# Component Configurer: A Design Pattern for Component-Based Configuration

Francisco Assis Rosa and António Rito Silva
INESC/IST Technical University of Lisbon
R. Alves Redol n°9, 1000 Lisboa, PORTUGAL
Tel: +351-1-3100287, Fax: +351-1-3145843
{fjar,ars}@albertina.inesc.pt

**Abstract**

*This paper presents a design pattern for component-based configuration. The pattern focuses on allowing the configuration of components along with its inter-component connections. It allows component creation, destruction and migration without affecting other components. The pattern promotes a decoupling between components and components connection, aiming at supporting ad-hoc dynamic reconfiguration and the migration of components with state transferring.*

## 1 Intent

Decouples component configuration from component functionality. It allows component ad-hoc dynamic configuration. It aims at providing a configuration of components with state transference. It complies with the concept of hierarchical configuration and migration clustering.

## 2 Motivation

How can a component-based application design take into account the problem of component configuration and in particular issues such as component creation, destruction, migration and connections changes ?

### 2.1 Domain Analysis

Configuration is frequently associated with evolutionary change of critical systems with long life duration. These systems typically need to evolve as human needs, technology or even application environment changes [Kramer 85]. These changes can range from changing existing functionalities to adding new ones.

Application configuration allows the specification and change at run-time of an application's building blocks and their collaboration structure.

Configuration can help solve problems such as:

- Software prototypes development - the use of a configurable programming base can help develop software in which different architectures can be tested with minimum effort.
- Flexible systems development - the production of an application with configuration capabilities can help obtain a higher flexibility for future application evolution. This flexibility can be of great importance in avoiding future legacy systems.
- Fault tolerance - damage caused by a node fault in a distributed application can be minimized by reconfiguring the elements which form the application. Elements running on the failed node can be re-deployed on new nodes and their cooperating elements instructed on where to redirect their cooperation requests.
- Critical systems evolution - systems considered to be critical need the capability of changing functionalities at run-time without stopping the system.

Due to its importance, a great amount of work has been done in this area and a great number of bibliographical references can be found. From this bibliography three main concepts can be identified:

- Component - the basic element of configuration. These elements can have a mapping to a piece of software developed in a programming language. In [Wegner 97] components are defined as entities with persistent identity and interfaces whose observable behavior is governed by a state. Components are almost never self-contained and usually interacting with other components.
- Connection - the support for cooperation between components. Components should cooperate using these elements.
- Application structure - description of components and its cooperation structure using connections. This description can include information

such as component mapping to the existing system nodes.

In [Kramer 85], configuration is classified into static and dynamic configuration and some required properties for dynamic configuration are presented. Static configuration is presented as the process of producing a load image of groups of components for each of the computer stations in the distributed system. This process is based on an application description using a configuration language. Dynamic configuration is presented as the process of modifying or extending an application without suspending its execution. This process is driven by change specifications such as, introduction of new components, modification of existing ones and modification of existing communication patterns.

In [Hofmeister 91], configuration is classified into module implementation configuration, structure configuration and geometry configuration. Module implementation configuration deals with the capability of specifying and changing a component's implementation. Structure configuration deals with the capability of specifying and changing the system's logical structure, i.e. the elements composing the system and the connections between them. Geometry configuration deals with the capability of specifying and changing the mapping of the application's components onto the distributed architecture. This work addresses a reconfiguration framework applied to the *Polylith* software bus [Purtilo 90], presenting the idea of component migration with state transference.

The work of *Conic* [Magee 89] presents a configuration language structuring components hierarchical using the concept of composite component. *Conic* presents, at the language specification level, dynamic configuration although its main contribution is in the scope of static configuration.

Design patterns for configuration issues include design patterns such as the *Service Configurator* pattern, the *Pipes and Filter* pattern, the *Callback* pattern and the *Broker* pattern.

The *Service Configurator* pattern [Jain 96] deals with service configuration into applications. It enables configuration and reconfiguration of services without affecting other services.

The *Pipes and Filters* pattern [Buschmann 96] deals with the configuration of pipeline based applications. It allows the change of a pipeline application by introducing, removing or changing pipeline elements.

The *Callback* pattern [Berczuk 95] addresses the issue of assembly and processing decoupling, providing some dynamic configuration at the component connection establishment.

The *Broker* pattern [Buschmann 96] presents a pattern for the establishment of cooperation between elements in a distributed application.

All these patterns present solutions to configuration issues. None of them however deals with the concept of component structure, e.g. hierarchical structure, and migration of components with state transference.

## 2.2 Example

Configurable application development encompasses both static and dynamic configuration. In particular dynamic configuration should consider topics such as changes in component connections and migration of components with state transfer.

Take the design of a shared agenda.

The shared agenda should have the following functionality:

- There are two kinds of sessions: Manager Session holds an Agenda Manager. Agenda Session is associated with a user and interacts with an Agenda Manager.
- Users are created, updated and deleted by a Manager Session using its Agenda Manager.
- Users create, update and delete personal (private) appointments through one Agenda Session.
- Users create, update and delete meetings through one Agenda Session. A meeting requires its creator and several participants.
- A user consults its agenda to see his appointments and the meetings he should participate using one Agenda Session.

A possible class diagram for this problem is presented in figure 1 using the Booch class diagram notation [Booch 94]. This class diagram presents the necessary classes for the basic Agenda functionality.

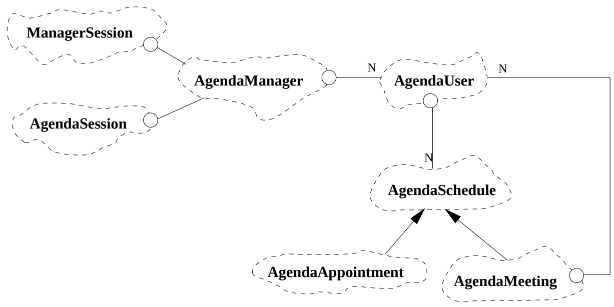


Figure 1: The shared agenda class diagram

An Agenda Manager will always be in charge of the manipulation of its list of users. Agenda Sessions will consult the agenda data by using this Agenda Manager.

Configuration aspects can appear with the following requirement:

- Several sessions may be executing simultaneously but only a single instance of Agenda Manager may exist at each moment.
- Only the highest ranked Manager Session will own the Agenda Manager. This requirement corresponds to the idea of the higher ranking manager being the only one allowed to manipulate users. Lower ranker Manager Sessions will have its managing functionality deactivated.

Figure 2 presents the architecture for the agenda application. Agenda Sessions communicate with the Agenda Manager contained in the Manager Session. The Agenda Manager contains several other objects. It contains user data objects (*AgendaUser* objects), appointment data objects (*AgendaAppointment* objects), and meeting objects (*AgendaMeeting* objects).

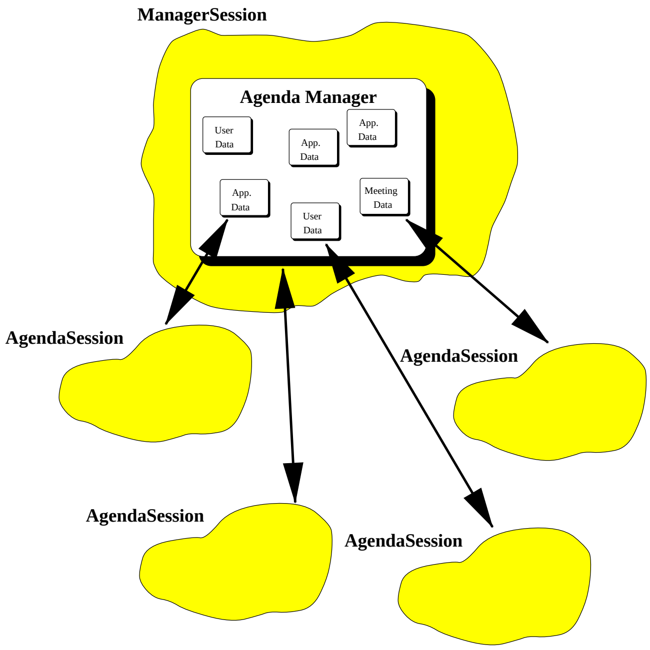


Figure 2: The shared agenda application

The deployment of a new higher ranker Manager Session will involve reconfiguration of the existing system. The Agenda Sessions should now "talk" with the new Agenda Manager and the new data managing objects (see figure 3).

This new configuration corresponds to migrating the agenda manager and its specific data managing objects to the new Manager Session's location.

The nature of this problem involves the existence of reconfiguration capabilities for the resulting application where the instant or the nature of reconfiguration is not defined at start. This problem requires capabilities for migrating existing software components and

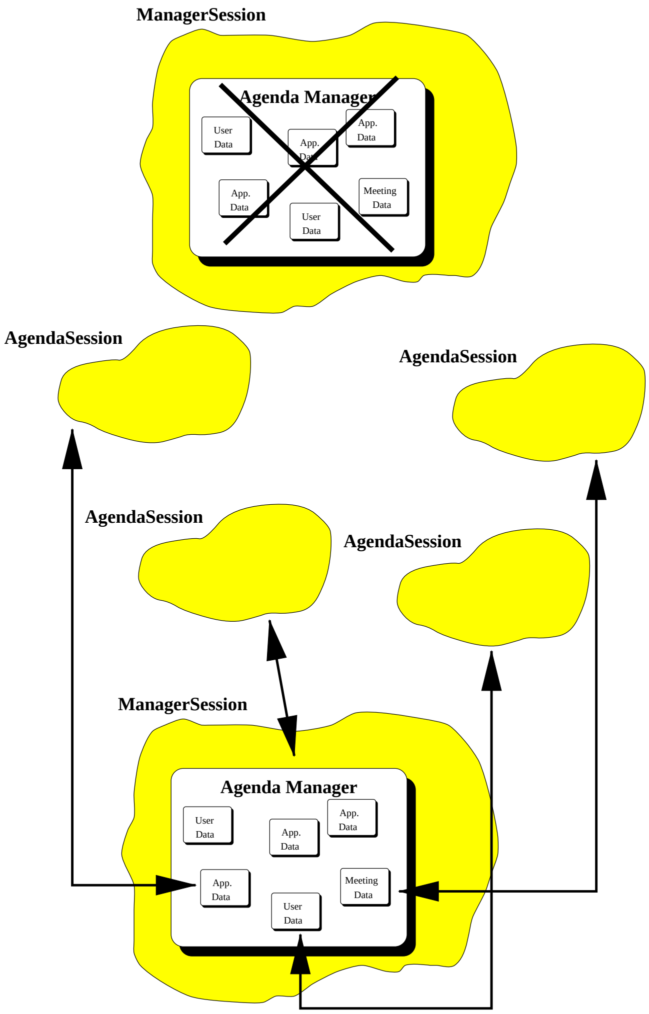


Figure 3: The shared agenda with a new Manager Session

reconfiguring cooperation connections between components at run-time. It also needs the existence of some kind of component composition technique along with its migration. This is the case when the Agenda Manager migration is concerned.

For performance optimization purposes, further reconfiguration requirements can be introduced:

- In order to optimize the data access, whenever an Agenda Session is created, it's user specific data is placed near the Agenda Session. This user specific data includes user data and user appointment data.

Again this requirement can be satisfied using configuration. Whenever an Agenda Session is deployed, it's corresponding user data is migrated from the Manager Session's Agenda Manager to the Agenda Session. When the Agenda Session concludes its execution, all the important data is again placed at the Agenda Manager. Figure 4 presents such a performance optimized architecture.

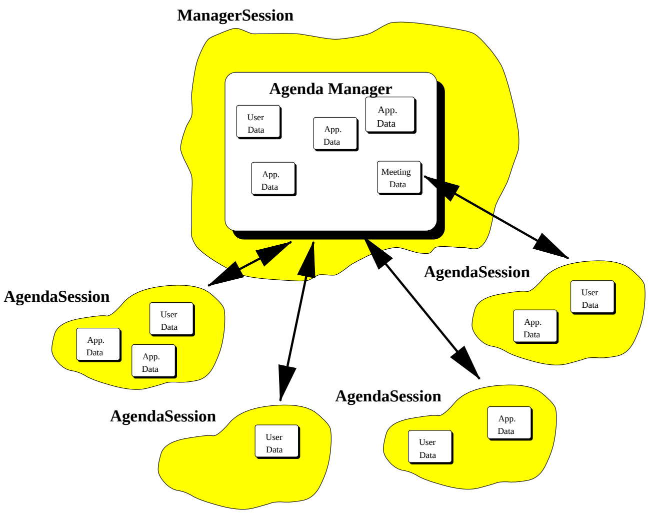


Figure 4: The shared agenda with data migration

## 2.3 Problem

Component-based applications are becoming increasingly common. One of the problems which should be addressed on these applications is the problem of being able to specify and change the components which form the application and their cooperation structure. How can a design take these considerations into account for a particular component-based application ?

## 2.4 Solution Objectives

A solution to this problem should have the following objectives:

- **Flexibility** - resulting applications should be flexible in the definition and change of components, component communication structure and component placement.

- **Static and dynamic configuration** - configuration abstractions should provide both static and dynamic configurations capabilities.

- **Incremental Development** - the incremental development of configuration properties into components should be an issue. In particular functional application prototypes should be capable of being enriched with configuration capabilities in a stepwise process.

## 2.5 Forces

The problem must consider the following forces:

- **Flexibility vs Performance** - flexibility is one of the objectives of the solution. However flexibility may result in a loss of application performance. There should be a trade-off between these two forces. According to the nature of applications being developed some issues of flexibility should be tradeable for better performance.

- **Incremental Development vs Class Explosion** - the process of incremental development can reduce the complexity of developing applications in one step. However the cost to pay for this incremental development can be the explosion of classes involved in the development.

## 2.6 Solution

The solution is a set of class structures for component-based configurable applications development. This solution takes into account issues such as component deployment in several "logical nodes", component destruction, establishment and change of connections between these elements and finally migration of these components with transference of its state.

The pattern offers an abstract structure which should be customized by developers to each specific application.

This solution takes into account the forces previously named. It favors flexibility as opposed to performance providing capabilities for component-based application composition definition and changes. It also favors incremental development as opposed to class explosion allowing the development of applications in several steps. Being integrated in the DASCo framework, this solution will allow the incremental combination of configuration with other concerns, e.g. distribution.

# 3 Applicability

The component-based configuration pattern should be used whenever one of the following apply:

- The application being developed is component-oriented with interaction between components.
- The application being developed is bound to suffer high architectural changes during development, therefore requiring a highly configurable and dynamic programming base.
- The application being developed must be dynamically configurable to satisfy the application requirements.
- The concept of configurable component clustering is present when thinking of component migration, i.e. grouping of configurable components which act as logical nodes that must be migrated as one single component.

# 4 Structure and Participants

## 4.1 Configurable Component Structure

The Configurable Component hierarchy provides the necessary abstractions for configurable component construction. A reconfigurable component will be a component capable of altering its interaction structure with other components and capable of transferring state to other components of the same class. Figure 5 presents a Booch class diagram [Booch 94] for the configurable component hierarchy. The main elements in this hierarchy are:

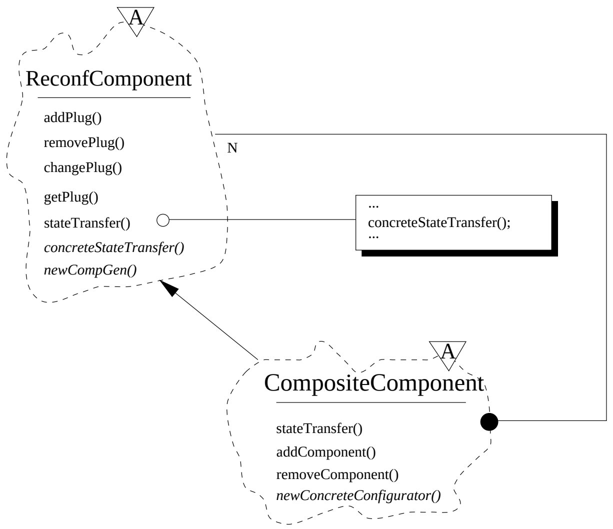


Figure 5: The Configurable Component Hierarchy

- *ReconfComponent* - defines the necessary interface for a reconfigurable component. It defines the reconfigurable component interface manipulation methods and the state transfer method for a basic reconfigurable component. Concrete reconfigurable components should derive from this base class. Methods *addPlug*, *removePlug* and *changePlug* will allow a reconfigurable component to add new *Plugs* accessing other components, change these *Plugs* or remove them (see section 4.3 for a description on *Plugs*). The *stateTransfer* method will allow the transfer of the component's state to another component. This method implements the *Plug* transfer between components and uses the *template method* pattern to allow customization of concrete reconfigurable components. Methods *concreteStateTransfer* and *newCompGen* are abstract methods. In *concreteStateTransfer*, specific class state transference behavior should take place. In *newCompGen*, the generation of concrete *ComponentGenerator* objects should take place.
- *CompositeComponent* - provides the necessary abstractions for component nesting and configuration. A *CompositeComponent* should be capable of nesting its reconfigurable components and be capable of directing the configuration (migration, interface redefinition, creation and destruction) of these reconfigurable components. It defines a particular configuration policy for its enclosed components.

Specific configurable components should derive either from *ReconfComponent* or from *CompositeComponent*. In both cases this new specific component should provide a definition for *concreteStateTransfer* where the component specific state transfer operation should occur and *newCompGen* where the generation of a class-specific *ComponentGenerator* should be performed.

## 4.2 *CompositeComponent* Structure

The *CompositeComponent* structure describes the class structure necessary to obtain configurable component nesting and configuration.

Figure 6 presents a Booch class diagram [Booch 94] for the *CompositeComponent* class structure. The main elements in this structure are:

- *CompositeComponent* - as previously presented.
- *ReconfComponent* - as previously presented.
- *NameManager* - associates components names with its location (its *Plug*). It will be capable of obtaining a component's location from it's name

through the use of it's *resolveName* method. Registration and de-registration of name/location pairs is performed through the use of the *bind* and *unbind* methods.

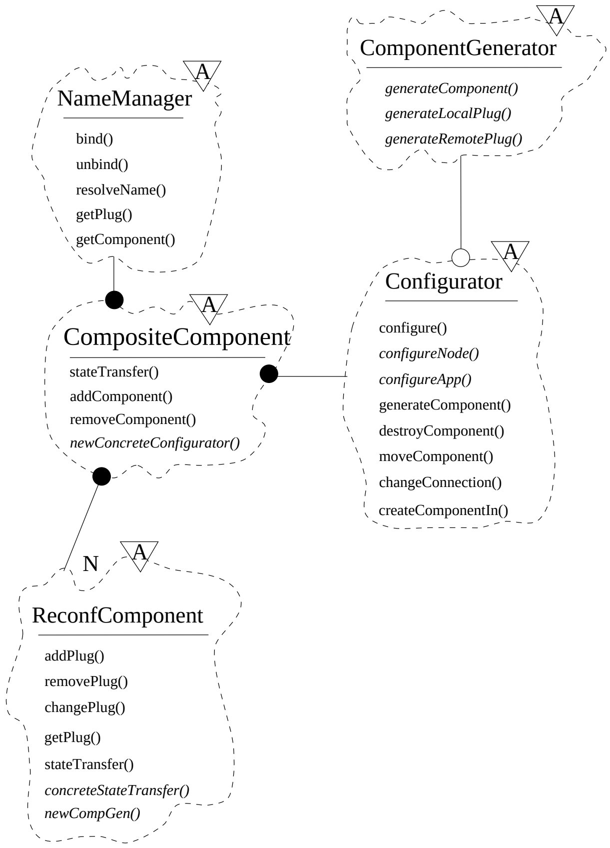


Figure 6: *CompositeComponent* class structure

- *Configurator* - implements the basic interface for component configuration. Possesses all the necessary interaction behavior between configurators to attain configuration. It is responsible for the configuration of the *CompositeComponent* at the time of its creation. It has three methods for this configuration: *configure*, *configureNode* and *configureApp*. The *configure* method calls the *configureNode* method followed by a call to the *configureApp* method. It should be used for full *CompositeComponent* configuration. At *configureNode*, node-specific configuration is performed namely *NameManager* initialization, generation and registration of plugs for external requests. At *configureApp*, application-specific configuration is performed, namely sub-component generation and connection establishment.

  *ReconfComponents* placed inside the *Configurator's CompositeComponent* are also configurated by the *Configurator*. Methods *generateComponent*, *destroyComponent*, *moveComponent* and *changeConnection* handle the configuration of *ReconfComponents* placed at *CompositeComponent*. Method *createComponentIn* handles requests for the creation of components on remote *CompositeComponent*.

- *ComponentGenerator* - implements the component generation functionality. It decouples the component configuration (placed in *Configurator*) from component generation. This decoupling allows a generic behavior of the *Configurator* class. Application specific behavior such as creation of application-specific components is placed in the *ComponentGenerator* class.

### 4.3 *Plug* Structure

Configurable components should communicate using *Plugs*. *Plugs* will be responsible for inter-component cooperation. To achieve maximum configuration capabilities all cooperation between components should be performed through these elements. Figure 7 presents the component cooperation structure.

The main elements in this structure are:

- *ReconfComponent* - it will have an *LPlug* and a set of associated *RPlugs*. To a *ReconfComponent* its corresponding *LPlug* will represent the entry point for requests from other reconfigurable components. The set of associated *RPlugs* will represent remote representations of other components.

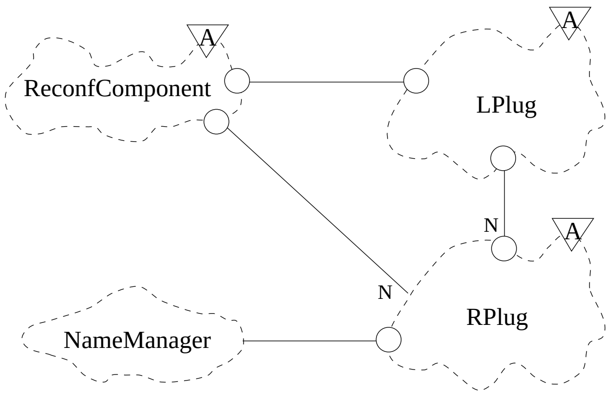


Figure 7: *Plugs* for Component cooperation

- *LPlug* - intercepts any order coming from a *RPlug* object. This class represents the entry point to request from other components.

  The *LPlug* delivers processing requests to its related *ReconfComponent*.

- *RPlug* - offers a local representative of a remote component. *RPlugs* are associated with the *Name* of the remote component. Using *NameManager's resolveName* method, an *RPlug* can reach its corresponding *LPlug* for communication.

  *RPlugs* decouple a component from its remote component location, therefore allowing remote component reconfiguration.

- *NameManager* - as previously presented.

From classes *RPlug* and *LPlug*, specific cooperation plugs are derived. These specific *Plugs* should conform to its corresponding component class interface.

To achieve maximum configuration, a reconfigurable component's code should use references to these *RPlugs* instead of referencing other configurable components directly.

# 5 Collaborations

As far as configuration is concerned there are four important functionalities which must be supported by the pattern: reconfigurable component generation, reconfigurable component migration, connection establishment between reconfigurable components and reconfigurable component destruction.

## 5.1 Reconfigurable Component Generation

A Reconfigurable Component generation involves cooperation between classes *Configurator*, *CompositeComponent*, *NameManager* and *ComponentGenerator*. A component generation is triggered by a request to *Configurator*. The first step for the *Configurator* is to request a new name from *NameManager*. The *Configurator* then interacts with *ComponentGenerator* which is responsible for the new component allocation and creation of the component's *LPlug* for future cooperations. The name obtained from the *NameManager* can now be associated with the component created by *ComponentGenerator*. The generation process ends with the *Configurator* placing the new generated component in its corresponding *CompositeComponent*. Figure 8 presents this collaboration.

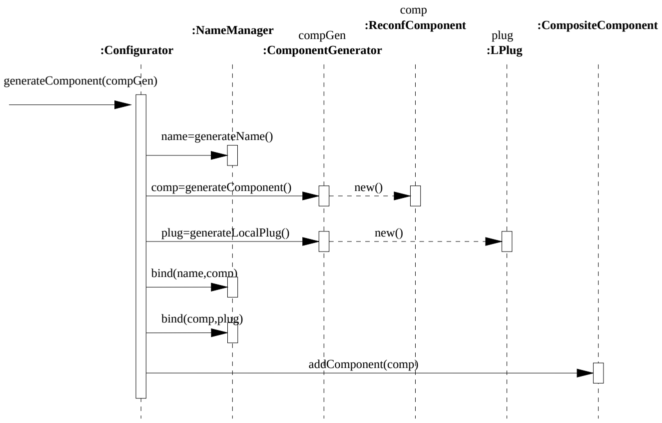


Figure 8: Reconfigurable Component Generation Collaboration Diagram

## 5.2 Reconfigurable Component Migration

A reconfigurable component migration involves cooperation between classes *Configurator*, *NameManager* and *ReconfComponent*. A component migration is triggered by a request to *Configurator*. The first step for the *Configurator* is to obtain the component associated with the given name. After this, the *Configurator* must obtain from this *ReconfComponent* a *ComponentGenerator* object capable of generating an instance of this *ReconfComponent*. This is done by calling the *newCompGen* method on *ReconfComponent*. Using this *ComponentGenerator* the *Configurator* cooperates with a remote *Configurator* in order to create the migrating component on another *CompositeComponent*. Next the *Configurator* interacts with the *ReconfComponent* by ordering it to transfer its state to the new component on the remote *CompositeComponent*. It is then that the *Configurator* renames the new component giving it the name of the old component and unregistering the location of the old component from *NameManager*. The process ends by destroying the old component.

Figure 9 presents this collaboration.

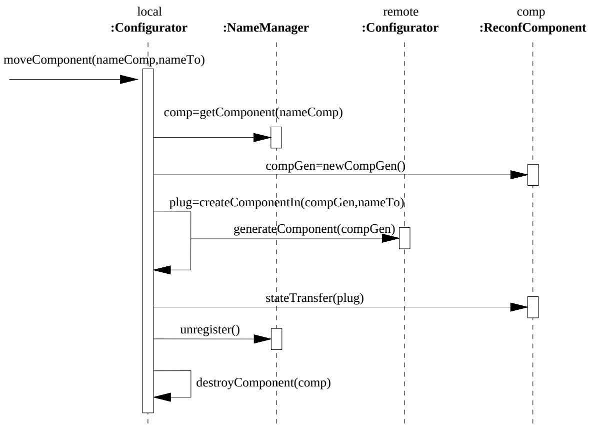


Figure 9: Reconfigurable Component Migration Collaboration Diagram

### 5.3 Reconnection Between Components

A Reconnection operation between components involves cooperation between classes *Configurator*, *NameManager*, *ReconfComponent* and *RPlug*. A component reconnection is triggered by a request to *Configurator*. This request identifies the original component whose connection must be changed, the connection that must be changed and the component to which the connection must be redirected to. The first step for the *Configurator* is to obtain the component whose connection must be changed. From this component the *Configurator* component can obtain the *RPlug* used as interface. Finally the connection can be redirected by invoquing the *reconfigureConnection* method in *RPlug*. Figure 10 presents this collaboration.

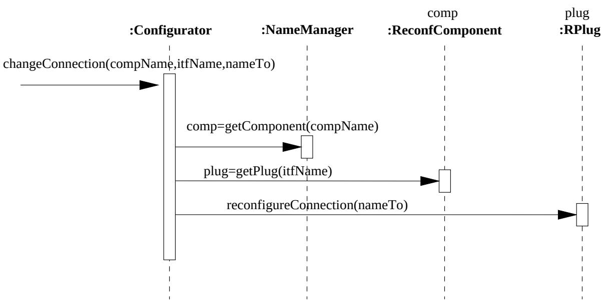


Figure 10: Reconnection between components

### 5.4 Reconfigurable Component Destruction

A reconfigurable component destruction involves cooperation between classes *Configurator*, *CompositeComponent* and *NameManager*. A component destruction is triggered by a request to *Configurator*. The *Configurator* interacts with *NameManager* obtaining the *LPlug* for this component. For this *LPlug* the *Configurator* gets its location. It now interacts again with *NameManager* un-binding the location and *LPlug* from it. As a final step it removes the component from its *CompositeComponent* and finally un-binds the component's name from the *NameManager*. Figure 11 presents this collaboration.

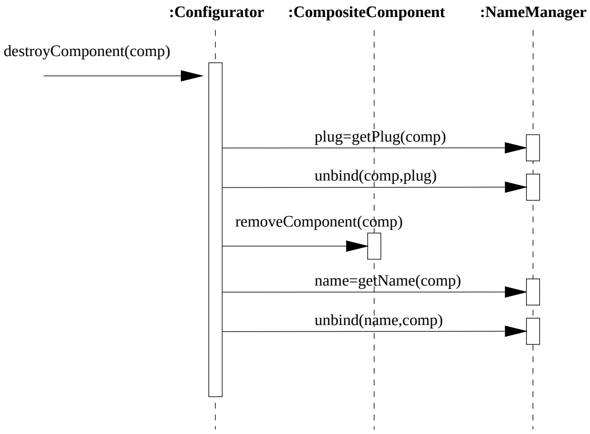


Figure 11: Reconfigurable Component Destruction Collaboration Diagram

## 6 Consequences

The use of the pattern has the following advantages:

- Configuration/computation decoupling - the use of this pattern allows strong decoupling between configuration and computation. The component's development effort can be directed to the functionality development. Configuration aspects are well isolated from the component's functionality and partially automated.

- Dynamic application development - the use of this pattern allows the development of highly dynamic applications. The use of elements such as *RPlug* and *LPlug* introduces location transparency at the component level allowing, among other operations, component migration to proceed transparently for components using migrating components.

- Configuration centralization - the configuration aspects of applications is located within the *Configurators* objects. These elements are the ones responsible for the application's structure therefore simplifying any application configuration code replacement.

- Concern separation - the presented pattern provides a solution to the specific concern of configuration. The pattern presents a design solution which promotes the decoupling of the configuration concern from other concerns such as naming and distribution.

- Migration of components - the pattern foresees the possibility of component migration with state transfer in opposition to simple stateless component migration.

The use of the pattern has the following disadvantages:

- Added complexity - the use of this pattern causes some class explosion which adds complexity to the overall design.

- Performance reduction - due to the introduction of a series of indirections, the use of the pattern can reduce performance.

# 7 Implementation

Several implementation variations can be introduced when implementing this pattern: *NameManager* variations, *Rplug's Name* evaluation variation and distribution introduction variation.

## 7.1 *NameManager* Variation

Different *NameManagers* can be used with this pattern. The used *Name's* properties can have some impact on the degree of configuration offered by the pattern.

In [Sousa 96] different variants for *Names* are presented.

Names are universal if they are valid in all applications that refer to an object and local if otherwise. Names are absolute if they refer to the same object in all applications where its valid and relative if they refer to different objects in different applications. Names can be pure if they contain no information concerning the objects they denote and impure otherwise.

The presented pattern deals with names as universal, absolute and pure. Different properties can be admitted, however this use of different properties can have some influence in the degree of configuration allowed.

Universal names are allowed by having every name being resolvable on every *CompositeComponent*. Local names are allowed simply by having names which are local to *CompositeComponents*.

Absolute and relative names can be supported in a similar way. Absolute names are supported by all names being interpreted the same way on every *CompositeComponent* and relative names by having different name interpretation on different *CompositeComponents*.

Pure and impure names are also allowed by the pattern. The resolution of names in the case of impure names will be greatly simplified. Impure names will be built out of a component's location and can therefore represent direct references to a component.

The nature of the application can therefore allow different properties for names.

Applications with a great number of component reference exchange between *Composite- Components* would be better suited to use universal and absolute names.

Applications in which clusters of *CompositeComponents* can be identified could use local and relative names.

Applications where reconfiguration is strictly at the component creation, destruction and connection reestablishment level could use impure names and gain the performance enhancement of using these names while applications using migration should use pure names.

Being aimed at applications with a great number of component references exchanged, where reconfiguration includes migration the pattern is better suited for universal, absolute and pure names.

## 7.2 *RPlug's Name* Evaluation Variation

The instant when *Names* are evaluated at the *RPlug* objects can have some impact on the implementation of this pattern. Two variations can be implemented: evaluation at the time of access and evaluation at the time of creation.

Evaluation at the time of access means that every time a *RPlug* needs to access its corresponding *LPlug*, its *Name* is resolved by the *NameManager*. This basically means that there shouldn't be any "dangling references" to destroyed or migrated *LPlugs*.

Evaluation at the time of creation means that *Names* are evaluated when the *RPlug* is created. This results in "dangling references" to destroyed or migrated *LPlugs*. The solution to such a problem involves enhancing the *LPlug* objects with the capability to reference not only *ReconfObjects* but also, if necessary, other *RPlug* objects and by this way implementing forwarding functionalities. *LPlug* components should also keep a counter of *RPlug* references to it. Each time a *LPlug* object is ordered to be destroyed it should check its counter and if necessary stay alive to receive incoming requests. If destruction should take place, *LPlug* components should return error messages to the requesting *RPlugs* which in turn should re-evaluate its *Name*. If migration should take place, *LPlug* should respond by redirecting the request to a new *RPlug* to the new component location. This variation involves some kind of garbage collection which is implemented by the existence of the counter and operations resulting from its change.

The nature of the application can have an influence on the choice of the variation to be used. Applications where component migration seldom occurs will be more suited to use the evaluation at the time of creation variation. On the other hand, applications where component migration may be frequent gain by using the evaluation at the time of access variation.

### 7.3 Distribution Variation

Distribution can also be added to the pattern. The pattern is well suited for compliance with concepts such as nodes and distributed objects.

The Configurer pattern introduces two levels of abstraction: *ReconfComponent* and *CompositeComponent*. From the distribution point of view, top-level *CompositeComponents*, i.e. *CompositeComponents* not contained in others, are well suited for node representation. Contained Components can be configured at each *CompositeComponent* and migrated between *CompositeComponents*.

Since communication between configurable components occurs via *RPlug* and *LPlug* objects, distribution can be introduced by enhancing classes *RPlug* and *LPlug* in order to support distribution. One possible implementation to this enhancing could be the combination of these elements with distributed proxies elements. Also locations used by *RPlugs* should be altered to express distributed locations.

There are to ways of enhancing these classes: delegation and inheritance. Using delegation, one can enhance these classes by having *RPlugs* and *LPlugs* reference its distributed proxies and by using them to send out its messages. Using inheritance, one can enhance these classes by creating a class *DistRPlug* derived from both *RPlug* and the distributed Proxy. Distributed communication can thus be achieved by using this class. The same process can be done on the *LPlug* class.

Distributed proxies will also provide a way to encapsulate specific communication mechanisms. These proxies can be socket-based, pipe-based, can use framework mechanisms such as *Reactor* and *Acceptor* [Schmidt 96] from the*ACE* framework [Schmidt 94] or even systems such as *Orbix* [Technologies 96] for component communication purposes.

A possible variation using the *ACE* framework can be shown.

The Component Configurer pattern is combinable with the Service Configuration pattern allowing the jump from the higher level components to the low-level socket-based connection.

Going from logical distribution to physical distribution involves identifying which are the components to be placed in diferent physical nodes. As previously stated, top-level *CompositeComponents* allow this identification.

The physical distribution implementation using the Service Configuration pattern can be achieved by adaptation mainly at the *Plugs* level in the following way:

- Each top-level *CompositeComponent* will have one *ACE_Service_Config* instance and one *ACE_Acceptor* derived class instance. This *ACE_Acceptor* derived class instance will be responsible for the acceptance of all communications into this physical node and will be responsible for the generation of node-specific *ACE_Svc_Handler* derived classes instances. Instances from classes derived from *ACE_Acceptor* and *ACE_Svc_Handler* will be registered into the *ACE_Service_Config* instance.

  The *ACE_Svc_Handler* derived class instances will be responsible for communication into the node. They will be in charge of receiving incoming messages, and calling the correct method on the correct *LPlug* using the correct parameters. The *LPlug*/method determination will be based on identifiying information sent in the message.

- Component *Names* are resolvable into the socket address of it's corresponding top-level *CompositeComponent's ACE_Acceptor* derived class instance.

- An *RPlug* connecting to a remote component will use an *ACE_Connector* derived class instance to establish connection to the remote node. It will then use, after a successfull connection, an *ACE_Svc_Handler* derived class instance for node interaction. This *ACE_Svc_Handler* derived class instance will be responsible for message marshalling with addition of target component and method identification.

  Only one pair *ACE_Connector* / *ACE_Svc_Handler* derived classes instances will exist on a node for connection to one other remote node, being this pair shared between *RPlugs* connecting to that same node.

This way there will be at most one opened socket between each pair of physical nodes and communication will occur through the *ACE* framework.

Figure 12 and figure 13 presents the structure of the use of the Service Configurator pattern in the Configurer pattern.

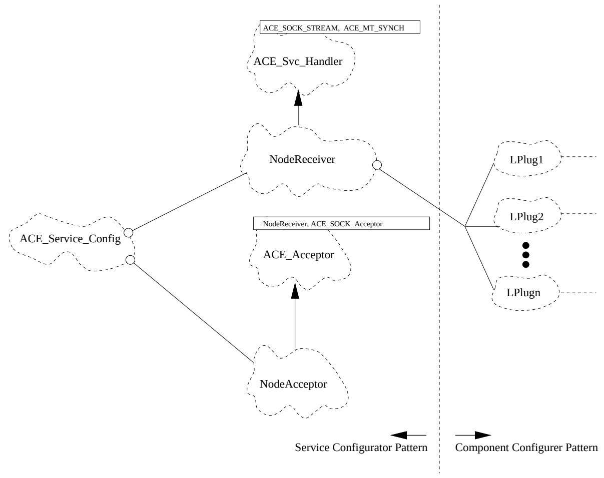


Figure 12: Local Plug class diagram using the *ACE* framework

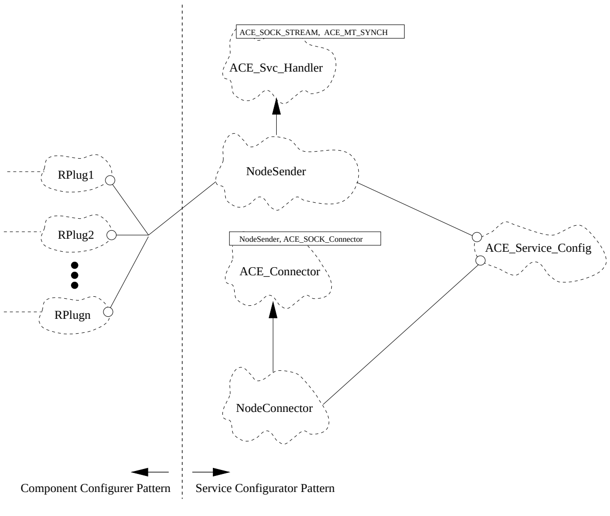


Figure 13: Remote Plug class diagram using the *ACE* framework

This way, the Service Configurator pattern complements the Component Configurer pattern by allowing process service configuration as opposed to component configuration.

# 8 Sample Code

The presented pattern has been applied to the shared agenda example.

Classes *AgendaUser*, *AgendaAppointment* and *AgendaMeeting* (from this point on referred to as the agenda specific classes), should act as reconfigurable components. Objects from these classes will have to be able to migrate with state transference. These classes will inherit from *ReconfComponent* the reconfigurable component behavior. For each of these classes, abstract methods *concreteStateTransfer* and *newCompGen* will have to be defined. Also, a remote plug class and a local plug class will be defined for each class. To attain maximum configuration capabilities, all intercomponent communication should now be performed through this plug classes.

The following code presents class *AgendaUser* and its *concreteStateTransfer* method. As presented, this method makes use of an *AgendaUser* specific remote plug in order to communicate with the remote *AgendaUser* for state transference.

```
class AgendaUser: public ReconfComponent
{
  public:
    AgendaUser (char*, CompositeComponent*);
    void addAppointment (RPAgendaAppointment*);
    void addMeeting (RPAgendaMeeting*);
    char* getUserName (void);

    RPlug* concreteStateTransfer (Name*);
    ComponentGenerator* newCompGen (void);
  private:
    char                        _userName[30];
    List<RPAgendaAppointment>   _appList;
    List<RPAgendaMeeting>       _meetList;
};

RPlug* AgendaUser::concreteStateTransfer (Name* toName)
{
// _parentComponent is inherited from ReconfComponent and
// references it´s parent CompositeComponent
  RPAgendaUser* rpau = new RPAgendaUser(_parentComponent,toName);
  ListIterator<RPAgendaAppointment> ait(&_appList);
  RPAgendaAppointment* el1;
  while ( !ait.over() )
  {
    el1 = ait.getElement();
    if ( el1 )
      rpau->addAppointment(el1);
    ait.next();
  }
  ListIterator<RPAgendaMeeting> mit(&_meetList);
  RPAgendaMeeting* el2;
  while ( !mit.over() )
  {
    el2 = mit.getElement();
    if ( el2 )
      rpau->addMeeting(el2);
    mit.next();
  }
  return rpau;
}
```

Class *AgendaManager* acts as a composite component, more specifically as composite of the agenda specific classes. Objects from this class must be able to migrate, since *AgendaManager* must migrate between *ManagerSessions*, but must also be able to configure any agenda specific object which it contains. Class *AgendaManager* can therefore be enhanced by inheriting from class *CompositeComponent* as presented in the following code.

```
class AgendaManager: public CompositeComponent
{
  public:
    AgendaManager (CompositeComponent*);
    ~AgendaManager (void);

    RPAgendaUser* addUser (char*);
    RPAgendaUser* getUser (char*);
    List<RPAgendaUser>* getUserList (void);
    RPAgendaAppointment* addAppointment
           (RPAgendaUser*, AgendaDate&, AgendaTime&);
    RPAgendaMeeting* addMeeting
           (RPAgendaUser*, AgendaDate&, AgendaTime&);

// Defined for CompositeComponent inherited behavior
    Configurator* newConcreteConfigurator ();

// Defined for ReconfComponent inherited behavior
    RPlug* concreteStateTransfer (Name*);
    ComponentGenerator* newCompGen (void);
  private:
    List<RPAgendaUser>  _userList;
};
```

Finally, classes *ManagerSession* and *AgendaSession* act as composite components. A *ManagerSession* object will be a holder for an *AgendaManager* object. To satisfy the higher rank *ManagerSession* requisite, the *AgendaManager* contained in a *ManagerSession* may have to migrate between *ManagerSessions*. This migration will be controlled by the higher ranked *ManagerSession's Configurator*. This configurator will collaborate with the current *ManagerSession AgendaManager* holder to perform this migration.

The following code presents the *AgendaManager* migration procedure implemented using the functionalities in the pattern. The *ManagerSession's Configurator* is used to perform the migration of the *AgendaManager* object to the higher ranked *ManagerSession*.

```
Name* ManagerSession::giveUpAgendaManager (Name* giveTo)
{
  AgendaManager* am = getAgendaManager();
  if ( am )
  {
    Name* name = _nameManager->getName(am);
    if ( giveTo && name )
    {
      _configurator->moveComponent(name,giveTo);
      setAgendaManager(0);
      return name;
    }
  }
  return 0;
}
```

Figure 14 presents a diagram with the nested structure promoted by the use of this pattern. The diagram presenting a specific scenario of the shared agenda shows how *AgendaManager* objects will be placed inside *ManagerSession* objects and be configured by *ManagerSessionConfigurator* using *ManagerSessionNameManager*. Specific agenda classes objects such as *AgendaUser*, *AgendaAppointment* and *AgendaMeeting* will be placed inside *AgendaManager* objects and by configured by *AgendaManagerConfigurator* using *AgendaManagerNameManager*. *AgendaManager* objects will migrate between *ManagerSession* objects taking along all its agenda specific objects.

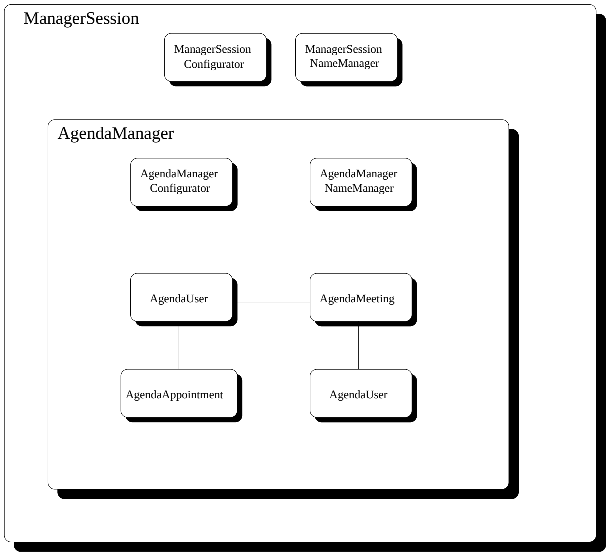


Figure 14: The Pattern applied to the Distributed Agenda

# 9 Known Uses

The presented pattern was developed in the scope of the DASCo framework [Silva 95]. Within DASCo this pattern represents a solution to the configuration concern. According to the DASCo development process, the pattern may be combined with other concerns, namely Naming and Replication [Silva 96].

The concept of *ReconfComponent* as primitive component and *CompositeComponent* as composite component is used in the configuration language *Darwin* [Magee 94].

The concept of configurator cooperation for the application configuration is used in the *Olan* system [Belissard 96].

# 10 Related Patterns

The presented pattern is related to the following patterns:

- *Composite* [Gamma 95] - the *CompositeComponent* represents a use of the *Composite* pattern.
- *Template Method* [Gamma 95] - the use of methods such as *concreteStateTransfer* in the *ReconfComponent* class, represents a use of the *Template Method* pattern.
- *Proxy* [Gamma 95]- the use of classes such as *RPlug* and *LPlug* can be seen as a particular use of the *Proxy* pattern.
- *Abstract Factory* [Gamma 95]- class *ComponentGenerator* represents a use of the *Abstract Factory* pattern. Classes derived from *ComponentGenerator* will encapsulate the generation of concrete *ReconfComponent*, *LPlug* and *RPlug* objects,
- *Factory Method* [Gamma 95]- method *newCompGen* in class *ReconfComponent* represent a use of the *Factory Method* pattern.
- *Distributed Proxy* [Silva 97] - *RPlug* and *LPlug* classes can make use of the *Distributed Proxy* pattern as a mean of supporting application partitioning.
- *Service Configurator* [Jain 96] - this pattern deals with service configuration into components. It aims at internal server component configuration, addition, change and deletion of component services, as opposed to inter-component configuration with addition, change and deletion of server components.
- *Pipes and Filters* [Buschmann 96] - this pattern again deals with the internal configuration of pipeline based components.
- *Callback* [Berczuk 95] - the pattern for the separation of assembly and processing can provide a way for component interaction. The structure of the pattern foresees the existence of some dynamic configuration allowing the connection establishment between components.
- *Broker* [Buschmann 96] - the existence of a *NameManager* class in the proposed pattern can be related in some way to the broker pattern. The *NameManager* is used as a repository of available *Names* and locations used to establish communication between components.

## Acknowledgments

The authors would like to thank Steve Berczuk for his valuable suggestions during the shepherding process.

## References

[Belissard 96] Luc Belissard and Michel Riveill. From Distributed Objects to Distributed Components: the Olan Approach. *Workshop Putting Distributed Objects to Work, ECOOP'96*, July 1996.

[Berczuk 95] S. Berczuk. A Pattern for Separating Assembly and Processing. In *Pattern Languages of Program Design*, Reading, MA: Addison-Wesley, 1995.

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

[Buschmann 96] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture: A System of Patterns*. John Wiley & Sons, 1996.

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

[Hofmeister 91] C. Hofmeister and J. Purtilo. A Framework for Dynamic Reconfiguration of Distributed Systems. In *Proceedings of the 11th International Conference on Distributed Computing Systems*, pages 560–571, 1991.

[Jain 96] Prashant Jain and Douglas Schmidt. Service Configurator: A Pattern for Dynamic Configuration and Reconfiguration of Communication Services. In *3rd Annual Pattern Languages of Programming Conference*, Allerton Park, Illinois, 1996.

[Kramer 85] J. Kramer and J. Magee. Dynamic Configuration for Distributed Systems. *IEEE Transactions on Software Engineering*, SE-11(4):424–436, April 1985.

[Magee 89] Jeff Magee, Jeff Kramer, and Morris Sloman. Constructing Distributed Systems in Conic. *IEEE Transactions on Software Engineering*, 15(6):663–675, June 1989.

[Magee 94] J. Magee, N. Dula, and J. Kramer. A Constructive Development Environment for Parallel and Distributed Programs. Technical report, Department of Computing, Imperial College, London SW7 2BZ, UK, 1994.

[Purtilo 90] J. Purtilo. The Polylith Software Toolbus. Technical Report CSD 2469, University of Maryland, 1990.

[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 and June 1994.

[Schmidt 96] Douglas Schmidt. A Family of Design Patterns for Flexibly Configuring Network Services in Distributed Systems. In *International Conference on Configurable Distributed Systems*, Annapolis, Maryland, May 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*, Brisbane, Australia, December 1995. IEEE Computer Society Press.

[Silva 96] António Rito Silva, Fiona Hayes, Francisco Mota, Nino Torres, and Pedro Santos. A Pattern Language for the Perception, Design and Implementation of Distributed Application Partitioning. *Presented at OOPSLA'96 Workshop on Methodologies for Distributed Objects*, October 1996.

[Silva 97] António Rito Silva, Francisco Assis Rosa, and Teresa Gonçalves. Distributed Proxy: A Design Pattern for Distributed Object Communication, September 1997. Submitted to the Fourth Conference on Pattern Languages of Programs, PLoP '97.

[Sousa 96] Pedro Sousa, António Rito Silva, and José Alves Marques. Naming and Identification in Distributed Systems: A Pattern for Naming Policies. *Conference on Pattern Languages of Programs (PLoP'96)*, September 1996.

[Technologies 96] IONA Technologies. Building Distributed Applications with Orbix and CORBA, 1996.

[Wegner 97] Peter Wegner. Frameworks for Active Compound Documents. Technical report, Brown University, 1997.