8.1 Protected System

Intent

Structure a system so that all access by clients to resources is mediated by a guard which enforces a security policy.

Also Known As

Reference Monitor, Enclave

Motivation

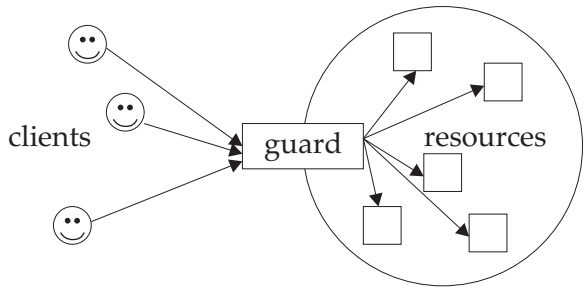
It is often desirable or imperative to protect system resources against unauthorized access. In order to do this it is necessary to evaluate requests to determine whether or not they are permitted by a policy. All requests must be evaluated against the policy; otherwise, unchecked requests might violate the policy.

To assure that all access requests are evaluated against the system's policy, a policy enforcement mechanism with the following properties must exist:

- The mechanism must be invoked on every access request.
- The mechanism must not be bypassable.
- The mechanism must correctly evaluate the policy.
- The mechanism's correct functioning must not be corruptible.
- The previous four properties must be verifiable to some stated level of confidence.

This pattern includes three elements:

1. An ''outside'', from which all access requests originate
2. An ''inside'', in which all resources are located
3. A correct, verifiable, incorruptible, non-bypassable ''guard'', which enforces policy on all requests from ''outside'' for resources ''inside''

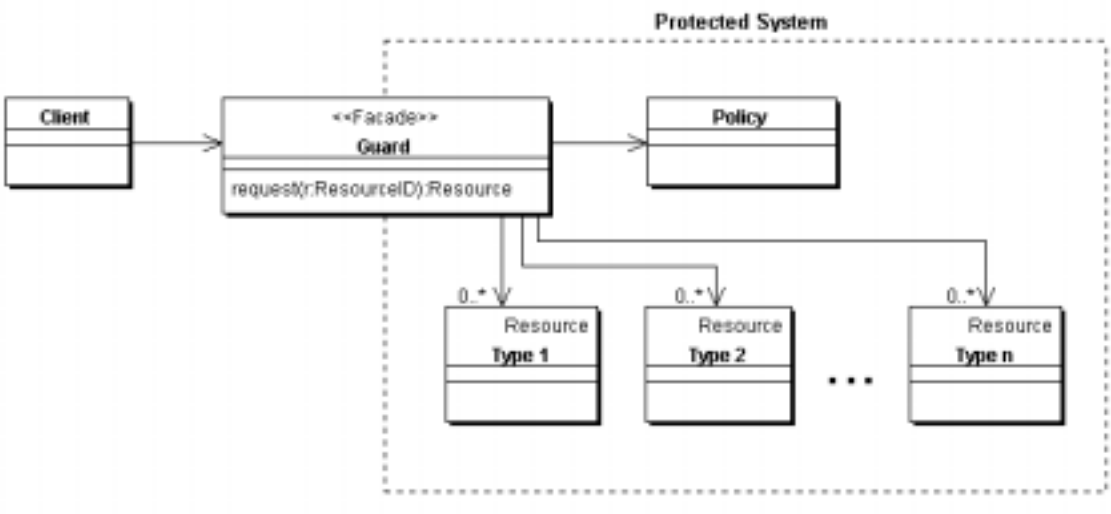


Applicability

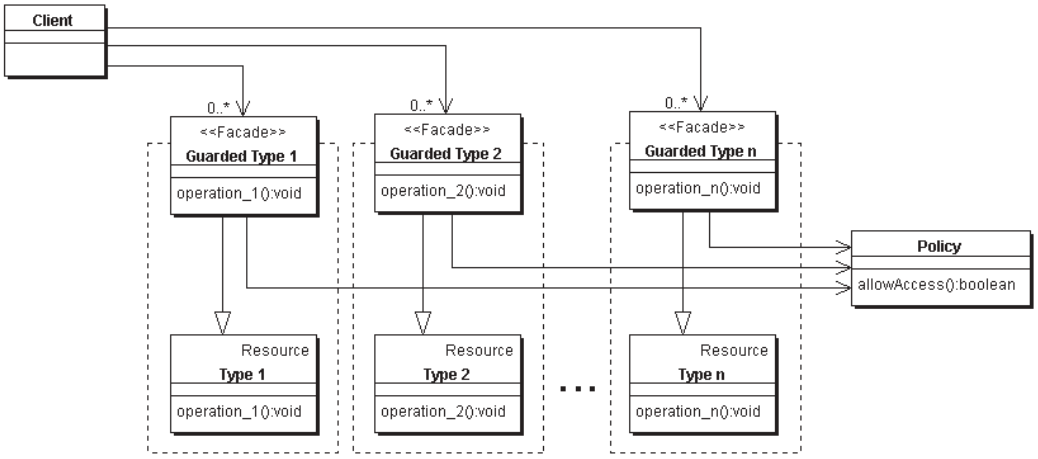
Use this pattern whenever access to resources must be granted selectively based on a policy. When designing secure systems by refinement, Protected System should be considered a candidate for the starting point of the refinement.

Structure

There are two main variants of this pattern. The first variant has a single centralized guard which mediates requests for all resources in the system.



The second variant distributes the responsibilities of the guard, such that there is a separate guard instance for each distinct type of resource.



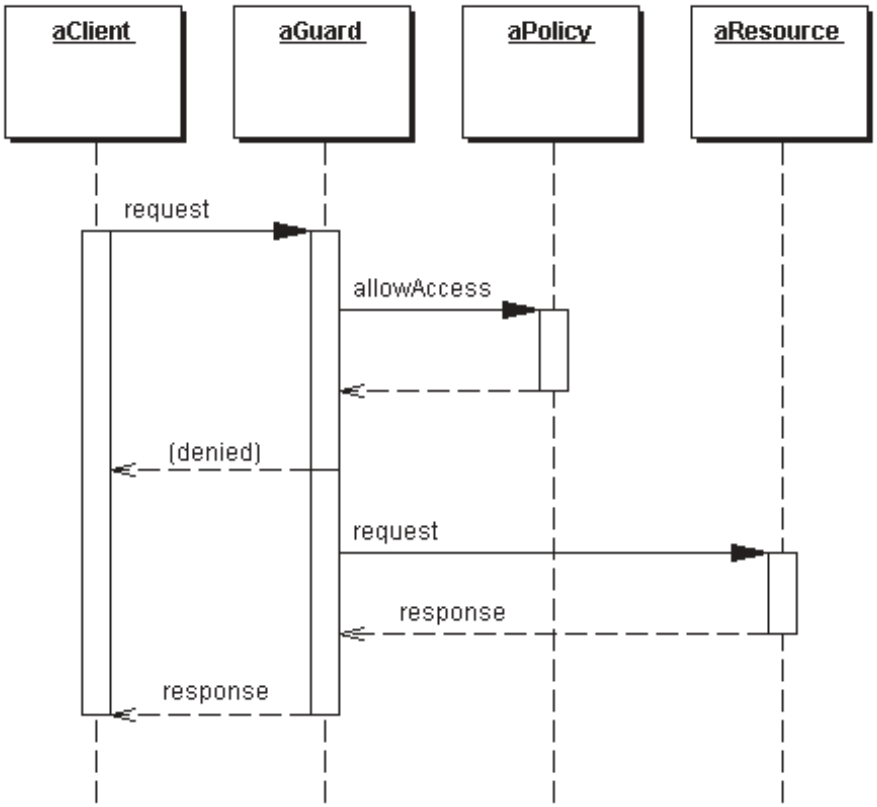
Hybrids of these variants are possible (see Implementation below).

Participants

- Client
  - — Submits access requests to guard.
- Guard
  - — Mediates requests to access protected resources. The Guard is a Facade [GoF].
  - — Evaluates each access request against a policy; grants requests which are permitted by the policy, and denies requests which are forbidden by the policy.
  - — Cannot be bypassed (no direct access by Clients to Resources is possible).
- Policy
  - — Determines whether access to the resource should be granted.
  - — Encapsulates the rules defining which Clients may access which Resources.
- Resource
  - — Services requests for access to protected resources.

Collaborations

- Clients submit access requests to the Protected System's Guard.
- The Guard determines whether access requests should be granted or denied by consulting the Policy.
- If the Guard determines that an access request should be denied, it discards the request and returns. If the Guard determines that the access request should be granted, it passes the request on to the Resource for fulfillment and returns the response to the Client.



Consequences

Use of the Protected System pattern:

- Isolates resources.

  The system's resources are isolated by the Guard from any accesses which do not conform to the security policy enforced by the Guard.

- Loosens coupling between security policy and Resource implementation.

  Resource implementations do not need to be aware of security policy and do not have to be modified when security policy changes, since the policy is enforced by the Guard.

- Improves system assurability.

  Only the Guard implementation needs to be evaluated for correctness in order to ensure that the system correctly enforces its security policy.

- Degrades performance.

  In almost all implementations, interposing a Guard between the Client and the Resource imposes a performance penalty; this penalty may be significant. In operating system kernel implementations, the performance cost to cross the kernel boundary is often much higher than the cost to make a procedure call when the caller and the called routine are both in ''user space'' or both in ''system space''. In network configurations, Guards are usually network proxies (for example, routers) and their use requires an extra network message for each resource access.

Implementation

Several issues need to be considered when implementing the Protected System pattern:

- Isolation

  In order to provide complete protection of resources, the Guard must be non-bypassable.

  A firewall is a good example of a Guard which may be bypassable; if modems permit intermittent dial-up access to machines ‘‘inside’’, but access to the modems does not go through the firewall, then it will be possible to bypass the firewall and its associated security policy.

  Many microprocessor designs do not support complete address-space isolation between programs running in ‘‘system state’’ and programs running in ‘‘user state’’. It is difficult or impossible to design operating system kernels which are not bypassable to run on such microprocessors.

  Virtual machine architectures also suffer from failures of address-space isolation; several versions of the Java Virtual Machine, for example, shared public static variables between thread address spaces, which violated the thread isolation property assumed by the Java security model.

- Guard self-protection

  In order to protect resources, the Guard must function correctly. Among other things, this means that the Guard must be incorruptible.

  Corruptibility is often a consequence of a failure to validate input data.

  Many systems (including many Internet servers) are vulnerable to buffer overflow attacks. Buffer overflow attacks result from the Guard’s failure to check the size of input parameters provided by the client. A buffer overflow attack causes the Guard to execute malicious code provided by the client.

  Some systems are vulnerable to data poisoning attacks; data poisoning attacks which result from the Guard’s designers failing to define error responses for all possible invalid input data values. Data poisoning attacks exploit the Guard’s unanticipated response to an ‘‘improper’’ input value.

- Assurance

  It must be possible to demonstrate that the Guard functions correctly, and that the Guard is non-bypassable and incorruptible.

  Assurance is very difficult, and its difficulty scales super-linearly with increasing system size. The Protected System pattern contributes to assurability by minimizing the amount of code which must be assured, and by modularizing it to the Guard.

  Typical assurance activities include: disciplined design processes; documentation of all aspects of system design, implementation, production, delivery, and operation; assurance inspections; rigorous testing; and formal correctness verification.

- Alternative structures for the Guard

  The first variant of this pattern shown in Structure above uses a single Guard instance which mediates access to all Resources. The client must invoke this Guard in order to perform operations on any Resource.

  The second variant shows separate Guards for each resource type. This is still considered a single protected system, as there is a common Policy controlling access to all resources,

although the access control decisions are triggered in multiple Guarded Types. Note that the interfaces presented by a Guarded Type may be identical to the Type which is being protected; in this case, the Guarded Type is a Proxy [GoF] for the underlying Type, and the Client need not be aware that it is invoking a Guarded Type.

A hybrid between these two variants is possible, such that the Client obtains its initial reference to the Resource from a centralized Guard, but the reference returned is to a Guarded Type which will perform supplementary access checks for each operation on the Resource. In this case, the centralized Guard is acting in a similar manner to an Abstract Factory [GoF].

Guards can be implemented as Proxies, or they can be embedded directly in the Resource implementation. Proxy Guards are easier to assure (because policy enforcement functionality is strongly separated from operational functionality), but they impose a larger performance penalty, because of the requirement for additional procedure calls or network messages to communicate requests and responses between the Guard and the Resource.

- Feasibility of externalizing policy enforcement

  In some systems, business rules are strongly integrated with policy enforcement, or policy is strongly dependent on the specific details of a resource request or of the resources to which access is being requested. In such systems, it may be very difficult or very inefficient to separate policy enforcement from processing of resource access requests.

  For example, if the desired policy depends on the specific values of all parameters of a resource access request, moving policy enforcement from the Resource to a centralized Guard may require essentially total duplication of the Resource manager's request processing code (and thus will impose substantial performance overhead without any corresponding gain in assurability of the policy enforcement code).

Known Uses

A very large number of secure system designs are instances of this pattern.

The Anderson Report first defined the structure described in the first variant of this pattern. It refers to the Guard together with the Resource managers it protects as a ''Reference Monitor'', and to the Guard itself as a ''Reference Mediation Mechanism''. In an operating system whose kernel is a reference monitor, the Guard is the operating system kernel (syscall) boundary, and the protected resources are files, pipes, and other operating system objects.

The Java 2 security architecture is an example of the second variant of the pattern. There is a single AccessController object which is responsible for making the decisions on whether operations should be permitted, based on Permissions granted to executing code. The checkPermission method of the AccessController is however invoked separately by each resource class, specifying the Permission that is to be checked for the requested operation.

The Java 2 AccessController design has one notable advantage over a monolithic reference monitor: new types of Resource can be introduced into the Protected System by simply defining a new Permission class. No change is necessary to the AccessController implementation (that is, the Policy component of this pattern).

A firewall is a Protected System whose guard is a router and whose resources are IP addresses and ports of systems ''inside'' the firewall.

A bank vault is a Protected System whose guard is the walls and vault door and whose resources are cash and gold bars.

Related Patterns

- As noted above, the Guard of a Protected System may be a Proxy [GoF]. GoF refers to a proxy used for this purpose as a ‘‘Protection Proxy’’.
- Further details relating to the Policy class are covered in the Policy [TG_SDP] pattern below.
- Role-based access control [APLRAC].
- Guard for Protected System [Brown-Fernandez].
- Security policy [Mahmoud].
- Security models, multi-level security [Fernandez-Pan].
- Object Filter access control framework [Object-Filter].
- Enabling application security, single access point, checkpoint [Yoder-Barcalow].