Bodyguard

Article · January 1997

CITATIONS
4

READS
484

2 authors, including:

Alejandra Garrido
CONICET: National Scientific and Technical Research Council
87 PUBLICATIONS **1,312** CITATIONS

SEE PROFILE

Bodyguard: A Pattern for Object Distribution

Fernando Das Neves and Alejandra Garrido
LIFIA. Laboratorio de Investigación y Formación en Informática Avanzada.
Dto. de Informática, Fac. Cs. Exactas, Universidad Nacional de La Plata.
50 y 115 1er piso, (1900) La Plata, Buenos Aires, Argentina.
e-mail: [babel17, garrido]@sol.info.unlp.edu.ar

Intent:

Allow to share objects and control their access in a distributed environment without system level support for distributed objects. The Bodyguard is an object behavioral pattern that simplifies the management of object sharing over a network. It provides message dispatching validation and assignment of access rights to objects in non-local environments, to prevent the incorrect access to an object in colaborative applications.

Motivation:

To illustrate the Bodyguard Pattern, consider a distributed environment for building and browsing the common knowledge of a team about World Wide Web pages, like the one depicted in figure 1. Pages are collected as team members visit WWW nodes by using WWW browsers. Those browsers capture pages, create "WWW Page" objects, and pass these objects to a repository running in a different host, that tentatively classifies the pages.

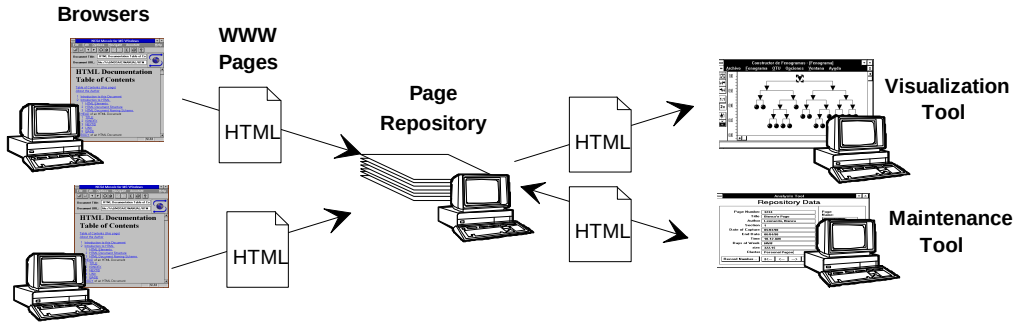

Figure 1. Some tools working together, probably on different computers, over a common repository of WWW pages. Some of the tools can add pages to the repository (like browsers), others can inspect them (Visualization tool), and others can do anything (Maintenance tool).

The repository is simultaneously accessed by many tools. Let us focus on the object sharing among the repository and the tools. It can be consulted by two kind of tools: those that are used to analyze and query the repository content, and those who are in charge of "cleaning" and organizing the repository, by deleting irrelevant pages and reclassifying some others which were mistakenly classified by the repository.

The forces fighting behind this problem are:

- we need to share objects among a server and many accessing and updating tools, marshalling messages to support the communication;
- any object which is distributed should be prevented from improper access depending on the context usage, so there is a need to control access rights. In this example, querying and visualization tools should not be allowed to delete entries in the repository;
- sometimes the operating system does not provide support for distributed objects (like CORBA), or you do not want to pay the price in resources for the flexible and wide range of services those systems provide,

when you do not need or do not take advantage of all of them. There are applications with limited needs for object distribution, running in platforms (e.g., low-end Unix boxes) where the supposition of CORBA availability conspires against the portability of the application.

A preliminar solution could be achieved by exchanging data with standard operating system mechanisms like RPC and message-passing, but there is an impedance mismatch between applications which benefit from objects throughout their design and the sharing of these objects using those low level mechanisms.

What we need is to decouple the object sharing from communication and from access control. In the example, the repository would only be in charge of maintaining pages and reply queries for the pages it contains. Another object, that we call Transporter, would be in charge of message marshaling and communication between different servers. We can use proxies to maintain surrogate objects in remote sites. A special object, called Bodyguard, would be in charge of maintaing and controlling the access rights to the objects for different message categories. These objects and their relationships conform the Bodyguard pattern.

The Bodyguard pattern is an alternative to get stuck by platform-dependent data-transport system calls and the unclear separation of access control from data transport. It allows decoupling of access control from transport mechanisms, dynamic change of access rights and method dispatching.

The following two figures show how the WWW page sharing scheme looks like, when designed following the Bodyguard pattern.

The first figure shows the scheme from the point of view of a maintenance tool, asking to delete a page:

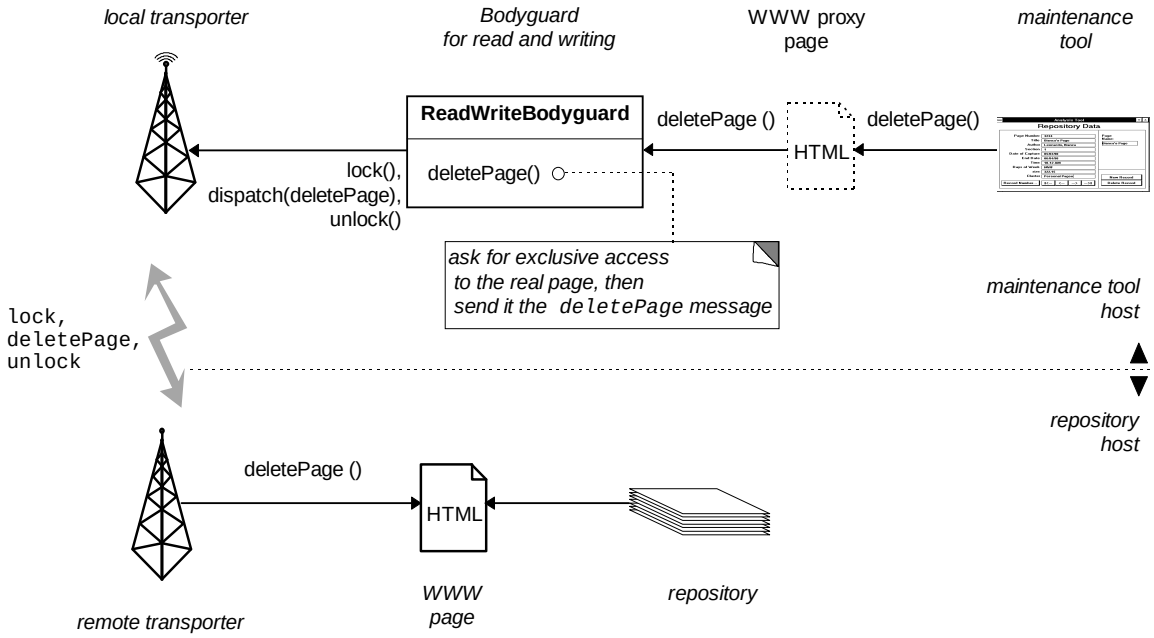

Figure 2. The page sharing scheme for the Mantainance tool. The tools sends messages to the proxy page, that in turn delegates the message to the bodyguard to check it. Since this bodyguard allows writing operations, the message deletePage() is sent to the real page. In this example, the repository is not notified of the operations.

The second figure shows how, with the same scheme, the visualization tool is not allowed to delete the page:

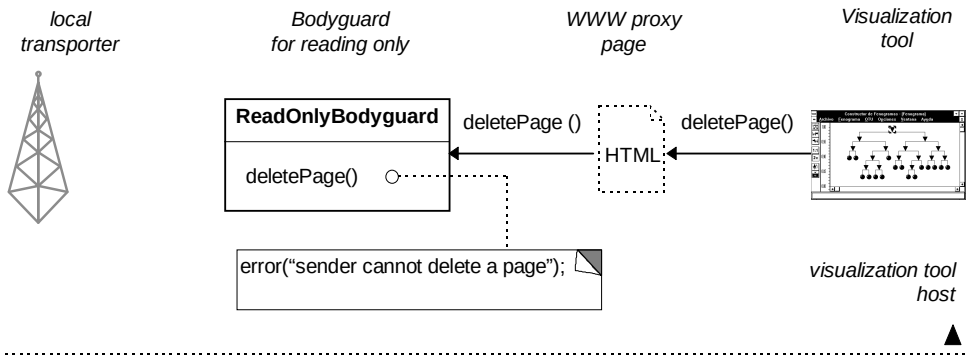

Figure 3. The page sharing scheme instantiated for the Visualization tool, showing how ReadOnlyBodyguard (a kind of Bodyguard that does not allows modification messages) denies the request to delete a page. Please note that since the bodyguard denies the operation, there is no communication with its transporter and between transporters.

The essencial components of the Bodyguard pattern are: the object to be shared, the object that control the access to that object (Bodyguard) and the one that is in charge of the communication among hosts (Transporter). There is one transporter in every host. Each transporter may know all objects that can be shared, and so they can also assure exclusive access to them. Proxies redirects messages to their bodyguards, and optionally packs the messages as objects. If the messages are allowed according to the Bodyguard's access rights, then the local transporter sends the request to the remote transporter, which in turn unpacks the request and send the message to the real object that the proxy stands for.

Applicability:

Use the Bodyguard Pattern when you:

- Have control over the the construction of applications that access shared objects.
- Need to share objects in a distributed environment with a common object implementation, but the platform lacks system-level support for distributed objects (CORBA or the like), and authentication control is not mandatory.
- Need to synchronize and pool message passing to shared objects.
- Need fine grained control of access restrictions, depending on who requests to share the object.
- Need dynamic change of access rights.
- Need identity checking and assigment of access rights to objects, in order to prevent an incorrect behaviour from objects of an application, but not to guard against security violations from those objects.

Do not use the Bodyguard Pattern when:

- Your application demands strict authentication of object identity in order to grant access rights and to prevent unauthorized access to restricted information.
- You need support for complex distributed object services, like licencing, externalization, and querying (as with CORBAServices).
- You have a very heterogeneous environment, with different object models (like having different inheritance and exception propagation models), and with different network protocols.

Structure:

Structure of the Bodyguard Pattern is illustrated in the following OMT diagram:

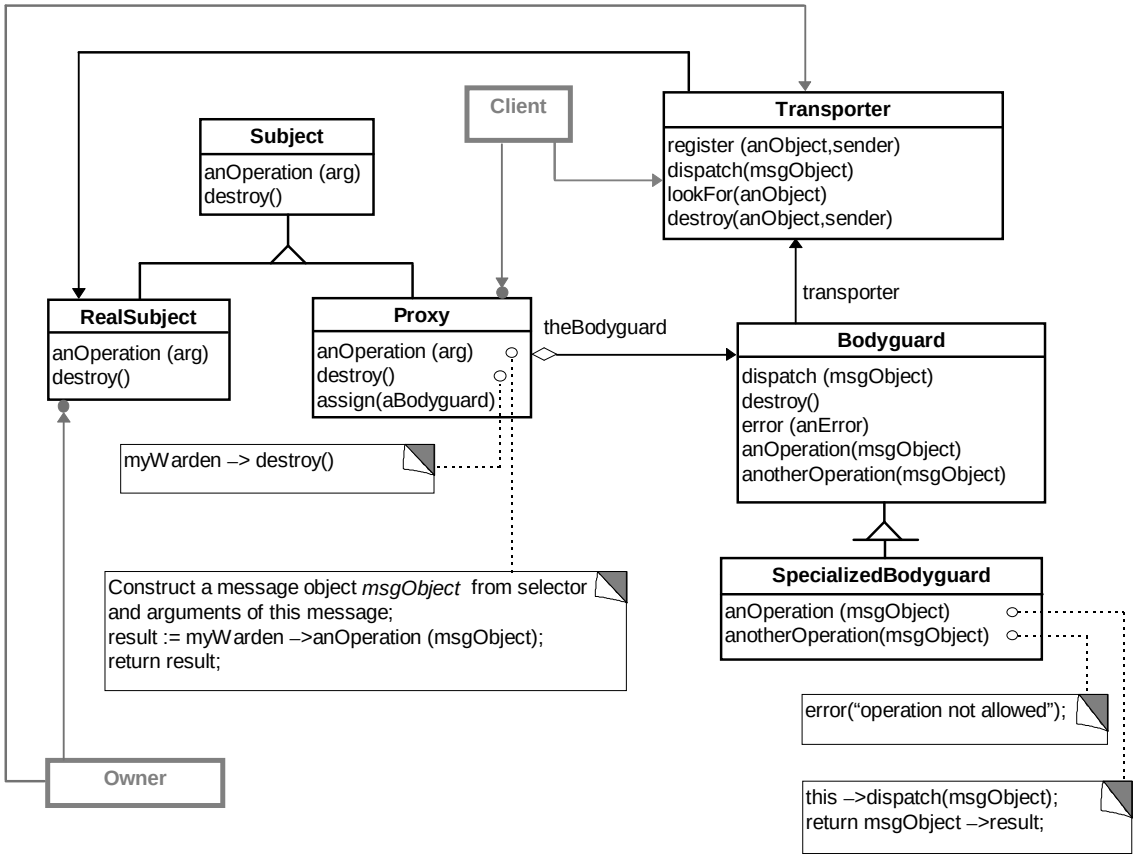

Figure 4. Structure of the Bodyguard pattern.

Coming back to the examples and looking at the figures 2, 3 and the pattern structure, WWW pages corresponds to RealSubjects, WWW proxy pages are Proxies, and ReadWriteBodyguard and ReadOnlyBodyguard are SpecializedBodyguards. When a tool needs a page, it asks for the page to the repository. If the repository has the desired page, it answers affirmatively, and a WWW Proxy Page is created by the transporter at the tool location. Depending on the access rights assigned to the new proxy by the repository, an instance of ReadWriteBodyguard or ReadOnlyBodyguard is created and designated as the bodyguard of the new page proxy. A ProxyPage class would provide a template for message dispatching. Instances of a Transporter class works as an external interface for communication I/O, marshals messages and their parameters, and registers objects that can be shared or their remote images (proxies plus Bodyguards). Classes ReadOnlyBodyguard and ReadWriteBodyguard would provide conformance checking for message dispatching, like refusing to delete a page in ReadOnlyBodyguard.

Participants:

Real Subject

- Defines the object meant to be shared.

Transporter

- Controls the communication I/O, performing the marshalling that is necessary in order to encode objects and messages to be sent over the communication channel. In fact we will have a transporter in each remote machine, so that marshalling will be encapsulated among the different instances, making it transparent to the rest of the application.
- Registers the associations of shared objects and remote proxies.
- Communicates with other transporters to optionally implement atomic operations over shared objects, like locking and total reference count of an object.
- Creates proxies of shared objects upon request to access them.

Bodyguard

- Works as a mediator between proxies and transporters, by sending method dispatching requests, notifying object destruction and optionally handling error notifications. Bodyguard is an abstract class, SpecializedBodyguard is the one that implements concrete Bodyguards.
- Controls the access to the real subject by the client, asking the transporter to carry the message out to the real subject, when permitted.
- Specifies the protocol that will be implemented by SpecializedBodyguards.

SpecializedBodyguard

- Implements methods to provide conformance checking for message dispatching both in access rights restrictions and low level synchronization and exclusion. There will be a subclass of SpecializedBodyguard for each kind or group of operations that needs a particular access right.

Proxy

- Acts as a surrogate to the real subject .
- Provides an interface identical to RealSubject.
- Maintains a reference to a Bodyguard for real subject and delegates every request to it.

Collaborations:

- An object that is intended to be shared will be passed as parameter of the message *register* sent to the transporter in order to become broadly known (see interaction diagram 4.1). The transporter then records the object. The sender of the request to register is designated as the owner of the object and is allowed to destroy the object.
- The first access to a remote object is performed asking for it to the local transporter. The transporter will search the object in the network. If it finds the object, then it will create a local bodyguard and proxy to control de access to the remote object (see interaction diagram 4.2).
- In following accesses, when a client needs to send a request to the remote object, it will interact with the proxy, that in turn delegates the request to the Bodyguard. The later will check the validity of the request and when appropiate, communicates with its local transporter, that is in charge of sending the request to the one in the remote host. The remote transporter will send the request to the remote object, and then will send back the proper result (see interaction diagram 4.3).

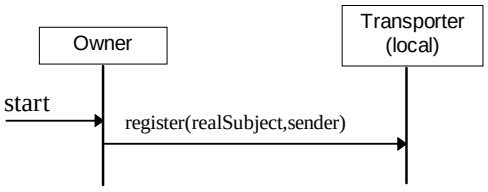

Figure 4.1: Interaction Diagram for "Object Registration"

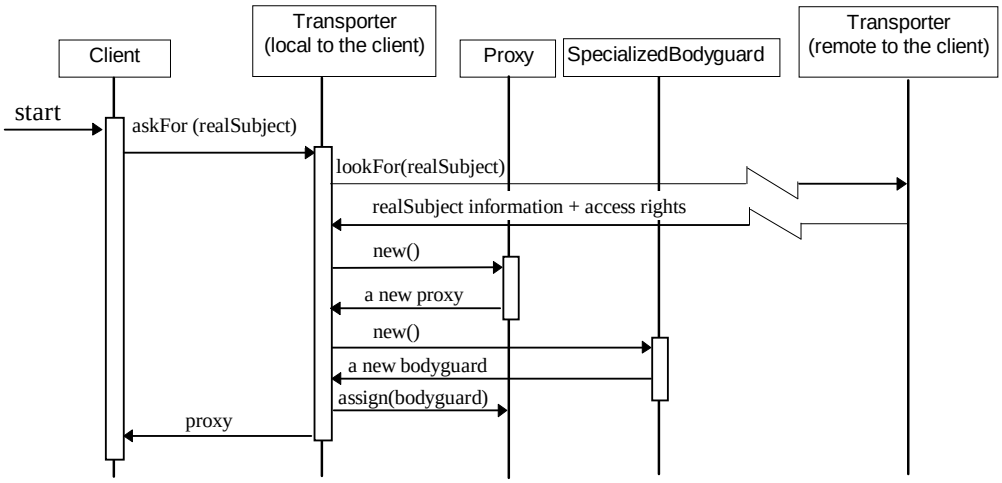

Figure 4.2: Interaction Diagram for "Remote Object Access"

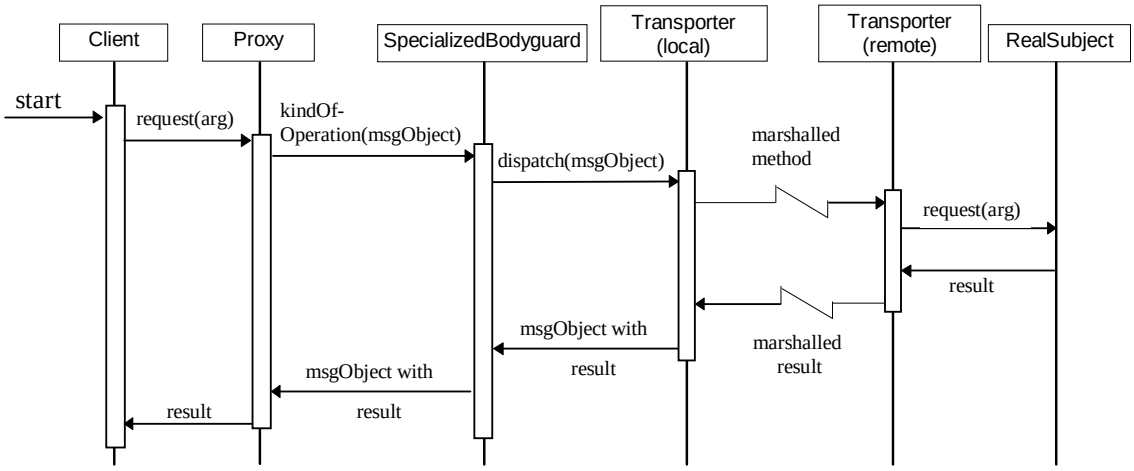

Figure 4.3: Interaction Diagram for "Remote Method Dispatching"

Consequences:

- It decouples access control from transport mechanisms, allowing the dynamic change of access rights. This decoupling allows the separation of data transport from the definition of different types of access rights over the objects meant to be shared, and from synchronization control. It also isolates access restriction mechanism, because Bodyguards are only visible for proxies and not for external objects.
- It trades generality for simplicity. In situations with restricted requisites and limited resources, the Bodyguard pattern offers an object sharing scheme that is easier to understand and implement than a full, system-level object sharing system.
- Bodyguards may also be used to record debugging or tracing information as completed messages, queued messages, etc.
- As a drawback, the pattern implies augmenting the levels of indirection to reach an object. If commnication speed is more important than access validation, then a scheme where objects talk directly to each other should be implemented.

Variants:

Rights Assignment by Class vs. by Identity: At the moment a transporter receives a request to access a shared object, and the availability of the object is confirmed, the transporter has to decide which access

rights to assign and create the corresponding Bodyguard. Access rights can be divided in general categories, and one of the categories is assigned deciding upon the class the client belongs to and the kind of object it is requesting, or upon the class of the object to share plus the object identity of the client. Assignment by class is best performed by the transporters; assignment by identity is best performed by collaboration between the transporter and the object owner.

General Bodyguards vs. Protocol Bodyguards: The number of Bodyguard subclasses that need to be implemented heavily depends on how specific the control should be and the different kind of access rights combinations. Bodyguards implement access control by the definition of a set of operations that express the general categories in which methods can be classified. Every operation stands for a kind of access, like reading, writting and deleteting. Proxies send requests for method dispatching by calling the proper type of operation in its Bodyguard for the category the method belongs to.

Granularity of the set of operations vary depending on the security enforcement needed by the classes whose instances will be shared. It ranges from checking only reading and writting permissions, to the extreme case in which every operation in a proxy has the same operation in a Bodyguard. In that case, operations in the Bodyguard are implemented knowing the semantic of the operations they will control in a particular proxy, at the cost of having to code a particular Bodyguard for every protocol to be checked. It should be noted that it is not always necessary to implement a Bodyguard subclass for every class whose objects are going to be shared, but instead a Bodyguard can watch for a whole hierarchy. For instance, in Smalltalk a single CollectionBodyguard class can be used to control the Collection hierarchy for one access right, as the protocol is completely defined in the abstract class *Collection*.

Implementation:

Message Marshaling: The way proxies redirect a message invocation to their Bodyguards for checking and dispatching, and the way the transporter expresses that remote invocation heavily depends on the language.

Languages that keep metainformation about classes make easier to know which method we are trying to remotely dispatch, and to transmit that knowledge to the remote site. For instance, messages in Smalltalk are objects in themselves, and for any message it is possible to obtain a symbol describing the *message selector* (message header), and conversely, having a selector and a class it is possible to find the method that matches the selector. This means that there exists an easy path from a message to a symbol and back, which the transporter can use to marshal the message.

On the opposite side reside languages that discard class and object metainformation at compile time. For those languages, a message object should be explicitly created by a Bodyguard every time one of its methods is called. Bodyguard checks access rights, the transporter at the origin marshals the parameter and the transporter at the destiny "decodifies" the message invocation and calls the real object.

Sometimes a back end is implemented to provide the metainformation the language discards and to add accessory information for garbage collection and atomic operations. A language independent implementation of a back end for supporting distributed objects on languages with no metaclass protocol can be found in *Amadeus* [McHugh93].

Implementation of Participants:

Proxies: A detailed explanation of implementations of proxies for remote access can be found in [Gamma94] as *Remote Proxies*.

Transporter: They manage the shared objects residing on every host. A Transporter is composed by two concurrent units: one is in charge of marshaling the arguments of a message for a remote message invocation, and deciding for each argument to pass-by-proxy or pass-by-copy [McCullough87]. There are values that cannot be passed by proxy, like numbers on hybrid languages like C++. The second unit manages the communication in the opposite direction, from a remote host to a local object.

Transporter implementation can vary from a concurrent process to an independent daemon, depending on the complexity and flexibility of the services provided. Nevertherless the context of the problem that Bodyguard addresses and all of the implementations up to date suggest using one process or many collaborative processes. It can be implemented using the Reactor pattern [Schmidt95]. Reactor manages three different handlers: *Request Acceptor* is created in every host to receive incoming request for sharing an object which resides on that host. It checks that the object resides there and sends an "OK to share", the object ID on

the host and an access right. At that time a *Message Handler* is created by the Request Acceptor for receiving marshaled messages from a given host, if not such an acceptor already exists.

At the host where the request for a remote object has begun, an *OK_to_share Acceptor* waits for the acknowledge to create the remote proxy and the associated Bodyguard, depending on the rights the origin Bodyguard has decided. After the acknowledge is received, the *OK_to_share Acceptor* is eliminated from its reactor.

Object Creation: Transporters must create proxies upon request to access shared objects. The strategy to create new proxies depends on the implementation; if the Transporter can access all instance variables of the objects, and if objects has a copy operator, then it can clone new objects from prototype objects. If the Transporter is able to know all classes and it metaclass information, like in Distributed Smalltalk [Bennet87], then it can directly create the instances from the class information. In the most general case, it can use a separate hierarchy of Abstract Factories [Gamma95] .

Access rights: If access rights are going to dynamically change, it could be the case that some object which references a shared object needs to know whether that object is local or remote, shared or private, and which are the remote object's current access rights before performing an operation. This behavior could be achieved even before the object is shared, by implementing a default protocol in the Subject class, and redefining it in its Proxy subclass. Methods like *isShared()* or *isRemote()* can be implemented in Subject to give a default answer when the queried action has not yet been performed (i. e., answering false for *isShared()* and is*Remote()*). At the local site, messages are answered without knowledge from the object by being inherited; at the remote site, messages are answered by the proxy, asking the Bodyguard when necessary.

Object destruction and garbage collection: Transporters have to be notified of object creation and destruction in order to keep the list of shared objects up to date. This is not a problem in languages with explicit object destruction, since at the moment the object destructor is called, it is trapped by the proxy. On those languages with garbage collection, either a hook should be triggered at the moment of object destruction (like the *finalize* method in Smalltalk-80 [ParcPlace94]) or the garbage collector must be modified to check for local and remote references, as is the case of Distributed Smalltalk [Bennett87]. Object destruction is one of the access rights the Bodyguard grants or prohibits.

Related Patterns:

The **Proxy** pattern [Gamma94] discusses some issues related to remote proxies and housekeeping tasks. It is evident that we need proxies in order to provide a local representative for an object in a different address space, and to count references to the real object. It also discusses about "protection proxies" as those which also provide access control. Nevertheless, the Proxy pattern is not enough for a distributed environment, because there is not explicit description of how to achieve low level synchronization mechanisms. The Bodyguard patterns uses the Proxy pattern but enriches the structure with additional control (Bodyguard) and transport related (Transporter) objects.

The **Reactor** and **Acceptor** patterns [Schmidt95] are also related with the way in which the Transporters works. Reactor and Acceptor interact together in order to receive login requests and create handlers for them. Transporter perform similar functions each time a remote shared request is made, creating proxies and Bodyguards in order to control their access. Transporter and Reactor both pull the incoming messages and maintain the association between objects and their proxies, or handlers and their callbacks and clients, respectively. Nevertheless, many differences may be found in the intents of both patterns, because in spite of both manage with distributed environments, Reactor is intended for a client/server architecture whereas Bodyguard pattern aims to a peer-to-peer environment.

The Bodyguard Pattern is strongly related to the **Broker** Pattern [Stal95]. In some sense, Bodyguard is a specialization of the structure of the Broker (a kind of *Indirect Broker System* in the Broker taxonomy). Transporters take the place of brokers and the task of message unmarshaling of server-side proxies, and Proxy class in the Bodyguard stands for client-side proxies in the Broker. The main difference between Bodyguard and Broker is that Broker is a high-level pattern: it describes the interaction among subsystems like transport subsystem (client proxy, broker, bridge) and language-mapping (server-side broker, object interfaces). As such, the typical application of the broker pattern involves large-scale and possibly heterogeneous networks, with total independence from system-specific details as a mandatory requierement. There is also a big distance from the Broker pattern to a concrete implementation like CORBA; a large amount of details must be filled before being able to produce an implementation.

Bodyguard is a middle-level pattern, in the sense that it expresses a design closer to the actual implementations than the broker does. As can be seen in the *Known Uses* section, the environment of a Bodyguard instantiation is more restricted than the one of a Broker. It involves cooperative applications, ussualy developed as a set, that use access control to ensure global invariantes in shared objects, rather than ensuring security. It the example of the *Motivation* section, bodyguards check the readers/writer invariant is not violated by any tool.

When transporters need to create proxies and bodyguards, **Abstract Factories** [Gamma95] are the most general scheme to manage it, although as described in the *Implementation* and *Known Uses* sections, it is often the case that simpler schemes, although less general, are used for the sake of performance.

Known Uses:

[Dollimore93] shows an implementation of shared objects with access restriction in Eiffel. Objects are remotely represented by proxies, which are associated to a Bodyguard known as a *filter*. Proxies and filters build together a PAC (Private Access Channel), which watches for unauthorized methods. Access rights can dynamically change by replacing the filter. Proxy manages remote invocation of methods by RPC and does not know the remote object directly, but instead they know the filter associated to the object.

[McCullough87] implements a forwarding mechanism for object sharing in Smalltalk-80, composed by proxies at the remote side and transporters (known here as TransporterRooms) on both sides. An associated PolicyMaker is used to decide to pass message parameters by-proxy or by-value. Object creation is directly managed by TransporterRooms, that works together with a distributed garbage collector for managing the list of available share objects. No access control is provided. TransporterRooms are implemented as concurrent processes.

Distributed Smalltalk (DS) [Bennett87] melts the Bodyguard role with the Transporter role (collectively known in DS as RemoteObjectTable). In DS it is possible to allow or inhibit access to an object, and also to designate an object as *agent* of another objects. Agents can process messages or redirect them to another object, in a manner similar to the way Bodyguards filter messages and redirect them. DS has no way of giving different access rights to proxies; access control is allowed only at the host where the original object resides. Local objects are created by the RemoteObjectTable, and are directly managed by the messageProcess, which works in coordination with a distributed garbage collector. RemoteObjectTable is implemented by three concurrent processes.

References:

[Bennett87] Bennett, J. *"The Design an Implementation of Distributed Smalltalk"*, in proceedings of OOPSLA'87, Conference on Object Oriented Programming Systems, Languages and Applications. ACM Press, 1987.

[Dollimore93] Dollimore, J. and Wang, X. *"The Private Access Channel: A Securiy mechanism for Shared Distributed Objects"*, in proceedings of TOOLS 10, Technology of Object-Oriented Languages and Systems. Prentice Hall, 1993.

[Gamma94] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *"Design Patterns: Elements of Reusable Object-Oriented Software"*, Addison Wesley, 1994.

[McCullough87] McCullough, P. *"Transparent Forwarding: First Steps"*, in proceedings of OOPSLA'87, Conference on Object Oriented Programming Systems, Languages and Applications. ACM Press, 1987.

[McHugh93] McHugh, C. and Cahill, V. *"Eiffel**: An implementation of Eiffel on Amadeus, a Persistent, Distributed Applications Support Environment"*, in proceedings of TOOLS 10, Technology of Object-Oriented Languages and Systems. Prentice Hall, 1993.

[ParcPlace94] *"Visual Works Object Reference"*, ParcPlace Systems, Inc., 1994.

[Schmidt95] Schmidt, D. *"Reactor: An Object Behavioral Pattern for Concurrent Event Demultiplexing and Dispatching"*, in Proceedings PLoP'94, 1st. Annual Conference on the Pattern Languages of Programs, 1994.

[Stal95] Stal, M. *"The Broker Architectural Framework"*. Workshop on Concurrent, Parallel and Distributed Patterns of Objects Oriented Programming, held at OOPSLA'95.