer, G.; "Supporting Reuse by Delivering Task-Relevant and Personalized Information", In: *24th International Conference on Software Engineering*. Orlando, USA, 2002.

[Yoder98] Yoder, J., Johnson, R., E., Wilson, Q., D. "Connecting Business Objects to Relational Databases". In: *PLoP'1998, Pattern Language of Progamming*, 1998.

About the authors



Daniel Lucrédio is a computer engineer in the Federal University of São Carlos, Brazil, and M.Sc. candidate in Computer Science in the Federal University of São Carlos. Researcher in the Software Engineering Environments are, is the author of MVCASE, an UML-based modeling tool used in several brazilian institutions. E-mail: lucredio@dc.ufscar.br.



Eduardo Santana de Almeida graduated in Computer science in Universidade Salvador (UNIFACS) and obtained his M.Sc. degree in the Federal University of São Carlos, Brazil. PhD candidate in the Federal University of Pernambuco, Brazil. Author of several papers on Component-Based Software Engineering.

Calebe de Paula Bianchini is a B.Sc. from University of São Carlos (UFSCar/Brazil), where he has also obtained his M.Sc. degree. Currently, he is working towards his Ph.D. in Electronic Engineering at the Polytechnic School-University of São Paulo (POLI-USP), where he works with Parallel and Distributed Systems and High Performance Computing.

Antonio Francisco do Prado graduated in Fortification and Construction Engineering in the Military Institute of Engineering (IME), where he also obtained his M.Sc. degree in informatics. PhD from PUC-Rio, Brazil. He currently conducts several researches in Component-Based Software Engineering area, and the development of tools and processes for CBSE.

Luis Carlos Trevelin graduated in computer science in São Carlos Institute of Mathematical Sciences – University of São Paulo, where he also obtained his M.Sc. degree. PhD in PUC-Rio, Brazil, and post-PhD through University of Kent at Canterbury, England. He conducts several researches on distributed systems, including the development of a CORBA middleware platform.