Warden: 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.
C.C. 11, (1900) La Plata, Buenos Aires, Argentina.
e-mail: [babel17, garrido]@ada.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 Warden is an object behavioral pattern that simplifies the management of object sharing over a network, and provides message dispatching conformance and assignment of access rights in non-local environments, to prevent the incorrect access to an object in colaborative applications.

Motivation:

To illustrate the Warden 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, creates “WWW Page” objects, and pass these objects to an application running in a different host. This appliction acts like a common page repository that stores and tentatively classifies the pages.

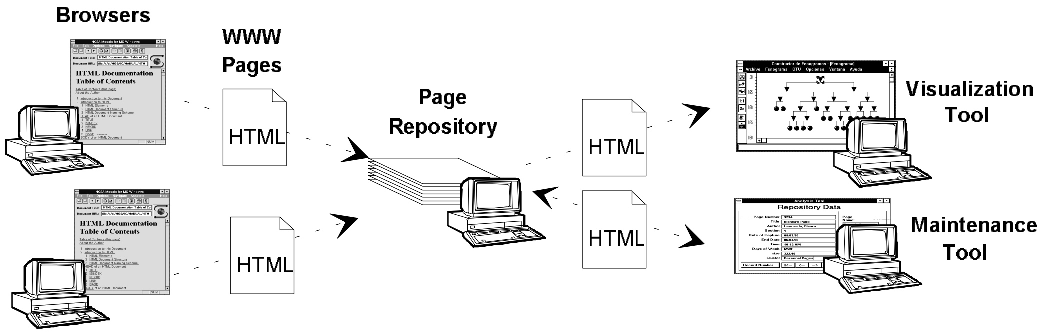

Figure 1. Some tools working together, probably on different computers, over a common repository of WWW pages. Some of the tools can add pages to the repository (like browsers), others can inspect them (Visualization tool), and others can do anything (Maintenance tool).

The repository is simultaneously accessed by many tools. For now ignore the travel of pages from the browsers to the repository, and focus on the object sharing among the repository and the consulting tools. The repository can be consulted by two kind of tools: those which 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. A preliminar solution could be achieve 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. A secondary concern is access control: any object which is distributed should be prevented from improper access depending on the context usage. In this example, querying and visualization tools should not be allowed to delete entries in the repository, but a maintenance tool can freely delete the repository entries.

Furthermore, while the maintenance tool is reorganizing the repository, maybe some page clusters should be locked, so it needs to disallow the visualization and query tools to access pages which belong to those clusters.

Another approach would be using a system-level object-sharing scheme, like CORBA. The first problem with this approach is that maybe you do not have such a scheme available, since up to date it is not part of most operating systems. The second concern is that those schemes usually provide support for a wide range of services, like platform and network independence, authentication, etc. The price for such flexibility is a high demand on system resources, even if you do not take advantage of all the services, and a very sloppy learning curve.

A better solution would be to decouple 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. Proxies may maintain a surrogate in remote sites. A special object, called Warden, would be in charge of maintaing and controlling the access rights for different message categories. These objects and their relationships conform the Warden pattern.

The Warden 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 diagrams in OMT notation illustrate an example of a WWW page sharing scheme designed according to the Warden Pattern. In the example, not all classes and methods in the Structure section are shown to simplify the explanation; for instance, AccessReadWrite and AccessReadOnly are subclasses of a more general Warden class. See figure 4 for the complete structure.

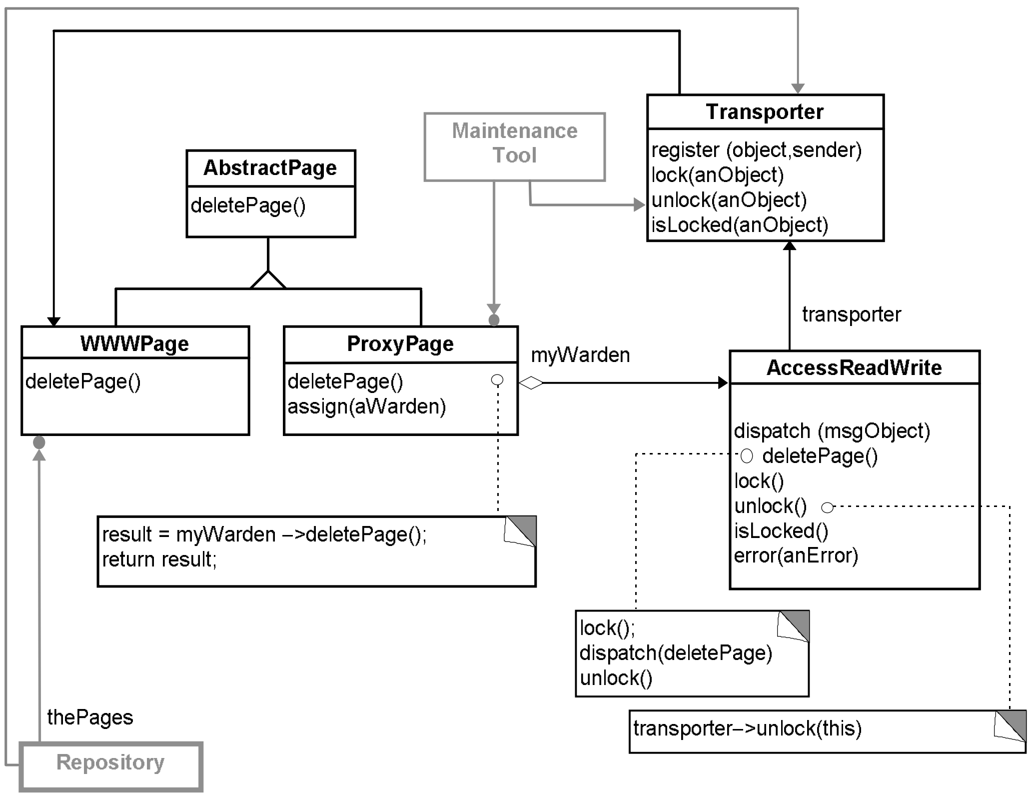

Figure 2. Warden pattern instantiated for the Mantainance tool, showing how ProxyPage (Proxy) and AccessReadWrite (Warden) interact to assure exclusive access for deleting a page. Plase note that the Repository, instances of WWWPage and an instance of Transporter reside in one host, while the Maintenance Tool, proxes, their wardens, and a different instance of Transporter are located in another host.

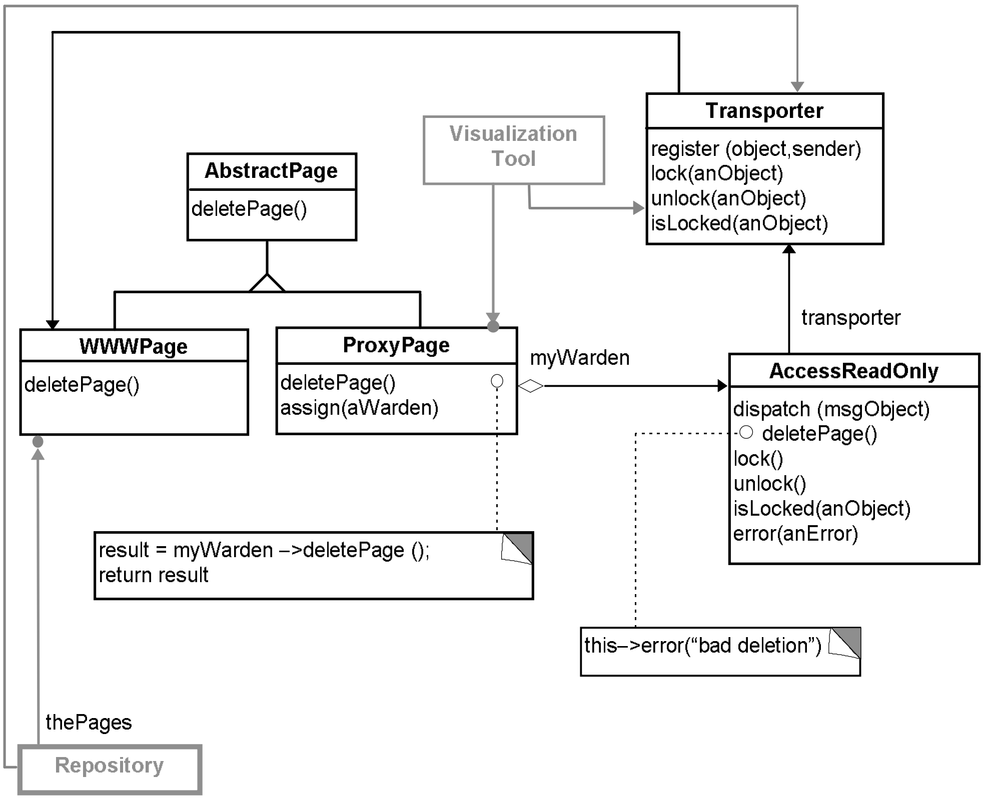

Figure 3. Warden pattern instantiated for the Visualization tool, showing how AccessReadOnly (a kind of Warden) denies the request to delete a page. Please note thar the distribution of instances is like the one in figure 1.

The essencial components of the Warden pattern are the following three: the object to be shared, the object that control the access to that object (warden) and the one that is in charge of the communication among hosts (transporter). There is one transporter in every host. Each transporter can know all WWW pages that can be shared. 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 an instance of ProxyPage 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 AccessReadWrite or AccessReadOnly is created and designated as the warden of the new page proxy. Methods in ProxyPage class provide a template for message dispatching and packs the messages to be sent with its parameters in a message object. Class Transporter 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 wardens). Classes AccessReadOnly and AccessReadWrite provide conformance checking for message dispatching, both in high level restrictions (like refusing to delete a page in the *dispatch()* method of AccessReadOnly) and low level synchronization and exclusion (like *lock()* and *unlock()* in AccessReadWrite, and *isLocked()* in AccessReadOnly).

Applicability:

Use the Warden Pattern when you 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.
- To synchronize and pool message passing to shared objects.
- Fine grained control of access restrictions, depending on who requests to share the object.

- Dynamic change of access rights.
- Identity checking of objects and access rights assigment, in order to to prevent an incorrect behaviour from objects of an application, but not to guard against security violations from those objects.

Do not use the Warden 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 full support for object services (as with CORBAServices).
- You have a very heterogeneous environment, with different object models and network protocols.

Structure:

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

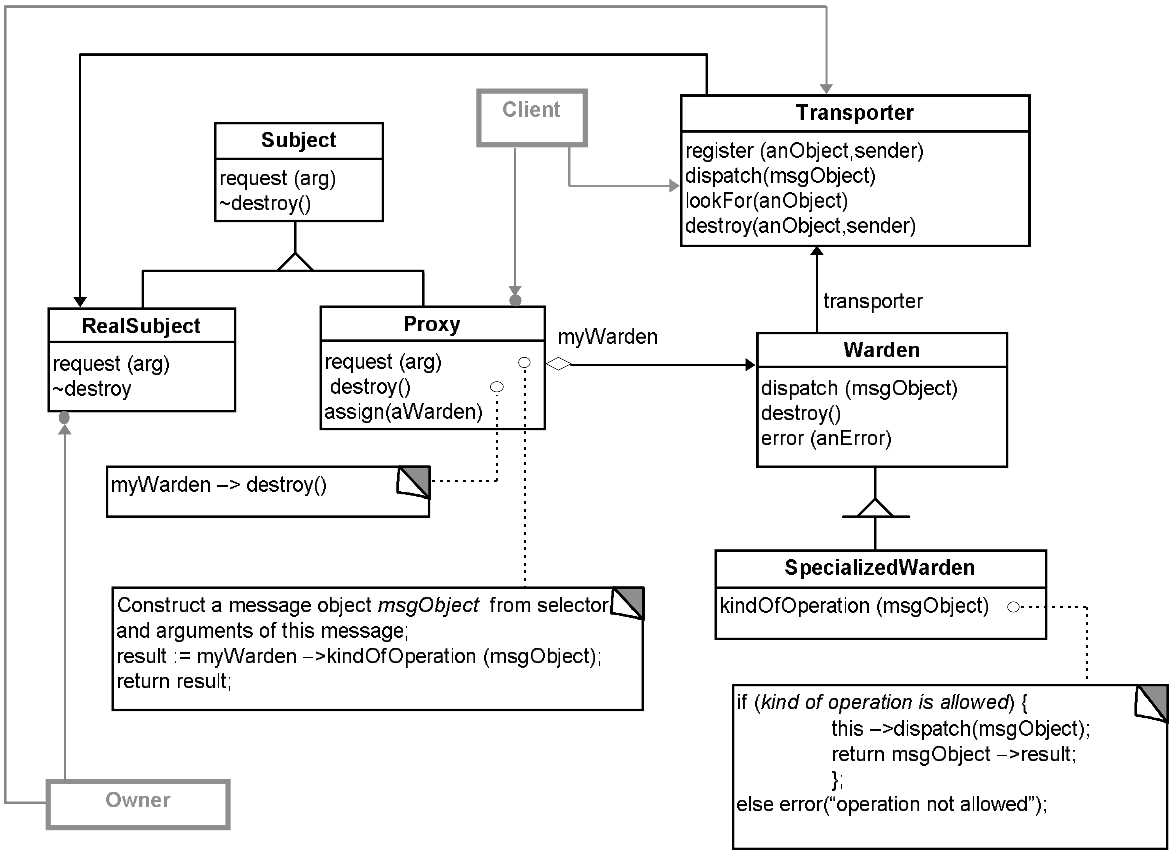

Figure 4. Structure of Warden pattern.

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.

Warden

- Works as a mediator between proxies and transporters, by sending method dispatching requests, notifying object destruction and optionally handling error notifications. Warden is an abstract class, SpecializedWarden is the one that implements concrete wardens.
- Controls the access to the real subject by the client, asking the transported to carry the message out to the real subject.

SpecializedWarden

- Provides conformance checking for message dispatching both in access rights restrictions and low level synchronization and exclusion.
- There will be a subclass of SpecializedWarden 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 warden 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 one that send and object to register is designated as the owner of the object and is allowed to destroy the object.
- Each time a system needs to access a remote object by the first time, it asks the local transporter for the object. The transporter will then check the object existence in some place over the network, and only in case that it exists, it will create a warden and a proxy in the system that requested for this object, to access and control the access to that remote object (see interaction diagram 4.2).
- In following accesses, when a client needs to send a request to the subject, it will interact with the proxy, that in turn delegates the request to the warden. The later will check the validity of the request and when appropiate, communicates with the transporter in charge of sending the request to the one in the remote host. The remote transporter will send the request to the real subject, and then will send back the proper result (see interaction diagram 4.3).

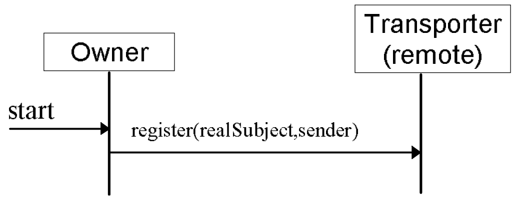

Figure 4.1: Interaction Diagram for "Object Registration"

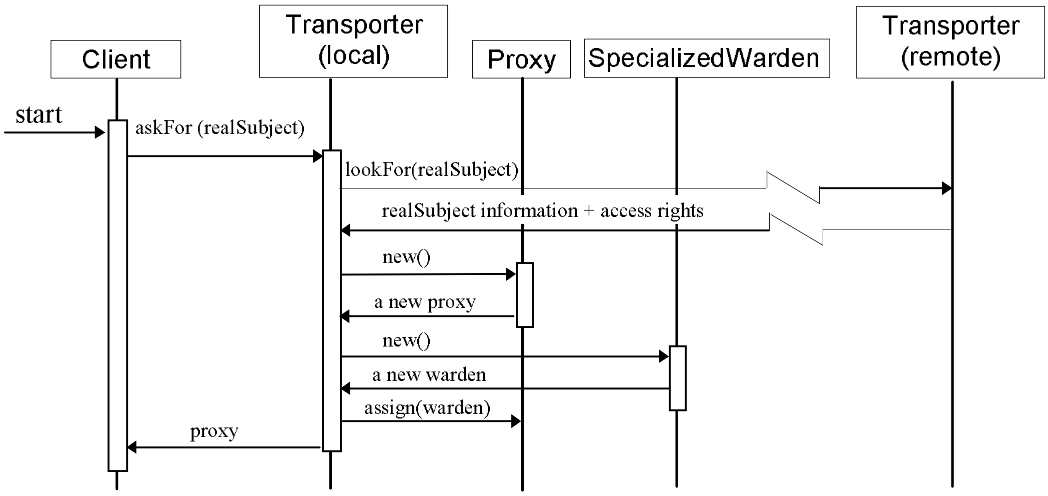

Figure 4.2: Interaction Diagram for "Remote Object Access"

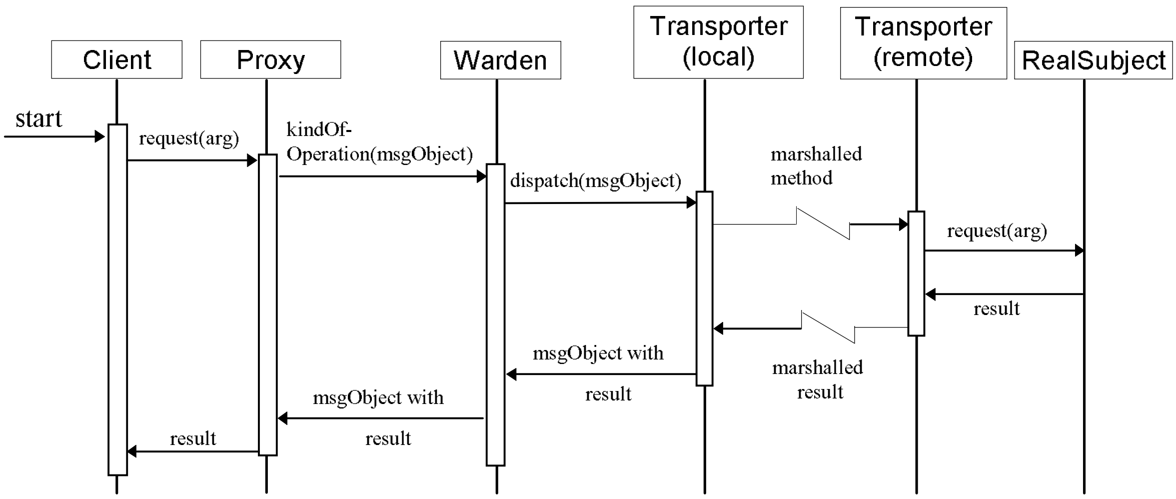

Figure 4.3: Interaction Diagram for "Remote Method Dispatching"

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 the synchronization control. It also isolates access restriction mechanism, because wardens are only visible for proxies and not for external objects.
- It trades generality for simplicity. In situations with and restricted requisites and limited resources, the Warden pattern offers an object sharing scheme that is easier to understand and implement than a full, system-level object sharing system.
- Wardens 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. This fact has to be taken in account if the necessary control is not so strict.

Variants:

Rights Assignment by Class vs. by Identity: At the moment a request to access a shared object is received by the transporter, and the availability of the object is confirmed, it has to decide which access rights to assign and create the corresponding warden. 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 Wardens vs. Protocol Wardens: The number of Warden subclasses that need to be implemented depends heavily on how specific the control should be and the different kind of access rights combinations. Wardens implement access control by the definition of a set of operations that express the general categories in which remote accesses can be classified. Every operation stands for a kind of access. Proxies send requests for method dispatching by calling a particular kind of operation in its warden.

Granularity of the operation set can vary depending on the security enforcement needed by the classes whose instances will be shared. The extreme case is the one in which every operation in a proxy has the same operation in a warden, like the examples in the *Motivation* section of this paper. In that case, operations in the warden are implemented knowing the semantic of the operations they will control in a particular proxy, at the cost of having to code a particular warden for every protocol to be checked. It should be noted that it is not always necessary to implement a Warden subclass for every class whose objects are going to be shared, but instead a Warden can watch for a whole hierarchy. For instance, in Smalltalk a single CollectionWarden 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 wardens 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 the other way round, 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 warden every time one of its methods is called. Warden 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, although the context of the problem

Warden 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 warden, depending on the rights the origin warden has decided. After the acknowledge is received, the *OK_to_share Acceptor* is eliminated from its reactor.

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 warden 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 warden grants or prohibits.

Related Patterns:

The Proxy pattern [Gamma94] discusses some issues related to remote proxies and housekeeping tasks. It's 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. In this way, the Warden patterns provide a new dimension about object sharing in distributed environments, using the Proxy pattern but enriching the structure with additional control (Warden) and transport related (Transporter) objects.

The Reactor and Acceptor patterns [Schmidt95] are also related with the way in which the Transporter class work. 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 wardens 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 Warden pattern is not.

The Warden Pattern is strongly related to the Broker Pattern [Stal95]. In some sense, Warden 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 Warden stands for client-side proxies in the Broker. The main difference between Warden and Broker is the size of the problem they attack, and the associated costs: Broker implements object sharing at the system-level in a wide context, addressing situations like the way objects expose their interfaces in an heterogeneous environment, and how objects in different networks will be able to communicate. Demands associated with the work in such environment are separated from the Broker by the implementation of services, like authentication, persistence and access rights control, that extend the scope of the Broker but are not strictly part of the pattern.

Known Uses:

There exist many implementations of the Warden Pattern:

[Dollimore93] shows an implementation of shared objects with access restriction in Eiffel. Objects are remotely represented by proxies, which are associated to a warden 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. TransporterRooms 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 Warden 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 wardens 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 directly manages 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.