A Pattern Language for the Perception, Design and Implementation of Distributed Application Partitioning

António Rito Silva[1], Fiona Hayes[2], Francisco Mota[1], Nino Torres[1] and Pedro Santos[1]
[1]INESC/IST Technical University of Lisbon, Rua Alves Redol n°9, 1000 Lisboa, PORTUGAL
[2]IONA Technologies Ltd., 8-10 Lower Pembroke Street, Dublin 2, IRELAND
Rito.Silva@inesc.pt, fhayes@iona.com

A previous version of this paper was presented at OOPSLA'96 Workshop on Methodologies for Distributed Objects, October 1996, San Jose, California, USA.

Abstract

This paper presents a pattern language for the perception, design and implementation of distributed application partitioning. The pattern language describes a stepwise development which offers the following qualities: expressiveness and support of a wide range of user models; smooth transformation of an object-oriented model onto a client-server model; selective transparency; customization; incremental development, test and simulation. It considers partitioning issues at both the problem and the solution spaces, and identifies several distribution concerns associated with partitioning at the solution space. Each distribution concern is associated with solutions described as design patterns. The pattern language is RM-ODP compliant.

Keywords: Pattern Language, Object-Oriented Analysis and Design Methods, Development of Distributed Applications, Design Patterns, Frameworks.

1 Introduction

Distributed object technologies, e.g. CORBA [Siegel 96], promise the support of the object-oriented paradigm on the development of distributed applications and suggest their use from the very beginning of the development process. However, the technology bias the problem perception and solution construction because of two main reasons: expressiveness and performance constraints.

- **Expressiveness bias**. The real world to be modeled is too complex and diverse to be captured using the paradigms supported by technology. As a consequence, the problem description will be biased by solution issues due to use of modeling concepts unfamiliar to the problem space.

- **Performance constraints bias**. Distributed applications have performance requirements due to communication limitations which forbids the developer of applying a pure object-oriented design. As a consequence, developers design distributed applications using the client/server paradigm which handles with large grained objects.

The use of distributed object technology too early also has disadvantages from an object-oriented process perspective.

- **Incremental simulation, test and debug**. The early use of a distributed object technology implies that test and debug of an application is done in a concurrent and distributed environment. However, application functionalities implementation is usually independent of distribution issues and could be tested and debugged in a amicable centralized and non-concurrent environment.

Computer supported cooperative work (CSCW) researchers [Rodden 91] identified some drawbacks in distributed technology:

- **Absence of detail**. Transparency does not allow distribution issues to be handled at different levels of abstraction and according to particular application needs.

- **Lack of support for non-traditional models**. Only traditional models are supported, as serialization and ACID properties of transactions. However CSCW applications require new models of interaction and cooperation.

So, an approach to the development of distributed applications which copes with these problems is required.

This paper defines a pattern language for the perception, design and implementation of distributed application partitioning. The pattern language defines a stepwise approach to development which delays decisions on distributed technology and client/server architecture as well as provides stages for definition of models of interaction and cooperation, allows incremental simulation, test and debug, and provides several levels of detail.

The pattern language is presented for the distributed application partitioning problem and it is applied in the development of a distributed agenda.

The rest of this paper is structured as follows. Next section describes the DASCo approach. The Distributed Agenda functionalities are described in section 3. Section 4 describes the goals and context of the pattern language as well as its requisites. Section 5 presents the pattern language for partitioning. The pattern language forces and qualities are discussed in section 6 while section 7 presents related work. Finally, section 8 presents conclusions.

2 DASCo Approach

The pattern language was defined in the context of an approach to the development for distributed applications with separation of concerns (DASCo) initially described in [Silva 95].

DASCo approach distinguishes the problem space from the solution space in the lines of [Hoydalsvik 93, Cook 94, Jackson 95]. This distinction is also extremely useful for distributed applications because the perception of the distributed application does not directly map to entities and constructs at the solution space. For instance, to be closer to the user's language a declarative language may be used in the problem space description.

Three levels of abstraction are considered by DASCo: model, policy and mechanism. Models describe users' expectations about the application or system behavior, e.g. the users expects his/her requests to be executed in isolation. Policies define algorithms to support application models, e.g. a pessimistic policy of concurrency control provides isolation of invocations to a resource. Mechanisms define functionality which is used by policies to implement their algorithms, e.g. a mutex is used to delay invocations to the resource. Mechanisms are further separated into abstract and concrete mechanisms. Abstract mechanisms fulfill the needs of models and policies and are platform independent, e.g. a C++ class defining the mutex interface. Concrete mechanisms are the mechanisms supported by the computational infrastructure and are used to implement abstract mechanisms, e.g. operation system mutex primitives used in the implementation of the C++ mutex class.

Models specification is done at the problem space while policies and mechanisms are part of the solution space.

DASCo approach considers separation of concerns at the both problem and solution spaces. Separation of concerns, considered a key technique of software engineering [Jackson 95], allows a "divide and conquer" strategy to development. Naturally, different kinds of concerns are considered for problem and solution spaces.

At the problem space DASCo considers the following three concerns: partitioning, concurrency and failure:

- **Partitioning**. Partitioning is the problem of application functionality and structure separation across the network.
- **Concurrency**. Concurrency problem deals with the interaction and cooperation between different activities of the distributed application.
- **Failure**. The distributed application robustness is dealt by the failure concern.

The problem concerns have their own models and specification languages. For instance, sequential centralized systems offer a set of "traditional" models, called strict models, where activities and data are isolated. In these models users expect system reaction considering only his/her own inputs, from the user's point of view the model is the transparency of partitioning, concurrency and failures. However, developers are requesting particular non-strict models for cooperative work [Rodden 91] where the user is aware, according to some criteria, of partitioning, concurrency and failure.

The DASCo approach also identifies in the solution space a set of distribution concerns. The solution concerns considered in this paper and associated with the partitioning problem are: replication, naming, configuration and communication:

- **Replication**. The replication concern defines replica objects that are dispersed throughout the distributed nodes and together behave consistently as a single object. The replication concern preserves some criteria of object consistency.
- **Naming**. The naming concern defines the name spaces needed by the distributed application such that names may coexist consistently. A name space establishes identification, which allows a distributed object to be distinguished from all others, and denotation, which is a mean to refer to distributed objects.
- **Configuration**. The configuration concern defines how distributed nodes and their objects react upon creation and destruction of other distributed nodes. Configuration concern also defines reconfiguration (dynamic configuration) procedures.
- **Communication**. The communication concern defines the distributed communication paradigms distributed objects use to communicate.

Solution concerns may have associated several policies. For instance: a primary-copy policy for the replication concern defines a central replica which synchronizes replica's changes; a unique and universal identifier policy for the naming concern defines a name space where objects have names which are unique identifiers and denote objects in all distributed nodes; a centralized policy for the configuration concern defines a central node to where other nodes connect when they are created; a pull-push policy for the communication concern defines an active communication object which reads data from a source and writes it into a sink.

Associated with solution concerns there are entities as object-oriented frameworks, object-oriented design patterns and distribution mechanisms. These entities provide concerns solutions and may be customizable to application-specific needs. Object-oriented design patterns describe a set of cooperating objects and classes which provide a solution in a context [Gamma 95]. An object-oriented framework is an aggregation of collaborating classes with a common purpose constituting a skeleton which should be fleshed out to build complete applications [WB90]. Pattern implementations are usually part of frameworks [Johnson 92].

Figure 1 shows the overall DASCo architecture. The distribution requirements specification define the user models at the problem space while patterns and frameworks, describe and implement, policies at the solution space. Classes and wrappers support abstract mechanisms while concrete mechanisms are provided by the platforms.

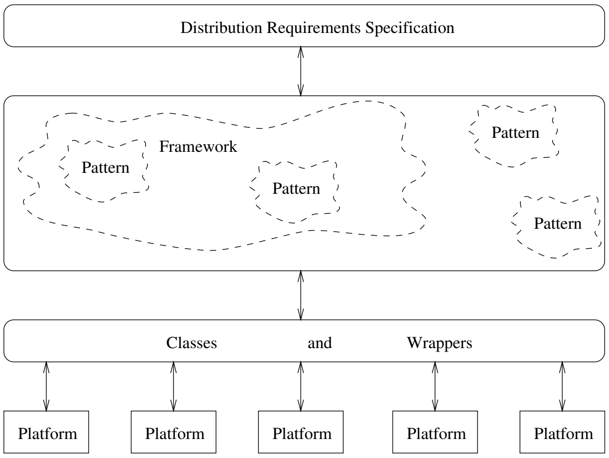

Figure 1: DASCo Architecture.

The architecture has variants depending on the concrete mechanisms and the user models they support. The use of powerful mechanisms supporting particular user models may reduce the architecture to the top and bottom layers where user models are directly supported by concrete mechanisms. For instance, platforms supporting the ACID (Atomicity, Consistency, Isolation and Durability) properties of transactions offer a strict model. However, as pointed out in the previous section, this approach has several drawbacks.

This paper will deal with the partitioning problem and will present a constructive development process at the solution space.

3 The Distributed Agenda Application

To illustrate the pattern language usage, the distributed agenda application will be used as an example. This section describes the distributed agenda functional and architectural requirements.

3.1 Functional Requirements

The distributed agenda application provides services to:

- Agenda manager create, update and delete of users.
- Users create, update and delete personal (private) appointments.
- Users create, update and delete meetings. A meeting requires its creator and several participants.
- A user consults its agenda to see his appointments and the meetings he participates in.

Several agendas may be simultaneously open but a single instance of agenda manager may be open in each moment.

3.2 Architectural Requirements

Users interact with the agenda application using any machine of a Local Area Network (LAN) or Wide Area Network (WAN).

4 Pattern Language Context and Requisites

The pattern language has a context and some prerequisites: design patterns and integration mechanisms with particular properties.

4.1 Context

The pattern language defines the purpose and activities of distributed application development process for the partitioning problem. The language is constituted by a sequencing of four patterns: distribution requirements, functionalities design, logical distribution design and physical distribution design.

In terms of DASCo levels of abstraction, the distribution requirements pattern describes the user's model while policies and abstract mechanisms are part of the logical distribution pattern, and concrete mechanism are used by the physical distribution pattern. The functionalities pattern is an intermediate stage where distribution issues are minimized.

4.2 Design Patterns

The pattern language uses a set of object-oriented design patterns. Design patterns describe the structure and behavior of a set of collaborating objects. Besides these generic characteristics, the design patterns provide solutions to distribution concerns at the solution space. Each concern may be supported by several policies. A policy defines an algorithm, e.g. to deal with the replication concern there are algorithms like voting or primary copy.

The design patterns used by the pattern language have the following particular characteristics which have impact on the pattern language properties:

- **Encapsulation**. The design pattern abstracts several policies which are encapsulated within the same interface. So, it is possible to replace policies avoiding change propagation.

- **Extensibility**. The design pattern can be customized in terms of the application-specific semantics. The more suitable policy depends on the domain objects semantics.

- **Modularity**. The design pattern associated with a concern is decoupled from the design patterns associated with other concerns. This orthogonality enhances separation of concerns and reuse of design patterns in several contexts.

- **Composition**. Design patterns should be composed in frameworks which constitute skeletons for the development of distributed applications. Composition requires the consistent integration of policies for the different concerns. Not all the policies combinations are compatible and so, the framework should forbid inconsistent combinations.

For instance in [Sousa 96] a design pattern is described which identifies several policies of naming in distributed systems, e.g. unique universal identifiers, absolute names, and so on. These policies are abstracted by the design pattern.

4.3 Integration

The pattern language requires an integration mechanism that allows incremental construction of specifications. This can be achieved using design solutions, languages or operating system mechanisms.

Design solutions for integration, like the Proxy pattern [Gamma 95] or the Surrogate pattern [Martin 95], allow the incremental definition of objects using an abstract interface which isolates clients from the implementation allowing, for instance, the late association of a replica manager to an object.

Reflection languages [Chiba 93] and operating systems [Yokote 92] provide meta-objects which define the semantics of objects. Incremental definition of objects is achieved by changing object's meta-objects.

The quality of such mechanisms determines the amount of code that have to be changed because of integration and whether it is or not possible to have run-time code replacement[1]. This is not a issue of the pattern language and in this work the design based solution, Proxy/Surrogate, is used.

[1] However to support run-time replacement the policies associated with each of the concerns should define the quiescent states where consistent replacement of policies can occur.

5 Pattern Language for Partitioning

In this section the pattern language is described. The pattern language is constituted by patterns which focus on the partitioning problem at different levels of abstraction (Figure 2).

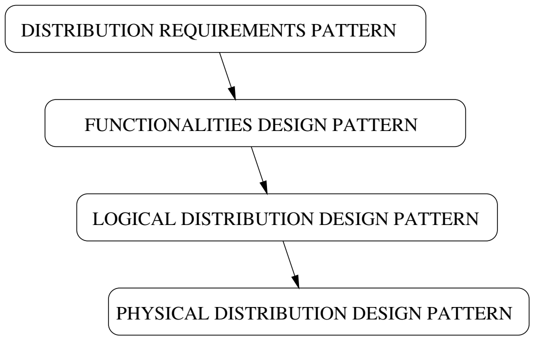

Figure 2: The Pattern Language.

Each pattern's description is divided into Context, Problem, Forces, Strategy, Result and Example.

Context defines the frame and pre-requisites of the pattern appliance. Problem formulates the question to be solved by the pattern and the Forces discuss the trade-offs and compromises that should be considered by solutions. Strategy sketches a solution procedure and Result describes the consequences on the context after applying the strategy. Finally, Example illustrates the pattern appliance to the distributed agenda application.

5.1 Distribution Requirements

5.1.1 Context

A distributed application should be distributed across the network, (partitioning). Application partitioning is twofold: functionality and structure. Functionality partitioning defines aggregates of application functionalities which are provided independently. Structure partitioning defines aggregates which provide non-functional support, e.g. persistency will be provided by a particular node.

Any development process needs a step where the problem is described. The problem space is different from the solution space and so, different specifications may be constructed [Hoydalsvik 93, Cook 94, Jackson 95] to describe them. The solution space may not necessarily result from an elaboration of the problem space. Entities of the problem space may not have direct correspondence with entities of the solution space.

5.1.2 Problem

How are distribution requirements of partitioning described in terms of the problem space?

5.1.3 Forces

1. **Transparency vs Non-Transparency**

The problem describes application functionalities. Often, problem description ignores partitioning and its consequences, e.g. replication. Partitioning consequences are considered a design issue and so they are transparent in the problem space.

However, users have expectations about the behavior of the distributed application. Traditionally, users expect strict models, where partitioning is not relevant from the user point of view. Despite that, new applications, as CSCW applications, require new models where partitioning and its consequences are a perceptible part of the application requirements, e.g. users expect an object to be temporarily out of date according to a particular consistency criteria.

2. **Procedural vs Declarative Languages**

   Frequently, partitioning requirements are described using a procedural language, like an object-oriented language. For instance, usually client objects and server interfaces are identified as part of requirements. However, a description of the problem using a procedural language already suggests a solution.

   On the other way, users describe their expectations about the application behavior using a declarative language. Furthermore, their expectations are not necessarily directly mapped to partitioning and the solution space. For instance a performance requirement does not necessarily map to a solution in terms of object replicas. However, some of the requirements may be naturally represented using a procedural language, e.g. the existence of a legacy application may be described by an object.

5.1.4 Strategy

Partitioning requirements should be captured from the users perspective. Whenever partitioning is relevant in the user's model it should be expressed as a distribution requirement, it will not be transparent.

Using, either a procedural or declarative language, partitioning requirements should be expressed in terms of the problem space and avoid suggesting any solution.

5.1.5 Result

The final specification holds two kind of partitioning requirements: explicit and implicit. The former are perceived by the user as associated with partitioning, while the latter have no direct correspondence with partitioning. However, partitioning consequences may be deducible from the latter.

For instance, a bank with several branches has explicit partitioning requirements due to geographic distribution. These explicit distribution requirements may establish functionality partitioning, the set of functionalities is not the same for every branch. Implicit partitioning requirements may define the grade of availability of an account. This implicit requirement may have consequences on account replication.

Explicit requirements may refer to both functionality, e.g. the existence of several applications, and structure partitioning, e.g., the existence of a legacy component which should be isolated.

5.1.6 Example

The distributed agenda is perceived as partitioned because of the existence of several users which share meetings and of the architectural requirements. Moreover, two applications are required, agenda and manager.

An explicit partitioning requirement is the partitioning of the set of functionalities into agenda functionality and manager functionality. This explicit requirement may be described with objects (procedural language), that are responsible for the functionalities.

An implicit requirements defines the grade of availability of user and meeting objects. In this example a strict model is considered, an up-to-date version is permanently available. This implicit requirement may be described with rules with pre- and post- conditions (declarative language) associate with object operations.

5.2 Functionalities Design

5.2.1 Context

To validate application requirements the user should be confronted, as soon as possible, with running parts of the application. Object-orientation is particularly suitable for an incremental development process, both in terms of functionality (vertical) increments and non-functionality (horizontal) increments, e.g. incremental support of partitioning. This development process is characterized by several passages from requirements to implementation where the specification is additionally constructed.

A specification of the problem is already constructed. This specification includes both, implicit and explicit partitioning requirements.

5.2.2 Problem

How are the application partitioning requirements validated without building the complete application and in particular without implementing partitioning?

5.2.3 Forces

1. **Partitioning vs Non-Partitioning**

   Distribution concerns increase the application complexity and consequently the development time. So, it will be faster to construct a non-partitioned version of the application centered on the implementation of its functionalities.

   However, partitioning requirements affect application functionality. User expectations will be frustrated if confronted with such implementation.

This way, the design and its implementation should enforce partitioning requirements. However, some of them may be difficult to support in a context ignoring partitioning. For instance, simultaneous queries to an object returning different values is a requirement difficult to support without specifying replication.

5.2.4 Strategy

A centralized version of the distributed application is constructed to validate application requirements.

Partitioning requirements are implemented while it is not necessary to decide on a particular distribution architecture. For instance, those explicit requirements associated with functionality and structure partitioning are implemented by objects which communicate using procedure calls. Afterwards, those distribution requirements that can not be completely implemented should be transformed into lower level requirements, associated with design entities. For instance, a consistency criteria should be specified on a class which objects return different values to simultaneous queries.

Considering the bank case, the central node, bank branches and account classes are implemented. Account objects are not replicated. Furthermore, if the account objects at the central node provide different services than at the branches two different interfaces should be defined for account objects. Since account objects are not replicated, any availability rule should be re-written, refined, in the context of central node, bank branch and account design classes.

5.2.5 Result

Due to the centralized implementation, the user will observe a semantically weaker application, according to requirements, which has the main functionalities but may not fulfill some of the requirements. However, note that these requirements are those which depend on the distribution architecture. A main characteristic of a distribution architecture is message asynchrony which is responsible for a non-deterministic order of messages and consequently non-deterministic application behavior. The non-distributed application will be more deterministic than expected by the user.

For those explicit requirements which are associated with a distributed architecture, a non-distributed implementation should be constructed. For instance, if a legacy component exists an object simulating the component should be created encapsulating it.

Application functionalities can be tested and debugged in a friendly non-distributed and non-concurrent environment. Distribution concerns at the solution space are still transparent to the programmer.

The application can be assessed and metrics will support future decisions on which distribution architecture to choose. The application is now ready for horizontal increments.

5.2.6 Example

Users, appointments and meetings are specified as classes. The Booch [Booch 94] class diagram in Figure 3 shows these classes. Class `USERS` has a single instance which holds a list of `USER` objects. Each `AGENDA` object contains a list of `SCHEDULE` objects.

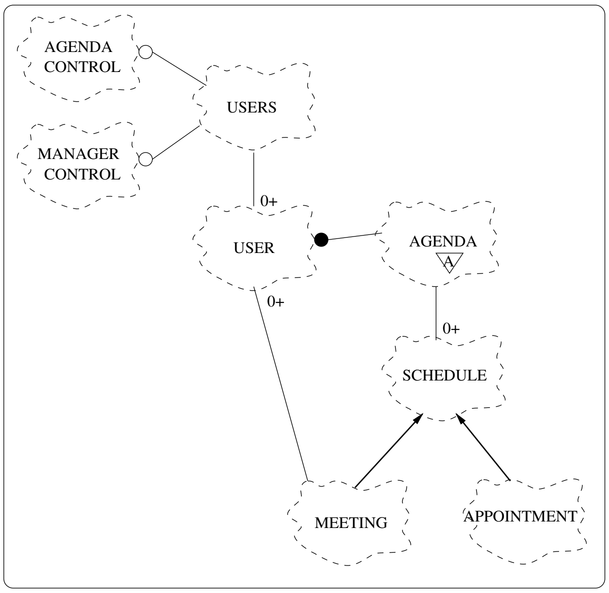

Figure 3: Agenda Functionalities Design.

The explicit requirement regarding partitioning of agenda and manager functionality results in the definition of classes `AGENDA CONTROL` and `MANAGER CONTROL`.

The implicit requirement has no implications on design since it establishes a strict model for `MEETING` and `USER` objects availability. The semantics of availability is automatically supported by the centralized implementation of `MEETING` and `USER`.

5.3 Logical Distribution Design

5.3.1 Context

A distributed computing system is defined in [Bal 89] as a system of multiple autonomous (logical) processors that cooperate only by sending messages over a communication channel. Note that this definition does not distinguish between physically separated components and logically autonomous modules communicating via messages.

There are several possibilities of distribution architectures. For instance, either the applications are clients of a server which stores shared data, or shared data is split between the applications using a replication policy.

The design and implementation of a centralized version of the application, supporting application functionalities, is already developed.

5.3.2 Problem

How is distribution architecture incorporated into the non-partitioned application in a stepwise manner?

5.3.3 Forces

1. **Transparency vs Non-Transparency**

   Solutions of distribution concerns can be transparently incorporated in the application using powerful communication mechanisms or abstract machines. However, this forbids the evaluation of different distribution architectures.

   Which distribution concerns at the solution space should be considered? If non-transparency is considered all concerns should be considered. However, it is possible that some of these concerns will not be necessary, e.g. if there are no requirements on availability then replication should be ignored.

2. **Fixed vs Customizable Solutions**

   Each of the concerns to be considered may have several possible solutions and it may be premature to decide on which solution to apply. So, solutions to distribution concerns to be incorporated in the application should be customizable.

   However the use of fixed and pre-defined solutions allow an higher level of transparency. For instance, transactional systems offer ACID properties and the programmer does not need to know the transactional system internal architecture.

5.3.4 Strategy

The non-partitioned design is enriched with logical distributed nodes that communicate only sending messages. This is a step further towards a distributed application without compromising on particular physical distribution mechanisms. It allows smooth evaluation of different distribution architectures.

The distribution concerns, at the solution space, to be considered are determined by logical distribution and partitioning requirements. The latter are the requirements defined at the functionalities design that result from the refinement of problem space requirements. Due to logical distribution, it is necessary to consider the communication, naming and configuration concerns. Due to distribution requirements it may be necessary to consider replication.

Configuration should handle the management of distributed nodes. Naming should define each node name space. Communication should define how objects belonging to different nodes communicate. Replication deals with the consistency of replicated objects. Note that logical distribution by itself does not require replication but replication may result either from an availability requirement or from a data consistency criteria requirement.

Since there are several possible solutions for each of the concerns, default solutions are used to achieve a rapid development and tuned afterwards.

The following design patterns are used:

- A design pattern for configuration that supports several policies of configuration, e.g. a single logical node contains all the shared objects.
- A design pattern for naming that supports several policies of naming, e.g. when a reference is sent to another node it is transformed into a Unique and Universal Identifier (UUID) [Sousa 96].
- A design pattern for distributed proxies that manages communication between logical nodes, e.g. it interacts with the naming pattern to convert object references sent between nodes.
- A design pattern for replication that supports several policies of replication, e.g. a primary-copy or a voting policy.

5.3.5 Result

The resulting application will consider a distributed architecture (partitioning) without compromising with particular physical distribution mechanisms. For instance, it is transparent whether the distributed proxies will be implemented using sockets or remote procedure call (RPC).

Transparency is achieved if the concern is not relevant from the requirements, e.g. replication concern, or if a predefined solution is used. In the former the concern is ignored while in the latter the programmer does not need to know the complexities of the solution.

Although solutions for some distribution concerns were already incorporated, the application can be tested and debugged in a friendly non-distributed and single-process environment. Physical distribution is transparent to the programmer.

The application can be assessed and metrics will help on customization of the policies supported by the design patterns associated with concerns.

5.3.6 Example

New classes are incorporated into the application: `NODE`, `PROXY`, `NAME MANAGER` and `CONFIGURATOR` (Figure 4). Class `NODE` represents a logical distributed node which has a `CONFIGURATOR` object and a `NAME MANAGER` object. Distributed objects, which are known in more than one node, have associated a `LOCAL PROXY` and several `REMOTE PROXY` objects. The `PROXY` objects define the node's border.

These classes are customized for the particular application needs. Three different subclasses of node are considered which represent the chosen distributed architecture:

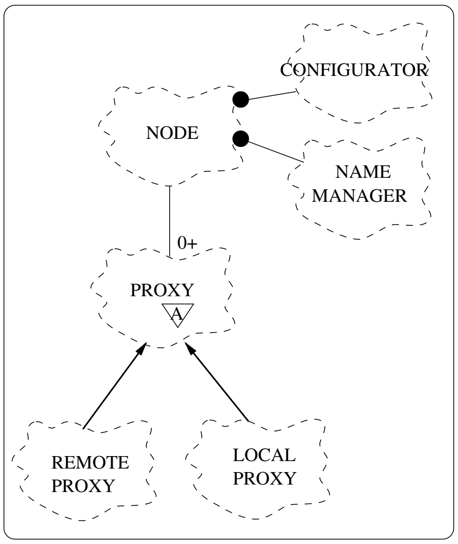

Figure 4: Logical Distribution Manager Classes.

AGENDA NODE, MANAGER NODE and CENTRAL NODE. The chosen distributed architecture is the simpler distributed architecture satisfying the explicit distribution requirements. The partitioning of functionalities between agenda functionality and manager functionality results in the specification of two logical nodes AGENDA NODE and MANAGER NODE. The CENTRAL NODE is the simpler solution to support the sharing of meetings and users between agenda and manager functionalities. The consistency criteria associated with MEETING and USER is supported by the CENTRAL NODE, the replication concern will not be considered.

The interaction diagram in Figure 5 shows a generic configuration procedure. Two phases of interaction are considered. Create managers phase creates the managers needed for logical distribution. Configure phase establishes the distributed architecture. Method configureNode defines the infrastructure communication architecture while method configureApplication defines the application-specific part of configuration.

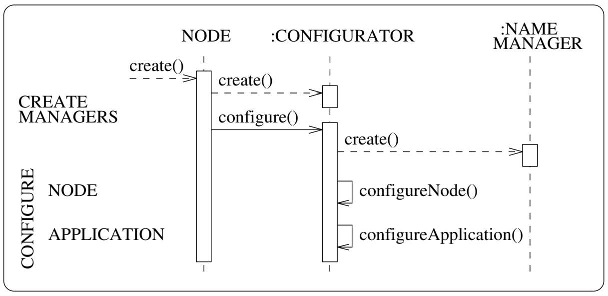

Figure 5: Logical Configuration.

The methods below define the logical nodes configuration policies. In this distributed architecture CENTRAL NODE address needs to be known by every other node: CENTRAL NODE is implemented as a Singleton [Gamma 95], a class which has a single instance.

An agenda node connects to the central node using method connectCentralNode of AGENDA CONFIGURATOR while method configureApplication initializes the agenda application with a remote proxy to USERS. After configuration, agenda application can start interacting with the REMOTE PROXY USERS object, for instance to get the agenda of a particular user.

```
int AgendaConfigurator::configureNode()
{
  // gets central node local proxy
  LPCentralNode* lpCentralNode = connectCentralNode();

  // unable to connect
  if (!lpCentralNode) return FAIL;

  // creates central node remote proxy
  // node_ refers the AGENDA NODE being configured
  RPCentralNode* rpCentralNode =
                 new RPCentralNode(lpCentralNode,node_);

  // associates local and remote proxies
  // in the name manager
  node_->nameManager()->bind(rpCentralNode,lpCentralNode);

  // registers remote proxy in node_
  node_->registerConfiguration(rpCentralNode);

  // returns status
  return OK;
}

int AgendaConfigurator::configureApplication()
{
  // gets a remote proxy of list of users
  RPUsers* rpUsers = node_->rpCentralNode()->getUsers();

  // registers users remote proxy in agenda control object
  agendaControl_->registerUsers(rpUsers);

  // return status
  return OK;
}
```

Method connectCentralNode connects the agenda node to the central node. CENTRAL NODE logical address is hard-coded in singleton pattern.

```
LPCentralNode* AgendaConfigurator::connectCentralNode()
{
  // Uses logical address of central node
  // CentralNode is a Singleton
  return CentralNode::instance()->localProxy();
}
```

The configuration procedure of MANAGER NODE is similar with configuration of AGENDA NODE, both connect to CENTRAL NODE and require a remote proxy of USERS to start working. However, a requirement states that a single manager application should be active each time. Thus configureApplication should have a different definition.

```
int ManagerConfigurator::configureApplication()
{
  if (!node_->rpCentralNode()->manager()) {
    // if a single manager is active
    // gets a remote proxy of list of users
    RPUsers* rpUsers = node_->rpCentralNode()->getUsers();
```

```
    // registers users remote proxy manager control object
    managerControl_->registerUsers(rpUsers);

    // return success
    return OK;
  }
  else
    // returns failure
    // a manager is already running
    return FAIL;
}
```

CENTRAL NODE does not connect other nodes, all calls are synchronous and nodes receive results when invocation returns, so method configureNode is empty. Method configureApplication creates a new USERS object and fills it with information in a persistent store, method fillFromBD.

```
int CentralConfigurator::configureApplication()
{
  // create users
  Users* users = new Users();

  // get info from data base
  users->fillFromDB();

  // return status
  return OK;
}
```

The object diagram in Figure 6 shows the state resulting from the configuration of an AGENDA NODE and the CENTRAL NODE. Objects of classes REMOTE PROXY CENTRAL NODE and LOCAL PROXY CENTRAL NODE were created by the application-independent configuration procedure, while objects of classes REMOTE PROXY USERS and LOCAL PROXY USERS were created by the application-specific configuration procedure. After configuration object :REMOTE PROXY USERS is available at the agenda node and execution can proceed as specified in the functionalities design. This distributed architecture was chosen to centralize data management and simplify sharing but other distributed architectures may be tried by redefinition of configuration methods.

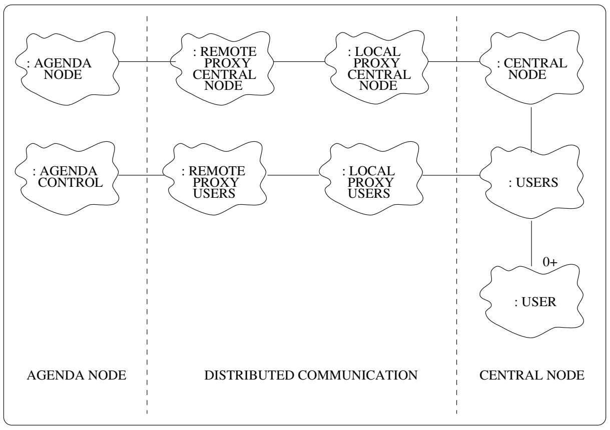

Figure 6: CENTRAL and AGENDA NODES After Configuration.

The interaction diagram in Figure 7 shows a generic logical distributed invocation. A logical distributed invocation has three phases in both kinds of proxies: remote and local. In the REMOTE PROXY it is necessary to convert the references which are being sent into names understandable by the LOCAL PROXY, and the names which are received into references understandable by the REMOTE PROXY. An inverse order procedure occurs at the local proxy. REMOTE PROXY invokes method m to LOCAL PROXY while LOCAL PROXY invokes method m to OBJECT. In both, REMOTE PROXY and LOCAL PROXY, NAME MANAGER objects are used to do conversion and to generate distributed names. NAME MANAGER objects support distributed naming policies.

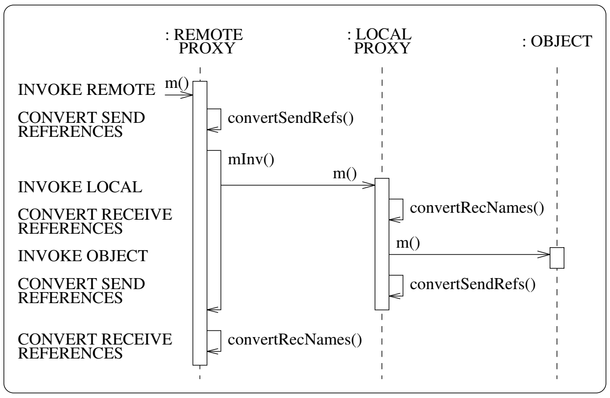

Figure 7: Logical Distributed Invocation.

The code below shows the logical distributed invocation associated with method getUser of class REMOTE PROXY USERS which, given the user's name, returns a REMOTE PROXY USER object. Methods convertSendRefs of REMOTE PROXY USERS and convertRecNames of LOCAL PROXY USERS do not need to be defined because the only entity sent, string name, is not an object, however, lpUser_ object needs to be converted.

The REMOTE PROXY USERS object invokes method getUser to LOCAL PROXY USERS and receives a local user proxy object (lpUser_). Afterwards, it is verified whether a remote proxy for lpUser_ exists already. If it is not the case, a new remote proxy is created and associated with the local proxy.

```
void RPUsers::getUserInv(const String* name)
{
  // invoke local proxy
  lpUser_ = lpUsers_->getUser(name);
}

void RPUsers::getUserConvRec()
{
  // verify whether remote proxy exists
  rpUser_ = (RPUser*)node_->nameManager()->getProxy(lpUser_);

  if (!rpUser_) {
    // create new remote proxy
    rpUser_ = new RPUser(lpUser_,node_);
```

```
    // associate remote and local proxies
    node_->nameManager()->bind(rpUser_, lpUser_);
  }
}
```

The LOCAL PROXY USERS object invokes method getUser to USERS and receives a user object (user_). Afterwards, it is verified whether a local proxy for user_ exists already. If it is not the case, a new local proxy is created and associated with user_.

```
void LPUsers::getUserConvSend()
{
  // verify whether local proxy exists
  lpUser_ = (LPUser*)node_->nameManager()->getProxy(user_);

  if (!lpUser_) {
    // create new local proxy
    lpUser_ = new LPUser(user_,node_);

    // associate local proxy and object
    node_->nameManager()->bind(lpUser_, user_);
  }
}
```

The naming policy of the agenda application defines a global name for each distributed object, objects visible in several nodes. The global name is the local proxy reference. Different policies are possible by replacing NAME MANAGER objects.

Note that, for the sake of simplicity, the local proxy references were used as distributed names but this does not need to be always the case. In this particular solution, the creation of another entity, distributed name, is avoided and method getUserInv is defined as a simple call instead of conversion of the distributed name to a local proxy reference using the NAME MANAGER of central node.

5.4 Physical Distribution Design

5.4.1 Context

Different technologies exist to support partitioning, e.g. sockets and CORBA [Siegel 96]. Besides communication mechanisms, these technologies offer solutions for other distribution concerns, e.g. CORBA naming service.

The design and implementation of a non-physically distributed version of the logical distributed application is already developed.

5.4.2 Problem

How is logical distribution implemented using distribution technology?

5.4.3 Forces

1. **Design vs Mechanisms**

   Some of the design solutions developed during logical distribution may be supported by the technology mechanisms. For instance, CORBA supports distributed names integrated with the communication mechanism.

   So, should the design be replaced by solutions encapsulated inside mechanisms or should it be implemented on top of mechanisms?

5.4.4 Strategy

Concerns considered during logical distribution have to be revisited.

The communication concern should always be considered. Distributed proxies have to be redefined and implemented using the communication mechanism offered by the technology. Moreover, new classes may have to be included, e.g. if ACE [Schmidt 94] is used the Reactor pattern [Schmidt 95b] can be used.

Configuration, naming and replication concerns are revisited if the technology supports them.

Two situations should be considered: adaption and replacement. The former maintains the previous design adapting some of the classes, e.g. implementing distributed proxies using ACE. The latter, changes the previous design by removing the classes associated with the concern and incorporated during logical distribution, e.g. remove the name managers and support distributed names using CORBA naming service.

5.4.5 Result

The resulting application is implemented using the technology. Several possibilities of implementation may be tried out, e.g. sockets or RPC. Heterogeneity is achieved if logical nodes are implemented on different distribution technologies.

The replacement of design solutions with mechanisms improves application performance but reduces the application maintainability and extensibility. The programmer is left with the set of policies offered by mechanisms.

The final distributed application executes in a real distributed, and maybe heterogeneous, environment.

5.4.6 Example

Classes incorporated during logical distribution should be redefined. An ACE [Schmidt 94] implementations is shown.

ACE (Adaptive Communication Environment) is an object-oriented network programming framework that encapsulate operating system concrete mechanisms with C++ classes. In this example the ACE features for interprocess communication will be used.

This implementation follows an adaption strategy.

In this implementation physical central node is a server of physical agenda and physical manager nodes which are its clients.

The Figure 8 shows the enrichment done to the logical central node using ACE's Reactor pattern [Schmidt 95b]. The REACTOR object registers, removes and dispatches

EVENT HANDLER objects. The EVENT HANDLER object is used by the REACTOR to dispatch method invocations. The NODE ACCEPTOR object, which is part of the Acceptor pattern [Schmidt 95a], is used to decouple the act of establishing a connection from the services provided once a connection is established. CALL HANDLER objects are created for each connection and they are responsible for redirecting the call to a PHYSICAL LOCAL PROXY object.

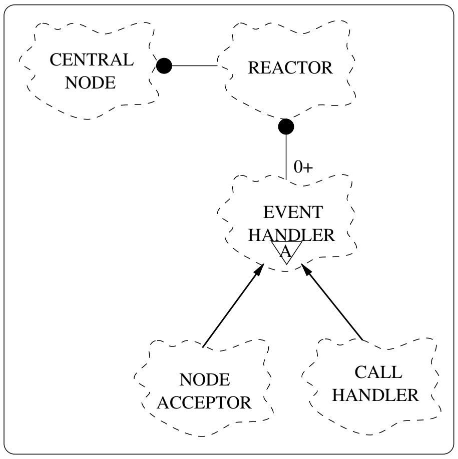

Figure 8: Physical Central Node.

A PHYSICAL CENTRAL CONFIGURATOR is defined for the central node. It is a subclass of logical CENTRAL CONFIGURATOR and it enriches method configure with the creation of REACTOR and ACCEPTOR objects.

```
int PhysicalCentralConfigurator::configure()
{
  // do logical configuration
  CentralConfigurator::configure();

  // create reactor
  ACE_Reactor reactor;

  // define address port
  ACE_INET_Addr addr(PORT);

  // create node acceptor event handler
  NodeAcceptor na(addr);

  // register acceptor
  reactor.register_handler(&na, READMASK);

  // starts dispatching
  reactor.dispatch();
}
```

Physical configurators are also defined for agenda and manager nodes. These physical configurators are subclasses of logical configurators, AGENDA CONFIGURATOR and MANAGER CONFIGURATOR. They redefine method connectCentralNode.

```
// returns a physical local proxy of central node
PhysicalLPCentralNode*
PhysicalAgendaConfigurator::connectCentralNode()
{
  // define address to PORT physical
  // address of central node
  ACE_INET_Addr addr(PORT);

  // ACE sock connector
  ACE_SOCK_Connector connector;

  // ACE communication stream
  stream = new ACE_SOCK_Stream;

  // connect central node
  connector.connect(*stream, addr, 0);
  connector.complete(*stream, &addr, 0);

  // uses stream to get from the central node
  // its physical local proxy
  return getPhysicalLPCentralNode();
}
```

Figure 9 shows a generic physical distributed invocation which is the ACE implementation of a logical distributed invocation. PHYSICAL REMOTE PROXY and PHYSICAL LOCAL PROXY are subclasses of respectively REMOTE PROXY and LOCAL PROXY.

PHYSICAL REMOTE PROXY redefines method mInv to do marshaling, write the packet to the socket, read from the socket and do unmarshaling. CALL HANDLER object reads proxy name from socket, gets proxy reference from the NAME MANAGER object and invokes invoke method on PHYSICAL LOCAL PROXY object. The PHYSICAL LOCAL PROXY reads (unmarshals) from the socket the method name and the method arguments, then it invokes method m defined in the logical phase. Finally, it is responsible for marshaling the results and write them to the socket.

6 Discussion

6.1 Forces

The forces driving the pattern language were resolved in the following way:

- **Procedural vs Declarative Languages**

  The pattern language distinguishes the specification of the problem from the specification of the solution. Partitioning requirements at the problem space are defined in the Distributed Requirements pattern. These requirements are further refined in the solution space entities in the Functionalities Design pattern and are used by the Logical Distribution pattern to identify which distribution concerns have to be considered and to establish the distributed architecture.

  The use of a procedural specification language may focus on the description of the solution space instead of on the problem space. So, the use of a procedural specification language should be carefully used in the Distributed Requirements pattern.

- **Transparency vs Non-Transparency**

  A main issue of the development of distributed applications is the level of transparency of distribution. Should the programmer be aware of distribution?

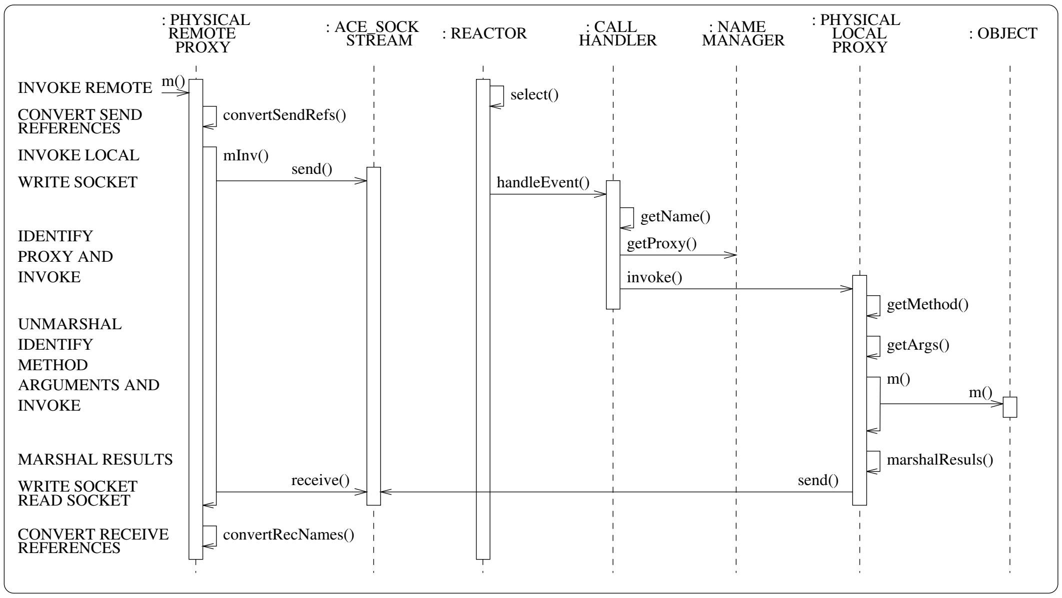

Figure 9: Physical Distributed Invocation.

The pattern language considers distribution in both the problem and the solution space. At the problem space, (Distribution Requirements pattern), distribution is non-transparent only if it is part of the users' expectations. In the Functionalities Design pattern, the distributed architecture is transparent but non-transparent requirements from the problem space should be fulfilled either by simulation or by their refinement as requirements at the solution space. In the Logical Distribution pattern a distributed architecture is defined and the pending distribution requirements are supported. The distribution concerns may be transparent, because they are not relevant for the pending distribution requirements, or the programmer does not need to know the complexities associated with the solution, because pre-defined solutions are used. Issues related with physical distribution are transparent during logical distribution. The Physical Distribution pattern revisits previous concerns in terms of their implementation in distribution technology and the use of abstract mechanisms allow the transparency of platform specifics.

- **Fixed vs Customizable Solutions**

  Todays distributed applications are requiring new flexible and customizable models of interaction. How are this requirements achieved at both, problem and solution space?

  The Distribution Requirements pattern describes the models of interaction users expect. Languages to describe these new models should be used. These models are implemented using design patterns in the Logical Distribution pattern. Each pattern abstracts policies for its concern and allows policy customization. The customization may be reduced if a replacement implementation is chosen in the Physical Distribution pattern with a possible performance gain.

6.2 Qualities

The pattern language holds the following qualities:

- **Expressiveness and support of models**.

  The separation in problem space and solution space avoids expressiveness bias. The languages used in the Distribution Requirements pattern should specify users' expectation and be free of solution bias.

  The Distribution Requirements pattern describes the models of interaction. The Logical Distribution pattern supports these models using customizable design patterns for each concern.

- **Performance constraints**.

  Performance requirements are implicit requirements which are satisfied in the Logical and Physical Distribution patterns. An object-oriented design is constructed first and it is afterwards transformed into a client/server design. Client/server design is delayed and based on tests and evaluations, large-grained objects are constructed during the final development stages.

  Functionalities design pattern define a (fine-grained) object-oriented design. Logical Distribution design pattern defines nodes which are large-grained objects, but inter-node communication is done between fine-grained objects. Performance requirements may determine a particular distributed architecture. Physical distribution design pattern constraints inter-node communication to inter-node channels. The number of channels between two nodes is determined by performance requirements.

- **Incremental simulation, test and debug**.

  The stepwise approach allows an incremental (horizontal) construction of specifications. This allows rapid development with further customization and optimization as well as incremental test and debug. Tests in a really distributed environment are delayed for the physical distribution stage.

  Logical Distribution design pattern is particularly suitable for tests and evaluations. Different implementations of logical proxies allow communication monitoring, fault simulation, communication delay simulation, and load balancing evaluation. Furthermore, the allocation of processes to physical nodes is simulated using logical proxies and configurators implementing the characteristics of the physical nodes. Actually, the different implementations constitute virtual machines where the logical distributed application executes.

- **Levels of detail**.

  Distributed applications are constructed using several levels of detail. Selective transparency and customization are two important qualities of the pattern language.

  Selective transparency can be based in levels of abstraction, e.g. in the Functionalities design concerns's solutions are transparent, and in separation of distribution concerns at the solution space, e.g. some of the concerns are ignored.

  The Logical Distribution design pattern allows customization of different policies for distributed concerns. Physical distribution can further customize the solution at the level of mechanisms, e.g. two physical nodes can communicate using one or two sockets without interfering with the logical distributed architecture design.

7 Related Work

Design solutions for the incorporation of multiprocessing into sequential code are presented in [Martin 95]. The solution is based on the surrogate pattern which is associated with design patterns for communication between lightweight and heavyweight processes.

DOCASE[Muhlhauser 93] defines a methodic approach to distributed programming which defines several concepts for a stepwise approach to partitioning. DOCASE describes a taxonomy of objects (configure, generate and relation) which characterize the distribution partitioning. DOCASE separates application specific algorithms from dynamic distribution algorithms, the operational code is separated from the functionalities code using a superimposition language. DOCASE distinguishes application configuration, supported by configure objects, from physical configuration, supported by logical nodes. It maps configure objects onto logical nodes.

In [Rasmussen 96b, Rasmussen 96a] the extension of MOSES [HS94] process and notation to address distribution is described. The extended MOSES notation offers representation for distribution issues as partitioning, replication and migration. Some of the activities proposed by the extended MOSES process are: user requirements elicitation, which defines distribution requirements at the problem space; partitioning and allocation, which defines the approach to partitioning as, for instance, client/server; event model construction, which defines the semantics of communication and service invocation; DCS classification, which defines the requirements for the distribution technology; security planning; and testing, which deals with the non-determinism problem.

The ROOM method [Selic 94] defines concepts, a language and a process for the development of real-time applications. ROOM language describes objects with state machines that are used, without discontinuities, from problem definition to solution construction. The ROOM objects are composed to model architectures of communicating concurrent objects. Room models constructed using the ROOM language are executable by a virtual machine, providing observability and controllability of the model state.

The G++ pattern language [Aarsten 96] defines an evolutionary development process that derives the final implementation from a prototype of the logical design. The pattern language is defined for the control systems domain. It considers several patterns, at the solution space, stressing non-distributed simulation and an evolutionary approach when moving from the logical design to physical architecture.

The ISO reference model for distributed processing ISO RM-ODP [ISO95] describes an architecture which is based on five viewpoints: enterprise, which is concerned about the purpose, scope and policies of the enterprise; information, which is concerned with information modeling; computational, which is concerned about interaction pattern between the components of the system; engineering, which is concerned about the design of distribution aspects; and technology, which is concerned with the provision of an underlying infrastructure.

8 Conclusions

This paper presents a pattern language for the perception design and implementation of distributed application partitioning. The language has been defined in the context of DASCo approach [Silva 95], which stresses the separation of distributed concerns in the development of distributed applications.

The pattern language supports specifications of both the problem and solution spaces, the Distribution Requirements pattern is used to describe the problem space. DASCo levels of abstraction are also supported by the pattern language: models are defined by distribution requirements; policies are part of logical distribution; and mechanism are used during physical distribution. The pattern language deals with application partitioning DASCo concern, at the problem space, and offers solutions for configuration, naming, replication and communication DASCo concerns, at the solution space.

The pattern language describes a stepwise development which offers the following qualities: expressiveness and support of a wide range of user models; smooth transformation of an object-oriented model onto a client-server model; selective transparency; customization; and incremental development, simulation and test.

As in [Martin 95], the pattern language emphasizes design solutions in terms of patterns and frameworks, instead of a new notation. However, the approach can be combined with a design notation and in particular with a notation for design patterns and frameworks. Current work [Berre 97] is being done to specify DASCo design patterns and frameworks using the role model approach [Reenskaug 96].

Application-specific code is separated from operational code, as in DOCASE [Muhlhauser 93], both by the separation of the functionalities pattern from the logical distribution pattern, and by the separation in the design patterns of application-specific from application-independent methods, e.g. respectively methods `configureApplication` and `configureNode` of `Configurator` class.

Implementations of the design patterns used in the logical distribution design constitute a virtual machine which allows different kinds of simulation as in G++[Aarsten 96] and ROOM[Selic 94]. Possible simulations include: nondeterministic simulation, fault simulation, and physical configuration simulation.

The pattern language is RM-ODP compliant: distribution requirements pattern conforms to engineering viewpoint, functionalities pattern conforms to computational viewpoint, logical distribution pattern conforms to engineering viewpoint, and physical distribution conforms to technology viewpoint.

Currently two other pattern languages are being designed for concurrency and failure DASCo concerns at the problem space. Associated with the pattern language for concurrency a framework[Silva 96b, Silva 96a] and design patterns[Silva 97, Silva 96c] have already been constructed.

An open research issue is the definition of languages for the problem space specification. For instance, in [Rusinkiewicz 91] it is described a constraint language for the specification of distributed data consistency. Languages need to be specified for the three distribution concerns at the problem space.

Acknowledgments. We would like to thank our collegues Francisco Rosa and Teresa Gonçalves as well as the partners of the DISGIS [DISGIS 96] (DIStributed Geographical Information Systems: Models, Methods, Tools and Frameworks) ESPRIT Project where the pattern language is being used.

References

[Aarsten 96] Amund Aarsten, Davide Brugali, and Giuseppe Menga. Designing Concurrent and Distributed Control Systems. *Communications of the ACM*, 39(10):50–58, October 1996.

[Bal 89] Henri E. Bal, Jennifer G. Steiner, and Andrew S. Tanenbaum. Programming Languages for Distributed Computing Systems. *ACM Computing Surveys*, 21(3):261–322, September 1989.

[Berre 97] Arne-Jorgen Berre, Jan Oyvind Aagedal, and António Rito Silva. SIMOD - An ODP-extended Role-Modeling Methodology for Distributed Objects. In *30-th Hawaii International Conference on System Sciences (HICSS-30), Object-Oriented Methods for Distributed Applications Track*, Wailea, Maui, Hawai'i, US, January 7-10 1997.

[Booch 94] Grady Booch. *Object-Oriented Analyis and Design with Applications*. The Benjamin/Cummings Publishing Company, Inc., 1994.

[Chiba 93] Shigeru Chiba and Takashi Masuda. Designing an Extensible Distributed Language with a Meta-Level Architecture. In *ECOOP '93*, pages 482–501, Kaiserslautern, Germany, July 1993.

[Cook 94] Steve Cook and John Daniels. Software isn't the real world. *Journal of Object-Oriented Programming*, 7(2):22–28, May 1994.

[DISGIS 96] DISGIS. Esprit Project 22.084: DIStributed Geographical Information Systems: Models, Methods, Tools and Frameworks, 1996. http://www.statkart.no/disgis.

[Gamma 95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1995.

[Hoydalsvik 93] Geir Magne Hoydalsvik and Guttorm Sindre. On the purpose of Object-Oriented Analysis. In *OOPSLA '93 Proceedings*, pages 240–255, Washington, DC, September 1993.

[HS94] Brian Henderson-Sellers and Julian M. Edwards. *BOOKTWO of Object-Oriented Knowledge: The Working Object*. Prentice-Hall, 1994.

[ISO95] ISO/IEC JTC1/SC21/WG7, ISO/IEC Copyright Office, Geneve, Switzerland. *ITU-T Recommendation X.901 | ISO/IEC 10746-1 ODP Reference Model Part 1. Overview*, May 1995.

[Jackson 95] Michael Jackson. *Software Requirements and Specifications*. Addison-Wesley, 1995.

[Johnson 92] Ralph E. Johnson. Documenting Frameworks using Patterns. In *OOPSLA '92, Proceedings*, pages 63–76, Vancouver, Canada, October 1992.

[Martin 95] Robert Martin. *Designing Object-Oriented Applications Using the Booch Method*. Prentice-Hall, 1995.

[Muhlhauser 93] Max Muhlhauser, Wolfgang Gerteis, and Lutz Heuser. DOCASE: A Methodic Approach to Distributed Programming. *Communications of the ACM*, 36(9):127–138, September 1993.

[Rasmussen 96a] G. Rasmussen, B. Henderson-Sellers, and G.C. Low. An Object-Oriented Analysis and Design Notation for Concurrency and Distributed Systems. *Journal of Object-Oriented Programming*, 1996.

[Rasmussen 96b] G. Rasmussen, B. Henderson-Sellers, and G.C. Low. Extending the MOSES Methodology to Distributed Systems. *Journal of Object-Oriented Programming*, 9(4):39–46,100, July/August 1996.

[Reenskaug 96] Trygve Reenskaug, Per Wold, and Odd Lehne. *Working With Objects: The OOram Software Engineering Method*. Manning Publications Co., 1996.

[Rodden 91] Tom Rodden and Gordon Blair. CSCW and Distributed Systems: The Problem of Control. In *Proceeding of the Second European Conference on Computer-Supported Cooperative Work*, pages 49–63, Amsterdam, The Netherlands, September 1991.

[Rusinkiewicz 91] Marek Rusinkiewicz, Amit Sheth, and George Karabatis. Specifying Interdatabase Dependencies in a Multidatabase Environment. *Computer*, 24(12):46–53, December 1991.

[Schmidt 94] Douglas C. Schmidt. The ADAPTIVE Communication Environment: An Object-Oriented Network Programming Toolkit for Developing Communication Software. In *11th and 12th Sun User Group Conferences*, San Jose, California and San Francisco, California, December 1993 and June 1994.

[Schmidt 95a] Douglas Schmidt. Acceptor and Connector: Design Patterns for Active and Passive Estabishment of Network Connections, August 1995. Presented at ECOOP'95 Workshop on Patterns Languages of Object-Oriented Programming.

[Schmidt 95b] Douglas C. Schmidt. Reactor: An Object Behavioral Pattern for Concurrent Event Demultiplexing and Event Handler Dispatching. In Jim Coplien and Douglas C. Schmidt, editors, *Pattern Languages of Program Design*. Addison-Wesley, 1995.

[Selic 94] Bran Selic, Garth Gullekson, and Paul T. Ward. *Real-Time Object-Oriented Modeling*. John Wiley and Sons, 1994.

[Siegel 96] Jon Siegel. *CORBA Fundamentals and Programming*. Wiley, 1996.

[Silva 95] António Rito Silva, Pedro Sousa, and José Alves Marques. Development of Distributed Applications with Separation of Concerns. In *Proceedings of the 1995 Asia-Pacific Software Engineering Conference APSEC'95*, pages 168–177, Brisbane, Australia, December 1995. IEEE Computer Society Press.

[Silva 96a] António Rito Silva, Luís Gil, and Joao Martins. Three-Layered Framework with Separation of Concerns: The Object Concurrency and Synchronization Framework, October 1996. Presented at the OOPSLA'96 Workshop on Exploration of Framework Design Principles.

[Silva 96b] António Rito Silva, Joao Pereira, and José Alves Marques. A Framework for Heterogeneous Concurrency Control Policies in Distributed Applications. In *Proceedings of the 8th International Workshop on Software Specification and Design*, pages 105–114, Schloss Velen, Germany, March 1996. IEEE Computer Society Press.

[Silva 96c] António Rito Silva, Joao Pereira, and José Alves Marques. Customizable Object Synchronization Pattern, July 1996. Presented at EuroPLoP '96.

[Silva 97] António Rito Silva, Joao Pereira, and José Alves Marques. Customizable Object Recovery Pattern. In Dirk Riehle Robert Martin and Frank Buschman, editors, *Pattern Languages of Program Design 3*. Addison-Wesley, 1997.

[Sousa 96] Pedro Sousa and António Rito Silva. Naming and Identification in Distributed Systems: A Pattern for Naming Policies. In *Conference on Pattern Languages of Programs, PLoP '96*, Allerton Park, Illinois, September 1996.

[WB90] Rebecca Wirfs-Brock, Brian Wilkerson, and Lauren Wiener. *Designing Object-Oriented Software*. Prentice-Hall, 1990.

[Yokote 92] Yasuhiko Yokote. The Apertos Reflective Operating System: The Concept and Its Implementation. In *Proceedings of the 1992 Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 414–434, Vancouver, Canada, October 1992.