APPLYING PATTERNS TO BUILD A LIGHTWEIGHT MIDDLEWARE FOR EMBEDDED SYSTEMS

D. BELLEBIA
CEDRIC-CNAM
292, rue Saint-Martin
75141 Paris cedex 03
FRANCE

Facilité Informatique
62, bis rue des peupliers
92100 Boulogne Billancourt
France
djbel.consulting@free.fr

J-M. DOUIN
CEDRIC-CNAM
292, rue Saint-Martin
75141 Paris cedex 03
FRANCE
douin@cnam.fr

ABSTRACT
Today, patterns are used in several domains (distributed applications, security, software requirements, architecture…). Our purpose is double: first, to know if existing patterns can be applied in the particular domain of embedded systems middleware, second, to establish the grounding towards a patterns language for that domain.

In this paper, we describe how to design and build a lightweight middleware for embedded systems with well known patterns such as Composite, Proxy, Visitor, Observer, Publish/Subscribe, Leasing, Evictor or Configurator. The patterns we selected and implemented allow keeping the memory footprint reduced. Yet, they were relevant to address the need of creating topology views of Networked Embedded Systems (NES), to monitor and to manage them. As a result, the middleware is modular, flexible, extensible, and lightweight (< 128 kb) according to targeted embedded systems requirements.

In addition, this paper describes a concrete case study, illustrating how to select appropriate patterns to build a dedicated middleware in order to interconnect numerous small devices.

Categories and Subject Descriptors
D.3.3 [**Programming Languages**]: Language Contructs and Features – *patterns, architecture, patterns sequences, java*

General Terms
Design, Experimentation, Languages

PLoP '06, October 21–23, 2006, Portland, OR, USA.

Keywords
Patterns, Lightweight Middleware for Embedded Systems, Requirements for Embedded Systems' Middleware, Use cases, Architecture, JavaCard

1. INTRODUCTION
Embedded systems are specific-purpose computer, which are completely encapsulated by the device they control. They are ubiquitous in our everyday life, in form of many devices such as cars, medical components, clothes, personal mobile devices, sensors [26].

Advances in electronic and wireless communications (e.g. Wifi, Bluetooth, Wireless USB…) enabled the advent of Networked Embedded Systems (NES), i.e. systems comprised of interconnected devices [50].

Besides, there is now a real trend to use middleware technologies in order to interconnect NES. Indeed, middleware is a distributed software layer sitting above the network operating system and below the application layer [16]. It provides common abstractions that can be reused across different applications within a specific domain. It supports tailoring in order to meet the requirements of each application [10 p30-3]. In addition, it hides the heterogeneity of the underlying environment, it simplifies programming task, and managing distributed applications. In other words, middleware is about integration and interoperability of applications and services running on heterogeneous computing and communication devices.

However, trying to develop a middleware for embedded systems introduces many challenges we must deal with. There is no stable network infrastructure, the disconnections are inopportune and it is required that each device discovers each other in ad-hoc manner. Furthermore, embedded systems are severely constrained by scarce resources such as low battery, few memory and slow CPU [44, 50].

Since their introduction, a decade ago, patterns provide proven solutions for many software design problems. Yet they enable to

reduce the complexity and improve the flexibility of software products in particular connection management, data transfer, de-multiplexing and concurrency control [51].

What motivated our work can be summarized in three points: first, to build embedded systems' middleware with all required functionalities having non-volatile memory footprint less than 128 kilobytes; second, how patterns can be applied to build that middleware and lastly, to establish the bases towards NES pattern language.

With Regard to previous works related to this topic, two[1] middleware JINI and TAO are the most significant because they quietly present the required capabilities and their design was driven by patterns.

JINI is a Java based middleware for building services oriented solutions. It provides 'plug and play' mechanism and supports ad-hoc networking by allowing services to be spontaneously added to a network [4 p185].

TAO [27, 12] is a CORBA (Common Object Request Broker Architecture) [54] compliant middleware built over ACE (Adaptive Communication Environment). It implements entire CORBA functionalities and provides real-time QoS (Quality of Service).

However, both of them were do not fit our specification because of their memory footprint was higher than 128 kilobytes.

Our middleware is designed with patterns such as Composite, Proxy, Visitor, Observer, Publish/Subscribe, Leasing, Evictor or Configurator, etc. It is lightweight (68 kb in ROM for interconnecting JavaCards) and keeps the common qualities of patterns such as loose coupling and flexibility.

This paper is organized as follows: section 2 presents the requirements, the use cases of the middleware and an overview of a case study. Section 3, reviews design patterns applied to build this middleware. Section 4, summarizes achieved results. Section 5, concludes this paper and suggests an outlook to future work.

2. EMBEDDED SYSTEMS MIDDLEWARE

In the following subsections, we give first the NES middleware requirements that we have selected basing on [10 chapter 30, 12, 16 chapter 11 and 50]. Next, we present the use cases derived from these requirements. Finally, we introduce a case study as a particular application of our middleware.

2.1 Requirements

2.1.1 Functional requirements

Composition support: There are many breeds of embedded systems and they are ubiquitous. Therefore, the middleware is required to have capabilities that facilitate the organization and the grouping of devices as form of trees.

Event notifications: Many of embedded systems are interacting continuously with their environment through sensors and actuators. They must react accordingly when their environment changes. The middleware has to provide mechanisms to notify embedded system when an event occurs within the environment.

Reduced memory footprint: The middleware should be deployed on small devices such as mobile phones. However, those devices are very constrained by limited resources such as small memory footprint. The middleware has to take into account these scarce resources and its memory footprint must not exceed 128 kilobytes.

Reliability: In NES, it is not rare that one device relies on another to perform seamlessly its task. However, more and more of embedded systems are connected through a wireless connection. Therefore, the middleware must ensure at least a minimal reliability because network connections may be broken due to several reasons.

Asynchronous communication: The middleware supports the synchronous communication model. That it is. However, it must also support the asynchronous communication paradigm in order to allow messages to be exchanged whether or not each endpoint is operating at the same time.

Ad-hoc discovery: The middleware should implement a mechanism to allow devices to discover each other and discover services and available protocols.

2.1.2 Non-Functional requirements

Location independence: When an embedded system asks for a service, it should not worry that service resides locally or remotely. In other words, the middleware has to hide the network distribution.

Security: Embedded systems' resources are shared. Therefore, the middleware has to provide mechanism for monitoring shared resources, both in term of authentication and concurrency.

Heterogeneity support: There are many kinds of embedded systems: mobile phones, sensors, music players, cars, Personal Digital Assistant (PDA). Some of them are java-based and accept only IrDA connections. Others are running over Windows or Linux and accept Bluetooth or Wifi connections. Therefore, the middleware has to be concerned about the environment heterogeneity, which involves hardware platforms, programming languages and operating systems.

Adaptability: Two factors motivate the need of adaptability: changes in environmental conditions and users requirements. The middleware should take into account those constraints and be able to change its behavior accordingly.

Configuration: The middleware should accept dynamic configuration and reconfiguration.

Evolutions Support: The environment of NES is constantly changing. Indeed, the user's needs change over time and technologies evolve. Thus, the middleware has to provide a way to upgrade or add new functionalities.

Modularity: The middleware should be modular enough to support to tailor it.

Scalability: As already mentioned, embedded systems are omnipresent and their proliferation is still going on. Therefore, the middleware has to scale according to the number of nodes or service.

[1] The reader can refer to [50] for a complete survey of middleware for networked embedded systems.

2.2 Use cases

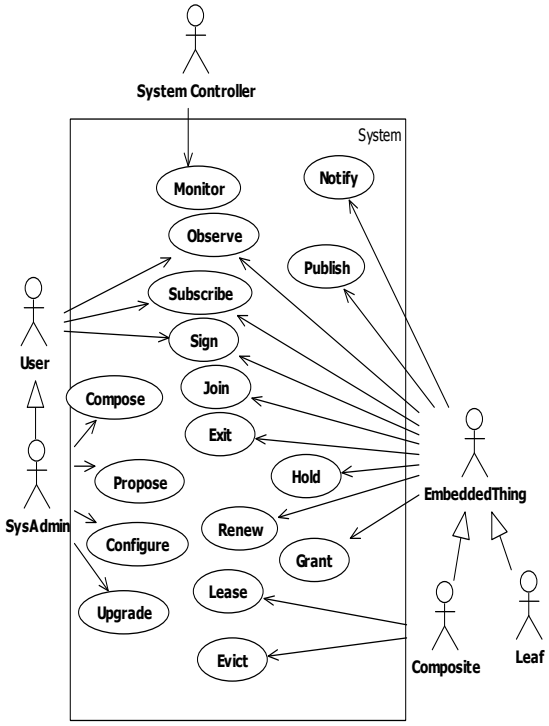


Figure 1. The middleware use cases diagram

2.2.1 Actors

EmbeddedThing is used as an abstraction of any embedded system. It can both fire and react to events and it can receive and send messages. It proposes its services to the other actors. In addition, it can connect and disconnect from the network (join and exit).

Composite is an extension of the *EmbeddedThing* actor. Therefore, it inherits the behavior of *EmbeddedThing*. It regroups several embedded systems. It accepts composition operations such as *remove* and *add children* and it is responsible for managing and providing resource references to its children.

Leaf is an extension of the EmbeddedThing actor. It inherits thus the behavior of EmbeddedThing. It is an abstraction of only one embedded system and is responsible for resource references inquiring.

User represents any user interested in the services provided by the middleware or any applications built over it. He initiates actions and waits for results. He can subscribe to events that may occur.

SysAdmin represents the application designer. He configures, composes and upgrades the application. He has to propose the implementation code corresponding to actions.

Controller is in charge of monitoring that the system is correctly performing.

2.2.2 Cases addressing functional requirements

Table 1. Use cases and functional requirements

Use case	Requirement
Compose	Composition support
Lease Evict Hold Renew	Small footprint
Observer/Notify	Events notification
Publish/Subscribe	Asynchronous communication
Join/Exit	Ad-hoc discovery
Monitor	Reliability

Compose provides to the *SysAdmin* actor all operations related to the way of regrouping embedded systems in order to compose hierarchical views, such as tree, that facilitating the traverse and the upgrade of the embedded systems.

Lease provides methods to create and manage resources by using leases. Since resources in embedded systems are limited, this use case provides to *Composite* all operations (create, delete, set duration, etc.) needed for managing its resources by using time-based leases.

Evict manages the lifecycle of a resource. It implements mechanisms to control resources usage. In addition, it allows freeing and recycling used resources.

Hold describes a way to acquire resources and leases. This use case symbolizes how an *EmbeddedThing* needs to use resources of another *EmbeddedThing*, obtains a reference of this resource and a time to live of the reference.

Renew allows renewing leases according to strategies, e.g. a time-based strategy. When an *EmbeddedThing* obtained a resource reference and if it wants to keep it, it has to renew this reference.

Observe and Notify allow to an *EmbeddedThing:* the observer, to register with another *EmbeddedThing:* the observable. The observer is notified when the observable's state is changed.

Publish and Subscribe allow to an *EmbeddedThing* to subscribe with filters to a publisher *EmbeddedThing*. The subscriber *EmbeddedThing* is notified according to its filters.

Join and Exit use cases address the mobility requirement introduced by wireless devices. It allows detection of arrival and departure of devices.

Monitor describes how to provide a minimal mechanism to ensure minimal reliability.

2.2.3 Cases addressing non-functional requirements

Table 2. Use cases and non-functional requirements

Use case	Requirement
Grant Sign	Security
Configure	Adaptability
Upgrade/Propose	Evolutions support
Configure	Configuration

Grant implements the security mechanisms such as authentication, authorization and accounting in order to control access to the *EmbeddedThing*.

Sign implements authentication mechanism to access to restricted *EmbeddedThing*.

Configure provides ability to configure and reconfigure dynamically the middleware according to different topics (type of the system, set of services, lease duration…).

Upgrade and Propose allow *SysAdmin* to modify or add new services.

2.3 Case study overview

JavaCard is a credit card-sized plastic card with an integrated micro controller chip inside. It is capable both to store information and run Java programs within the JavaCard Runtime Environment (JCRE) [16 chapter 15] it contains. Thereby, it is a perfect specimen of an embedded system. However, JavaCard suffers from it is isolated from the network. Nevertheless, when it is combined with an additional network-based device: called Card Acceptance Device (CAD), it can turn then into a node like others within the network. Although it is severely constrained by memory limitations, this does not prevent us to upload cardlets inside it and invoke them as services from the web.

At CNAM Paris, there are several JavaCards widespread among the different buildings. Most of them are embedded within iButton™ or TINI cards. In addition, there are some students possessing their own JavaCards.

It is possible using our middleware to group all these JavaCards to build new collaborative applications to provide useful services (such as authentication, courses planning, exams results, labs solutions, etc.) for both the students and the teachers.

By now, we have concretely used the middleware to offer to students a Web-Based JavaCard Development Platform (WBJDP), helping them getting more practice with this technology. The picture below depicts the architecture of the WBJDP.

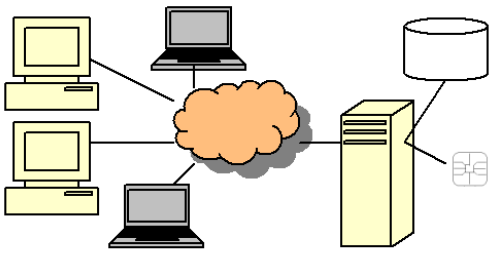


Figure 2. The WBJDP Architecture

The JavaCard contains several cardlets and it is plugged in a CAD. The CAD runs a Brazil [53] web server and is reachable via the web for other students

3. Patterns to build the middleware

According to [18], design patterns and middleware complement each other. They describe a generalized solution to a commonly occurring problem [8]. The patterns that we present here, address the architectural concerns and the requirements outlined in the previous section. In the following sections, we describe first the patterns we have selected for our middleware and how we concretely applied them (subsections 3.1 to 3.7). Next, we bring your attention on the relations between the patterns and the underlying pattern language (subsection 3.8).

3.1 Architecture

We deal first with the architectural patterns since they have governed the whole middleware design [2 p26, 8 142]. Those patterns deal with the organization of the system's elements into subsystems and components. They also specify the responsibilities of each element and the rules defining their relationship. Both the *Layers* and the *Microkernel* patterns fall into this category. The first one helps to structure systems into groups of subtasks. The second one is considered as a specialization of the *Layers* pattern.

3.1.1 Layers Pattern

When we are facing a large or complex system, we intuitively try to decompose it progressively into smaller and more manageable entities. The *Layers* pattern described in [2 p31, 8 p142] is suitable for such decomposition. It allows to structure systems, so that they can be decomposed into groups of subtasks in which each subgroup is at a particular level of abstraction. There are three ways to implement this pattern.

- *Closed layered*, one layer can only invoke its own services or those provided by the next layer down.
- *Open layered*, one layer can invoke its own services or those provided by any layer below it.
- *Layering through Inheritance*, lower layers are implemented as base classes from which higher layers inherit.

We applied this pattern to decompose the middleware architecture onto two layers (cf. figure 3): the lowest layer is dedicated to the core functionalities of the middleware while the higher is concerned with user application. Doing so, we isolated the core services such as resource management from the application's services.

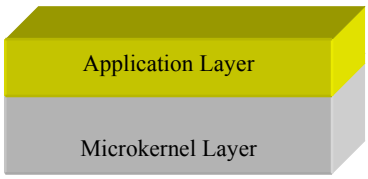


Figure 3. The middleware multi-layered architecture

3.1.2 Microkernel Pattern

The *Microkernel* pattern [2, 8] is considered as a specialized layer among the layers of the *Layers* pattern. In order to build an

adaptable system, it allows defining minimal core services of a system that can be extended at build-time with a variety of additional services.

First, it allows decomposing a whole system onto three subsystems:

- The Microkernel subsystem, which provides the minimum core set of services such as communication facilities and resource management,
- The Internal Services subsystem, which comprises the core functionalities having incidences both on the complexity and the memory footprint of the Microkernel,
- The External Services subsystem, which provides optional services, bound to the Microkernel.

Next, it surrounds the whole system with an API shared by the three subsystems, which is accessible from the outside scope of the system.

We have implemented the main-core layer according to *Microkernel* pattern. Therefore, it was possible to start the system only with a minimal core required elements such as the web server and service for the resource management. Other services are invoked just in time according to occurring events or incoming requests.

3.2 Topology management

3.2.1 Composite

By using *Composite* [1], one can recursively create with composites - i.e. containers - and leaves either complex or hierarchical structures like trees. From the user's point of view, this pattern provides a unique interface. So, the user can address in the same way leaves and containers.

For our middleware, we applied *Composite* in order to organize elements hierarchically. The following picture shows a typical composition of a Networked Embedded Systems (NES).

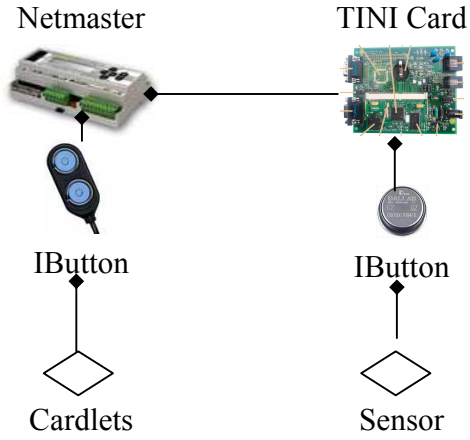


Figure 4. Overview of NES

The Netmaster [47] is hosting the IButton JavaCard containing several cardlets. The TINI card [48], in which we have plugged a sensor, is added as child to the Netmaster.

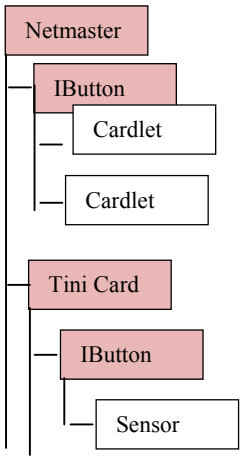


Figure 5. NES composition

The logical tree corresponding to this composition is showed above. Colored box represents a composite: the Netmaster, the TINI card and the IButton. While blank box represents leaves: cardlets and sensors.

3.2.2 Visitor

The *Visitor* [1, 7] is related to the *Composite*. Suppose, on one hand, you have an operation applicable both on leaves and on containers objects of the structure. On the other hand, suppose also that the implementation of this operation depends on the type of the object, i.e. composite or leaf. Applying the *Visitor* pattern, i.e. implement the operation in a separate subclass, it allows you to keep the structure loosely coupled with the operation. Furthermore, as this pattern is aware about the underlying *Composite* structure, you can easily adapt the operation accordingly [20].

All operations that imply the propagation through the topology structure (devices grouping, updating…) are implemented as concrete visitors.

3.3 Resource management

3.3.1 Leasing

It is commonly admitted that managing resources in distributed systems is more complicated than in the centralized systems. In deed, errors may be occurred for different reasons in distributed systems: resource corruption, network congestion or failure, remote host crashing, etc. For this purpose, the *Leasing* [4] allows managing resources by using time-based leases.

That occurs as follows:

- Step 1: a lease is associated with a resource as it is acquired.
- Step 2: if the lease is never renewed, the resource is automatically released when the time-lease expires.

When an embedded system is added as child to a composite children table (c.f. figure 4: the TINI card is added to the Netmaster), the composite creates a time-based lease for that child and sends back to the child the duration of the lease. If the child never renews the lease before its expiration, the composite delete the child from its children table.

3.3.2 Evictor

The resource management is a key concern in embedded systems [25]. Therefore, in order to cope with that, one can use the *Evictor* pattern as described in [4]. Indeed, this pattern focuses on how and when resources must be released. It allows you to apply different strategies to determine automatically and optimally, which resources should be released and when they should be released.

In order to optimize system resources, this pattern is applied to evict leases that have never been renewed and those that have been the least recently used.

3.4 Reliability

3.4.1 Heartbeat

Consider a system composed by two subsystems: a subsystem A and a subsystem B. Actually; *Heartbeat* [6 p209] is useful when A is performing an operation on behalf B or when it is used by B for providing reliability. Concretely, the subsystem A must periodically send signals to B in order to indicate that it is still alive.

Critic nodes in NES have to send "still-alive" notifications to nodes monitoring them. For example, we configured a node hosting a sensor to send periodically you a message on your cellular phone.

3.4.2 Watchdog

Watchdog [30, 8] is closely related to the *Heartbeat* pattern. It is intended to control that the whole system is processing as required. Consider once again the system we discussed above, where B is using A in order to provide reliability. In this case, *Watchdog* is commonly implemented in B and it monitors that "still-alive" messages, coming from A, are received at the right time: neither too quickly nor too slowly. Otherwise, it moves the part-whole system into a fail-safe state.

In NES, some nodes such as those hosting sensors are critic. Thereby, to provide the reliability, it is required for those systems to be checked periodically. We applied the *Watchdog* for this purpose.

3.5 Events notification

3.5.1 Observer

The intent of *Observer* [1, 7] is to keep a set of objects, i.e. the observers, up to date when the state of an object they depend on, i.e. the subject, has changed. In other words, this pattern implements a one-to-many dependency between the subject and the observers. Concretely, it allows you to attach anonymously a set of observers to a subject. Then, when the state of the subject changes it automatically invokes the callback update method of each observer.

For instance, we wanted to log any modification of a JavaCard in a HSQLDB [52] database. In this case, the database is registered as an observer with the Netmaster and the TINI card. Thus, every time their state is updated, a notification is sent to the database.

3.5.2 Publish Subscribe

Publish Subscribe (Pub/Sub) is a special case of the *Observer*. It is applicable when there are several distributed entities, which should communicate with each other and remain loosely coupled. It exposes four primitives: pub, sub, unsub and notify [36]. One can implement the *Pub/Sub* according to two different strategies: topic-based and content-based. The first strategy is fairly the same as newsgroups strategy. The system notifies the subscribers whenever a publication related topic occurs. The second strategy gives to the subscribers the ability to specify predicates over what they exactly want to be notified about [14].

We applied this pattern in order to provide asynchronous communication. Indeed, some notifications are important they must thus be delivered even if the receiver is not present. Concretely, these notifications are queued in a Distributed Hashtable (DHT) replicated in some identified stable nodes.

3.6 Network communication

3.6.1 Proxy

The intent of the *Proxy* [5 p79, 4 p199] is to provide a surrogate object to a real object. Actually, the surrogate receives client method calls and invokes the same method on the real object. The surrogate object and the real object share the same interface or super class. Hence, the client is not aware that it is calling *Proxy*'s methods rather than the methods of real object.

We have applied this pattern for remote invocation. More precisely, we created in the Netmaster and TINI nodes (from the picture 4) a proxy to the HSQLDB database. When these nodes receive a request, they automatically notify the database using this proxy.

3.6.2 Strategy

Strategy [1] let us to define a family of algorithms, encapsulate each one and make them interchangeable. In fact, it allows the algorithm to be unaware about the client using it.

In NES, there are several protocols to connect peers: Ethernet, Wifi, Bluetooth, IrDA, and Serial… Thereby, we applied *Strategy* in order to let applications to use one protocol or another according to the device network interfaces.

3.7 System and Services configuration

3.7.1 Configurator

In order to achieve more flexibility, the *Services Configuration* or *Configurator* [35, 3 p75] provides a way for decoupling the behavior of services from the moment when their implementations are configured into applications. It allows services to evolve independently from configuration issues such as concurrency model or location. In addition, it allows linking and unlinking services implementation to an application at runtime without having to modify, recompile, or statically relink the application. Besides, it centralizes the administration of the services that it configures, allowing therefore automatic initialization and termination of services.

This pattern is used to define the eviction strategy and the list of services and interceptors per embedded system. In addition, it is applied to set the nature of each system (i.e. composite or leaf) and other metadata such as listening port.

Hereafter, an example of a TINI card's configuration file is presented.

```
#
# TINI card configuration.
#
handler=main
log=5
root=.
port=1111
host=localhost

#
main.class=sunlabs.brazil.server.ChainHandler
main.handlers=thing

# feuilles
thing.class=application.interceptor.Dispatcher
thing.prefix=/thing/

#may be empty, SimpleThing, CompositeThing
_INTERFACE=CompositeThing
_CHILDREN_TABLE_SIZE=5

#Token use to split strings
_TOKENS_SEPA=,

#Interceptors list
_INTERCEPTORS_PKAG=application.interceptor.interceptors.
_INTERCEPTORS_LIST=ResourceManagment,Logging,Security,RequestToService,ServiceLocator,ServiceInvocator


#Services list
_SERVICES_PKAG=kernel.external.service.services.
_SERVICES_LIST=discovery,configuration,composition,reaction,visite,upgrade,
_SERVICES_LIST_SIZE=5

#Resource optimisation
_EVICTION_STRATEGY=LastReccentlyUse
 LEASING DEFAULT TIME=30
```

Figure 6. The TINI card configuration file

One can see that the TINI card is defined as a composite node and it can have up to five children. In addition, it has a set of interceptors and a set of services.

3.7.2 Interceptor

Interceptor as described in [3 p109, 37] is suitable when designing frameworks or middleware systems. Indeed, the intent of this pattern is to allow services to be added dynamically and triggered only when certain events occur. Therefore, applications using the frameworks or middleware can add services addressing their own functional or non-functional requirements without changing the system implementation. Besides, services provided by the system can be modified without altering its core architecture.

We applied this pattern in order to provide adaptability.

3.7.3 Chain of Responsibility

The *Chain of Responsibility* (CoR) [1, 7] pattern aims at decoupling the request sender from the receiver by interposing a chain of object handlers between them. Each handler in the chain may either handle the request; pass it on to the next handler or both. This pattern allows greater flexibility since it let handlers decide what to do with the request and users to dynamically modify or add handlers in the chain.

This pattern is applied in order to compose a chain of interceptors.

3.8 Putting all together

3.8.1 No pattern is an island

The picture below shows an overview of the patterns we applied to build the middleware. The dashed arrows are used to represent the connections between the patterns.

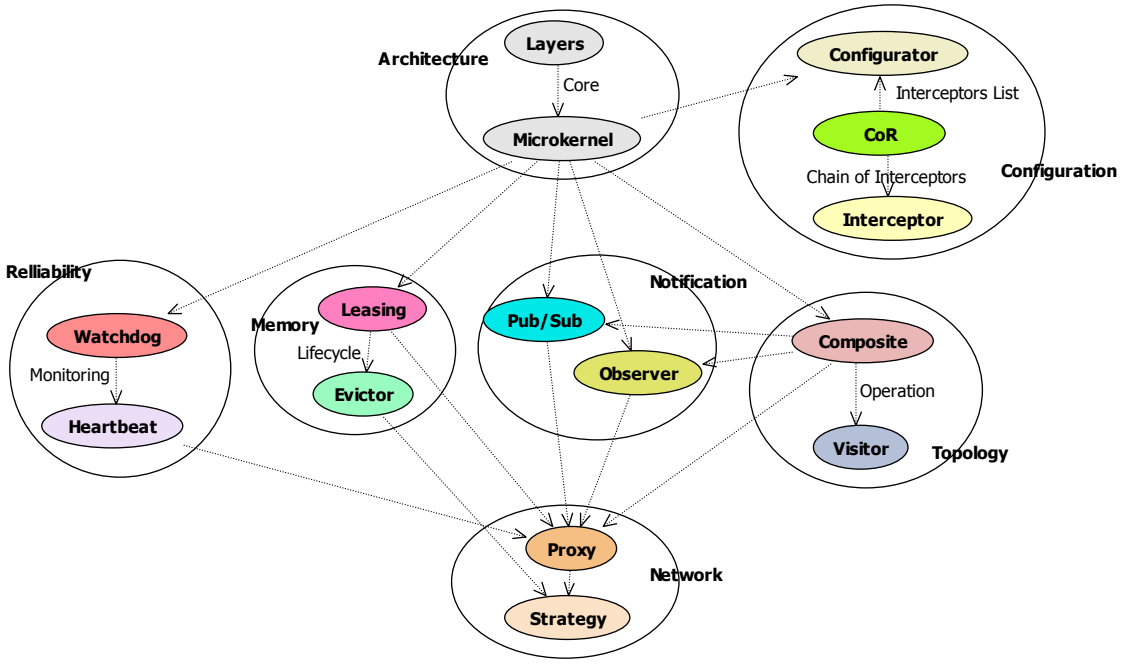


Figure 7. Patterns relationship

Layers uses *Microkernel* to design a multi-layered architecture of the middleware in order to separate core concerns (protocol implementation, or resource management) from user's concerns (how to configure applications or select the services needed).

Microkernel is a specialized layer from *Layers*. It uses: *Watchdog* for ensuring reliability, *Leasing* for memory management, *Observer* for synchronous notifications, *Pub/Sub* for asynchronous communication, *Composite* for topology management and *Configurator* for system and services configuration.

Watchdog and *Heartbeat* are both used for implementing reliability. *Watchdog* is a daemon running on *System Controller* monitoring that "still-alive" notifications are received at the right time. *Heartbeat* uses *Proxy* to send "still-alive" messages to the *Watchdog*.

Leasing and Evictor are applied in order to optimize memory usage. *Leasing* is used in order to manage memory resource with time-based leases; it uses *Proxy* when a resource holder has to renew leases with a remote resource provider. *Evictor* manages the lifecycle of the memory resource.

Observer and *Pub/Sub* are respectively applied for synchronous and asynchronous notifications. Both of them use *Proxy* for remote communications.

Composite is applied with *Visitor* in order to achieve a loosely coupling between the topology structure and operations that must be performed on it. *Composite* uses *Observer* to notify its state change and to observe changes that may occur within the network. In addition, it uses *Pub/Sub* for publishing and subscribing to events; and *Proxy* to have surrogates to remote devices.

Proxy and *Strategy* are applied to deal with network distribution. *Proxy* takes a place as a surrogate of remote devices. *Strategy* subclasses provide protocol communication implementation to *Proxy* and they implement eviction strategies, which are useful to the *Evictor*.

Configurator, Interceptor and CoR are applied to implement the adaptability. *Configurator* provides mechanisms to access and modify the system configuration. *CoR* uses Interceptor and *Configurator* to define the interceptors list in order to compose a chain of interceptors.

3.8.2 *Towards a Pattern Language for embedded systems middleware*

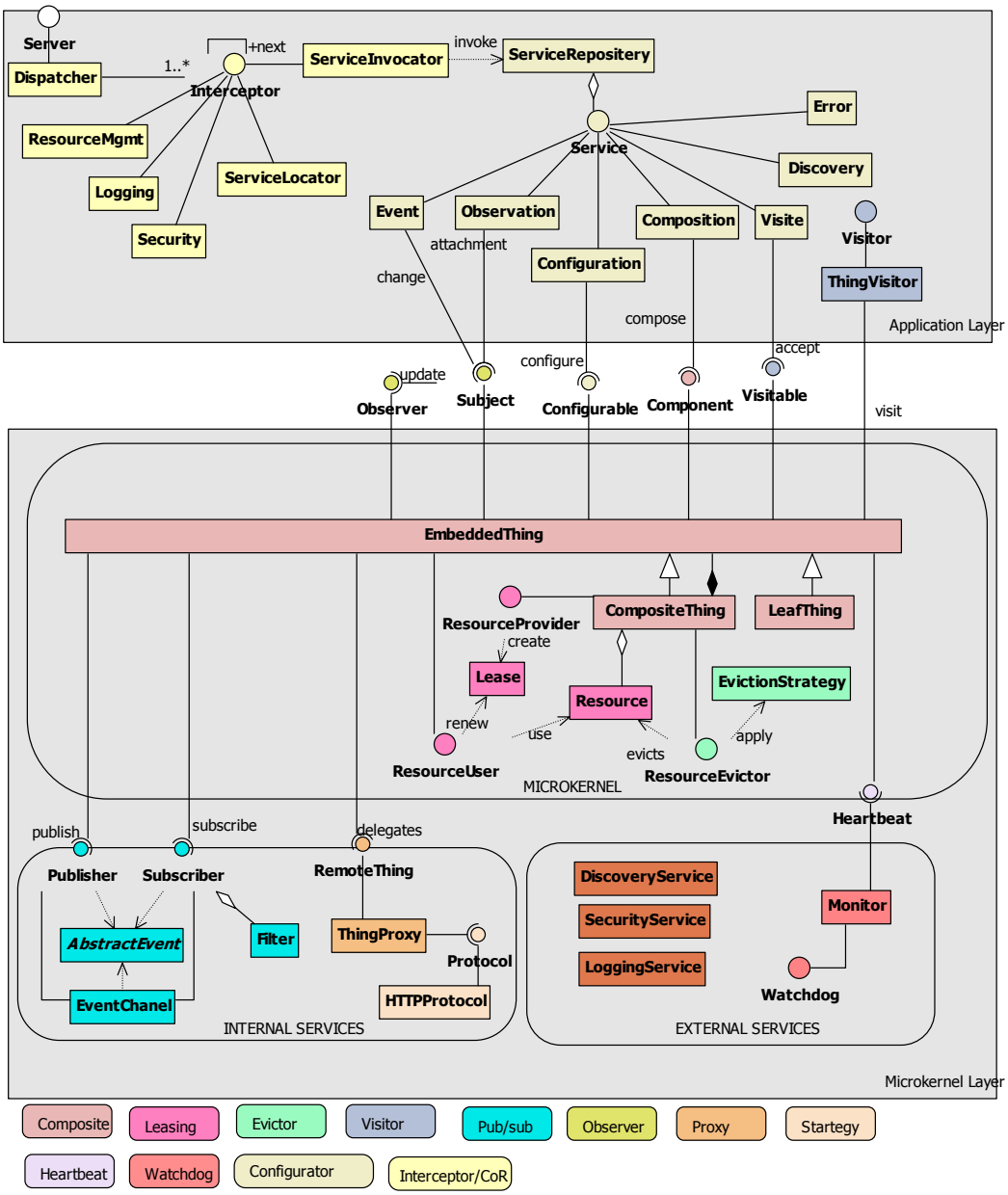


Figure 8. Towards a pattern language for embedded systems middleware

The previous picture shows which patterns we applied and how we arranged them among the middleware layers. The MICROKERNEL, within the microkernel layer, is the core of the middleware. It allows composing (*Composite*) NES topology views, resource management (*Leasing* and *Evictor*) and operations performing (*Visitor*). It uses the INTERNAL SERVICES subsystem for remote connection (*Proxy*) and protocol adapter (*Strategy*), as well as for asynchronous events subscribing and publishing (*Pub/Sub*). The EXTERNAL SERVICES provides additional services such as monitoring (*Watchdog* and *Heartbeat*). One can observe that the *CompositeThing* class can act as resource provider and resource evictor. This "schizophrenia" can also be observed with the *EmbeddedThing* class that can at the same time act as a subject

and an observer (*Observer*) or publisher and subscriber (*Pub/Sub*). In addition, the *EmbeddedThing* class, which is in fact an abstraction of any embedded system, may change its configuration according to the context (*Configurator*). At the application layer, one can notice that both the *Interceptor* and *CoR* are applied for user's requests or events handling.

4. CASE STUDY IMPLEMENTATION

4.1 A dedicated Pattern Language

Not all functionalities of the middleware are needed to the implement the WBJDP (Web-Based JavaCard Development Platform) case study. Actually, according to the requirements[2] of this application, only the memory management, the topology composition, the synchronous events notifications and the configurations support functionalities are required.

The next picture shows the dedicated Pattern Language for WBJDP.

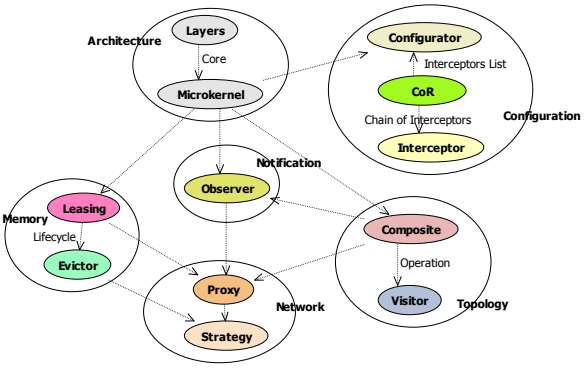


Figure 9. WBJDP dedicated Pattern Language

4.2 The Pattern Language application

Hereafter, an excerpt of the network topology tree corresponding to application of *Composite*.

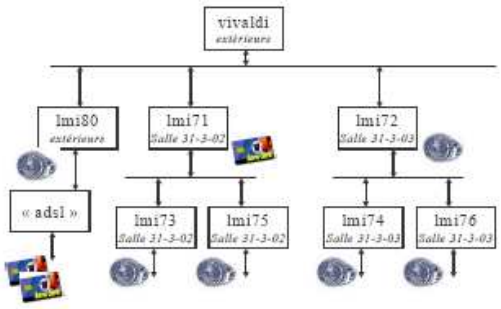
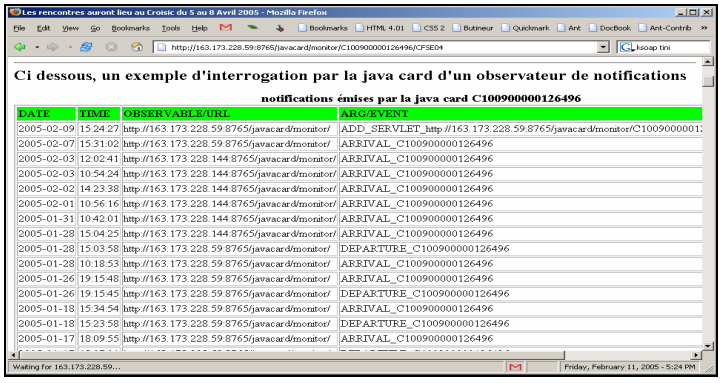


Figure 10. A tree of JavaCards (from [11])

[2] The requirements of the case study are very based on the middleware requirements. That is why we chose not detail them here.

The picture above shows how, behind each node, we attached indifferently cards or other nodes. We assume that cards can contain more than one cardlet. Each cardlet is stored in binary format.

The following picture shows a logs view that illustrating a server-side application of *Observer*.


Ci dessous, un exemple d'interrogation par la java card d'un observateur de notifications

notifications émises par la java card C100900000126496

DATE	TIME	OBSERVABLE/URL	ARG/EVENT
2005-02-09	15:24:27	http://163.173.228.59:8765/javacard/monitor/	ADD_SERVLET_http://163.173.228.59:8765/javacard/monitor/C1009000001
2005-02-07	15:31:02	http://163.173.228.59:8765/javacard/monitor/	ARRIVAL_C100900000126496
2005-02-03	12:02:41	http://163.173.228.144:8765/javacard/monitor/	ARRIVAL_C100900000126496
2005-02-03	10:54:24	http://163.173.228.144:8765/javacard/monitor/	ARRIVAL_C100900000126496
2005-02-02	14:23:38	http://163.173.228.144:8765/javacard/monitor/	ARRIVAL_C100900000126496
2005-02-01	10:56:16	http://163.173.228.144:8765/javacard/monitor/	ARRIVAL_C100900000126496
2005-01-31	10:42:01	http://163.173.228.144:8765/javacard/monitor/	ARRIVAL_C100900000126496
2005-01-28	15:04:25	http://163.173.228.144:8765/javacard/monitor/	ARRIVAL_C100900000126496
2005-01-28	15:03:58	http://163.173.228.59:8765/javacard/monitor/	DEPARTURE_C100900000126496
2005-01-28	10:18:53	http://163.173.228.59:8765/javacard/monitor/	ARRIVAL_C100900000126496
2005-01-26	19:15:48	http://163.173.228.59:8765/javacard/monitor/	ARRIVAL_C100900000126496
2005-01-26	19:15:45	http://163.173.228.59:8765/javacard/monitor/	DEPARTURE_C100900000126496
2005-01-18	15:34:54	http://163.173.228.59:8765/javacard/monitor/	ARRIVAL_C100900000126496
2005-01-18	15:23:58	http://163.173.228.59:8765/javacard/monitor/	DEPARTURE_C100900000126496
2005-01-17	18:09:55	http://163.173.228.59:8765/javacard/monitor/	ARRIVAL_C100900000126496

Figure 11. Journal's view (from [11])

The application logs in a HSQLDB database each event corresponding to the JavaCards plugging-in or plugging-off. It also logs all events related to the modification of a JavaCard. From above, you can see a generated view from the journal table.

The next figure is an applet screenshot showing the client-side application of Observer**.**

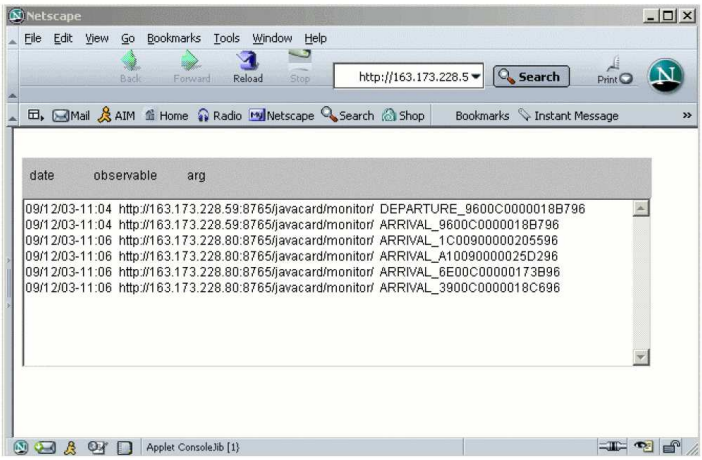


Figure 12. An applet as an observer (from [11])

When the applet starts, it registers itself with a node hosting a JavaCard as an observer. When the state of the JavaCard changes, it calls back the notify method of the applet.

The picture below shows the events propagation tree corresponding to the application of *Visitor*.

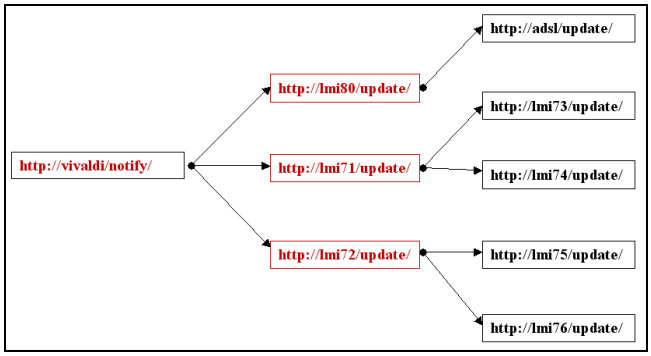


Figure 13. Request forward-propagation (from [11])

When a node receives stimuli, it propagates them forward to all its children. In the picture, above, you can see such propagation.

Hereafter, the table shows the non-volatile memory footprint of the case study.

Table 3. WBJDP memory footprint

Module	Size (kb)
Brazil Server tailored for TINI (implementing: Configurator, CoR)	43
Interceptor	4.471
Composite & Observer	5.994
Proxy	2.663
Leasing	3.727
Evictor	1.448
HTTP command	4.494

The final size of the middleware compressed jar file is about 68 kilobytes:

The middleware jar file's size is about 24 kilobytes.

The configuration file is 1 kilobyte.

Brazil-TINI Server library is 43 kilobytes.

5. Conclusion and outlook

5.1 Synopsis

Embedded systems technologies are an ever-changing field. Therefore, the challenge is to find a solution to interconnect them. That solution has to take into account their constraints (limited memory) and their constant evolution.

We stated in this paper that the middleware technologies are the suitable solution to achieve that. Also, we agreed that design patterns are a good approach for the construction of the middleware since they represent proven techniques. Furthermore, they enable loosely coupling and object oriented structure.

However, despite patterns provide solutions to almost all software design problems; they do not yet deal with specific embedded systems topic such ad-hoc networking. Yet, that did not prevent us to apply them to design JavaCards' middleware with a memory footprint less than 70 kilobytes in ROM and to establish groundings toward NES pattern language.

5.2 Work in progress

Security and ad-hoc networking represent two important topics to focus on. Even there are not yet patterns to settle the matter of ad-hoc networking; [9] describes a set of patterns to solve security issues. We are still working on how to apply those security patterns.

Also, basing on previous work [29, 41], we are investigating on how AOP (Aspect Oriented Programming) can help us to improve our middleware and reduce the overall memory footprint and best address the modularity. As for autonomic computing, we are waiting results from another project in progress in order to integrate such technologies.

Yet, we have already identified other patterns like Abstract Factory, Lazy loading, and Coordinator to implement in the near future.

6. ACKNOWLEDGMENTS

We would like to express our great gratitude to our PLoP 2006 shepherd Uwe Zdun. In addition, we thank all persons who have reviewed this document to help to improve either the correctness of the English language or technical aspects.

7. REFERENCES

[1] E. Gamma, R. Helm, R. Johnson, J. Vlissides: *Design patterns – Elements of Reusable Object-Oriented Software*, Addison-Wesley Professional, 1995

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

[3] D.C. Schmidt, M. Stal, H. Rohnert, and F. Buschmann: Pattern oriented software Architecture – Patterns for Concurrent and Networked Objects, John Wiley & Sons, 2000

[4] P. Jain, M. Kircher: Pattern oriented software Architecture – Patterns for Resource Management, John Wiley & Sons, 2004

[5] M. Grand: *Patterns in Java*, John Wiley & Sons, 1998

[6] M. Grand: *Java Enterprise Design Pattern*, John Wiley & Sons, 2001

[7] O. Maassen, S. Stelting: *Applied JAVA Patterns*, Prentice Hall, 2001

[8] B. P. Douglass: Real-Time Design Patterns – Robust Scalable Architecture for Real-Time Systems, Addison-Wesley, 2003

[9] M. Schumacher, E. Fernandez-Buglioni, D. Hybertson, F. Buschmann, P. Sommerlad: *Security Patterns – Integrating Security and Systems Engineering*, John Wiley & Sons, 2005

[10] R. Zurawski: *Embedded Systems Handbook*, Taylor & Francis, 2005

[11] N. Bonardelle : *Motifs de Conception et Intergiciel pour Systèmes embarqués*, in Conférence Française sur les Systèmes d'Exploitation (CFSE'05), Croisic, April 2005

[12] F. Eliassen, A. Andersen, G. S. Blair & co: Next Generation Middleware – Requirements, Architecture, and Prototypes, 60, The Seventh IEEE Workshop on Future Trends of Distributed Computing Systems, 1999

[13] G.S. Blair, G. Coulson, P. Robin, M. Papathomas, *An Architecture for Next Generation Middleware*, Proc. IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'98), Kluwer, September 1998.

[14] P.Triantafillou, Ioannis Aekaterinidis: Content-based publish-Subscribe over Structured P2P networks,In DEBS, 2004 http://www-serl.cs.colorado.edu/~carzanig/debs04/debs04triantafillou.pdf

[15] Microsoft: Patterns and practices – Publish/Subscribe http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnpag/html/despublishsubscribe.asp

[16] Q. H. Mahmoud: *Middleware for Communications*, John Wiley & Sons, 2004

[17] C. Britton, P. Bye: *IT Architectures and Middleware – Second Edition*, Addison-Wiley, 2004

[18] U. Zdun, M. Kircher, M. Volter: *Remoting Patterns*, IEEE Internet Computing, vol. 08, no. 6, pp. 60-68, Nov/Dec, 2004

[19] J. Rees, P. Honeyman: *Webcard: a Java Card web server,* (Proc. IFIP CARDIS 2000 http://www.citi.umich.edu/techreports/reports/citi-tr-99-3.pdf

[20] J-M. Douin, J-M. Gilliot: Collaboration patterns for networked embedded servers, in ETFA, 2003 http://www-info.enst-bretagne.fr/publication/2003-06.pdf

[21] S. Vinoski: *Chain of Responsibility*, EEE Internet Computing, vol. 6, no. 6, 2002, pp. 80–83 http://csdl.computer.org/dl/mags/ic/2002/06/w6080.pdf

[22] Sun Microsystems: *Why Jini Now?*, 1998 http://www.di.uniovi.es/~falvarez/whyjininow.pdf

[23] J. Barber: The Smart Card URL Programming Interface, Proceedings of Gemplus Developer Conference (GDC'99), Paris, France, 21-22 June 1999

[24] F. Fahrion: Embedded Ethernet Systems – Application tips for 2004, TechOnLine, 2004 http://www.techonline.com/community/ed_resource/tech_paper/36916

[25] J-M. Douin, J-M. Gilliot: A Pattern Oriented Lightweight Middleware for Smartcards, in CARDIS'04, 2004 http://www-info.enst-bretagne.fr/publication/2004/ENSTBrINFORR2004.019.pdf

[26] ERCIM: *Special Embedded Systems*, News No 52, 2003 http://www.ercim.org/publication/Ercim_News/enw52/EN52.pdfm

[27] TAO, http://www.theaceorb.com/

[28] D. Bakken: MicroQoSCORBA: A Configurable Middleware Framework for small Embedded Systems that Support Multiple Quality of Service Properties, Washington University, 2005 http://www.comp.lancs.ac.uk/computing/research/mpg/reflection/papers/MicroQoSCORBA-Lancaster-25April2005.ppt

[29] J. Hannemann and G. Kiczales: *Design Pattern Implementation in Java and AspectJ*, in OOPSLA 2002 http://www.cs.ubc.ca/labs/spl/papers/2002/oopsla02-patterns.pdf

[30] C. Webel, I. Fliege, A. Geraldy, R. Gotzhein: *Developing Reliable Systems with SDL Design Patterns and Design Components*, in ISSRE04 Workshop on Integrated-reliability with Telecommunications and UML Languages, 2004 http://www.sdl-forum.org/issre04-witul/papers/witul04_developing_reliable_systems.pdf

[31] G. Hohpe, B. Woolf: *Enterprise Integration Patterns – JMS Publish/Subscribe Example* http://www.enterpriseintegrationpatterns.com/ObserverJmsExample.html

[32] L. Aldred, Wil M.P. van der Aalst, M. Dumas, and A. H.M. ter Hofstede: *On the Notion of Coupling in Communication Middleware*, In Proceedings On the Move to Meaningful Internet Systems - 7th International Symposium on Distributed Objects and Applications (DOA), pages pp. 1015-1033, 2005

[33] STARUML, http://www.staruml.com

[34] Wikipedia: http://en.wikipedia.org/wiki/Embedded_system

[35] P. Jain, D. C. Schmidt: *Dynamically Configuring Communication Services with the Service Configurator Pattern*, in Third USENIX Conference on Object-Oriented Technologies (COOTS), 1997 http://www.cs.wustl.edu/~schmidt/PDF/O-Service-Configurator.pdf

[36] L. Fiege1, F. C. Gärtner, O. Kasten, and A. Zeidler: *Supporting Mobility in Content-Based Publish/Subscribe Middleware*, Proceedings of the 8th ACM international symposium on Modeling, analysis and simulation of wireless and mobile systems, 2005 http://lpdwww.epfl.ch/upload/documents/publications/neg--1241122820log_mobility_mw03.pdf

[37] P. Aschenbrenner, M. Förster: *The POSA Interceptor Pattern*, in Conceptual Architecture Patterns Seminar, 2003 http://wendtstud1.hpi.uni-potsdam.de/SCAP/presentations/ThePOSAInterceptorPatternNEU.pdf

[38] F .A. Rosa, A. R. Silva: *Component Configurer: A Design Pattern for Component-Based Configuration,* in Proceedings of the 2nd European Conference on Pattern Languages of Programming (EuroPLoP '97). Siemens Technical Report 120/SW1/FB. Munich, Germany: Siemens, 1997 http://francisco.assisrosa.com/pubs/europlop97-1.ps

[39] S.Baehni1, P. Th. Eugster, R. Guerraoui : *OS Support for P2P Programming: a Case for TPS,* in ICDCS 2002 (Vienna, Austria, 2002).

[40] E. A. Lee: What's Ahead for Embedded Software?, IEEE Computer Magazine, September 2000, pp. 18-26, 2000 http://www.cs.utah.edu/classes/cs6935/papers/lee.pdf

[41] C. Zhang, H-A. Jacobsen, *Refactoring Middleware with Aspects*, IEEE Transactions on Parallel and Distributed Systems, vol. 14, no. 11, pp. 1058-1073, Nov., 2003.

[42] D. Harel, M. Politi: *Modeling Reactive Systems with Statecharts*, Mcgraw-Hill, 1998

[43] M. Panahi, T. Harmon, R. Klefstad, *Adaptive Techniques for Minimizing Middleware Memory Footprint for Distributed, Real-Time, Embedded Systems*, Proceedings of the IEEE 18th Annual Workshop on Computer Communications. 18, pp. 54-58. 10.1109/CCW.2003.1240790, 2003 http://repositories.cdlib.org/postprints/656

[44] M. Kircher, C. Schwanninger *Enterprise meets Embedded*, Workshop - Reuse in constrained environments, OOPSLA 2003, Anaheim, USA, 2003 http://www.kircher-schwanninger.de/michael/publications/KircherSchwanninger.pdf

[45] R. Klefstad, M. Deshpande, C. O'Ryan , A. Corsaro, A. S Krishna, S. Rao, K. Raman *Real Time CORBA with ZEN*, University of California, 2002 http://doc.ece.uci.edu/publications/zen-performance-2002.pdf

[46] J2ME Specifications http://jcp.org/en/home/index

[47] Elsist, the Netmaster manufacturer web site, http://www.elsist.net/

[48] TINI web site, http://www.maxim-ic.com

[49] A. Corsaro, D-C. Schmidt, R. Klefstad, C. O'Ryan,

Virtual component – A design Pattern for Memory-Constrained Embedded Applications, 2002

http://www.cs.wustl.edu/~schmidt/PDF/virtual-component.pdf

[50] C.Mascolo, S.Hailes, L.Lymberopoulos, and all,

SIXTH FRAMEWORK PROGRAMME PRIORITY 2 "Information Society Technologies" – Survey of Middleware for Networked Embedded Systems, 2005

http://www.ist-runes.org/docs/deliverables/D5_01.pdf

[51] D-C. Schmidt, C. Cleeland, Applying a pattern language to Develop Extensible ORB Middleware,2000

[52] http://www.cs.wustl.edu/~schmidt/PDF/ORB-patterns.pdf

http://www.hsqldb.org

[53] http://www.experimentalstuff.com/Technologies/Brazil/index.html

[54] CORBA, http://www.corba.org/