Unit 4

SOME DESIGN PATTERNS

- Structural decomposition: Whole – Part;
- Organization of work: Master – Slave;
- Access Control: Proxy.

UNIT 4
SOME DESIGN PATTERNS

INTRODUCTION:

Design patterns are medium scale patterns. They are smaller in scale than architectural patterns, but are at a higher level than the programming language specific idioms.
We group design patterns into categories of related patterns, in the same way as we did for architectural patterns:

- **Structural Decomposition**
  This category includes patterns that support a suitable decomposition of subsystems and complex components into co-operating parts. The Whole-Part pattern is the most general pattern we are aware of in this category.
- **Organization of Work.**
  This category comprises patterns that define how components collaborate together to solve a complex problem. We describe the Master-Slave pattern, which helps you to organize the computation of services for which fault tolerance or computational accuracy is required.
- **Access Control.**
  Such patterns guard and control access to services or components. We describe the Proxy pattern here.
- **Management.**
  This category includes patterns for handling homogenous collections of objects, services and components in their entirety. We describe two patterns: the Command Processor pattern addresses the management and scheduling of user commands, while the View Handler pattern describes how to manage views in a software system.
- **Communication.**
  Patterns in this category help to organize communication between components. Two patterns address issues of inter-process communication: the Forwarder-Receiver pattern deals with peer-to-peer communication, while the Client Dispatcher-Server pattern describes location-transparent communication in a Client-Server structure.

STRUCTURAL DECOMPOSITION:

Subsystems and complex components are handled more easily if structured into smaller independent components, rather than remaining as monolithic block of code.
We discuss whole-part design pattern that supports the structural decomposition of the component.

WHOLE-PART

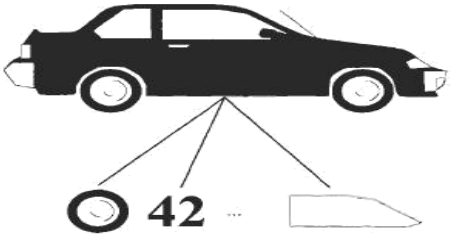
Whole-part design pattern helps with the aggregation of components that together form a semantic unit.

An aggregate component, the whole, encapsulates its constituent components, the parts, organizes their collaboration, and provides a common interface to its functionality. Direct access to the parts is not possible.

Example:

A computer-aided design (CAD) system for 2-D and 3-D modelling allows engineers to design graphical objects interactively. For example, a car object aggregates several smaller objects such as wheels and windows, which

themselves may be composed of even smaller objects such as circles and polygons. It is the responsibility of the car object to implement functionality that operates on the car as a whole, such as rotating or drawing.



Context:
Implementing aggregate objects

Problem:

- In almost every software system objects that are composed of other objects exists. Such aggregate objects do not represent loosely-coupled set of components. Instead, they form units that are more than just a mere collection of their parts.
- The combination of the parts makes new behavior emerge- such behavior is called emergent behavior.
- We need to balance following forces when modeling such structures;
  - ✓ A complex object should either be decomposed into smaller objects, or composed of existing objects, to support reusability, changeability and the recombination of the constituent objects in other types of aggregate.
  - ✓ Clients should see the aggregate object as an atomic object that does not allow any direct access to its constituent parts.

Solution:

- Introduce a component that encapsulates smaller objects and prevents clients from accessing theses constitutes parts directly.
- Define an interface for the aggregate that is the only means of access to the functionalities of the encapsulated objects allowing the aggregate to appear as semantic unit.
- The principle of whole-part pattern is applicable to the organization of three types of relationship
  - ✓ An *assembly-parts* relationship which differentiation b/w a product and its parts or subassemblies.
  - ✓ A *container-contents* relationship, in which the aggregated object represents a container.
  - ✓ The *collection-members* relationship, which helps to group similar objects.

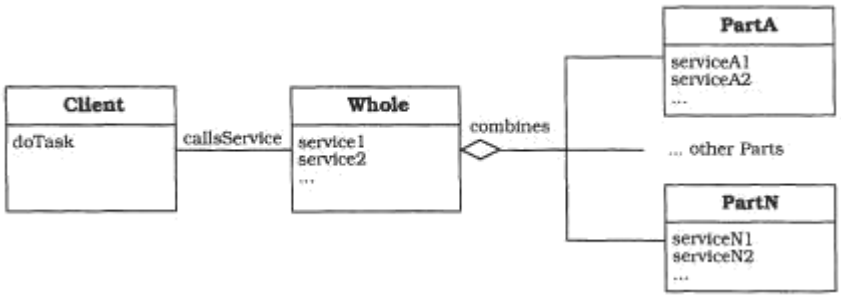
Structure:
The Whole-Part pattern introduces two types of participant:

- **Whole**
  - Whole object represents an aggregation of smaller objects, which we call parts.
  - It forms a semantic grouping of its parts in that it co ordinates and organizes their collaboration.
  - Some methods of whole may be just place holder for specific part services when such a method is invoked the whole only calls the relevant part services, and returns the result to the client.
- **Part**
  - Each part object is embedded in exactly one whole. Two or more parts cannot share the same part. Each part is created and destroyed within the life span of the whole.

Class Whole	Collaborators • Part
Responsibility • Aggregates several smaller objects. • Provides services built on top of part objects. • Acts as a wrapper around its constituent parts.	

Class Part	Collaborators -
Responsibility • Represents a particular object and its services.	

Static relationship between whole and its part are illustrated in the OMT diagram below



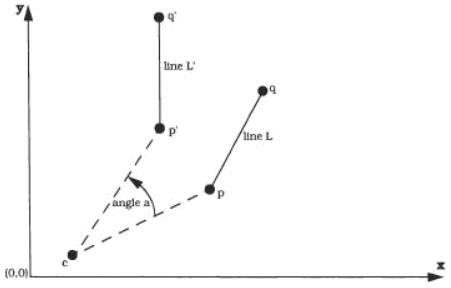
Dynamics:

The following scenario illustrates the behavior of a Whole-Part structure. We use the two-dimensional rotation of a line within a CAD system as an example. The line acts as a Whole object that contains two points p and **q** as Parts. A client asks the line object to rotate around the point c and passes the rotation angle as an argument.

The rotation of a point **p** around a center c with an angle a can be calculated using the following formula:

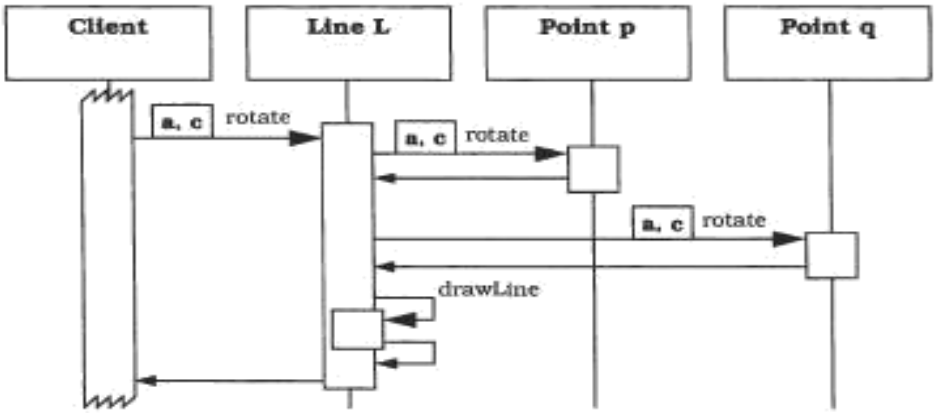
$$\vec{p}' = \begin{bmatrix} \cos a & -\sin a \\ \sin a & \cos a \end{bmatrix} \cdot (\vec{p} - \vec{c}) + \vec{c}$$

In the diagram below the rotation of the line given by the points p and q is illustrated.



The scenario consists of four phases:

- A client invokes the rotate method of the line L and passes the angle a and the rotation center c as arguments.
- The line L calls the rotate method of the point p.
- The line L calls the rotate method of the point q.
- The line L redraws itself using the new positions of p_1 and q_1 as endpoints.



Implementation:

1. **Design the public interface of the whole**
   - Analyze the functionality the whole must offer to its clients.
   - Only consider the clients view point in this step.
   - Think of the whole as an atomic component that is not structured into parts.
2. **Separate the whole into parts, or synthesize it from existing ones.**
   - There are two approaches to assembling the parts either assemble a whole 'bottom-up' from existing parts, or decompose it 'top-down' into smaller parts.
   - Mixtures of both approaches is often applied
3. **If you follow a bottom up approach**, use existing parts from component libraries or class libraries and specify their collaboration.
4. **If you follow a top down approach, partition the Wholes services into smaller collaborating services** and map these collaborating services to separate parts.
5. **Specify the services of the whole in terms of services of the parts.**

   Decide whether all part services are called only by their whole, or if parts may also call each other. Two are two possible ways to call a Part service:

   @ If a client request is forwarded to a Part service, the Part does not use any knowledge about the execution context of the Whole, relying on its own environment instead.

   @ A delegation approach requires the Whole to pass its own context information to the Part.
6. **Implement the parts**

   If parts are whole-part structures themselves, design them recursively starting with step1 . if not reuse existing parts from a library.
7. **Implement the whole**

   Implement services that depend on part objects by invoking their services from the whole.

Variants:

- **Shared parts:**

  This variant relaxes the restriction that each Part must be associated with exactly one Whole by allowing several Wholes to share the same Part.
- **Assembly parts**

  In this variant the Whole may be an object that represents an assembly of smaller objects.
- **Container contents**

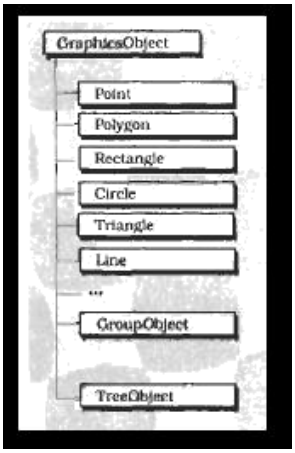
  In this variant a container is responsible for maintaining differing contents
- **Collection members**

  This variant is a specialization of Container-Contents, in that the Part objects all have the same type.
- **Composite pattern**

  It is applicable to Whole-Part hierarchies in which the Wholes and their Parts can be treated uniformly-that is, in which both implement the same abstract interface.

Example resolved:

In our CAD system we decide to define a Java package that provides the basic functionality for graphical objects. The class library consists of atomic objects such as circles or lines that the user can combine to form more complex entities. We implement these classes directly instead of using the standard Java package awt (Abstract Windowing Toolkit) because awt does not offer all the functionality we need.



Known uses:

- The key abstractions of many **object-oriented applications** follow the Whole-Part pattern.
- Most **object-oriented class libraries** provide collection classes such as lists, sets and maps. These classes implement the Collection- Member and Container-Contents variants.
- **Graphical user interface toolkits** such as Fresco or **ET++** use the Composite variant of the Whole-Part pattern.

Consequences:

The whole-part pattern offers several *Benefits:*

- **Changeability of parts:**
  Part implementations may even be completely exchanged without any need to modify other parts or clients.
- **Separation of concerns:**
  Each concern is implemented by a separate part.
- **Reusability in two aspects:**
  - Parts of a whole can be reused in other aggregate objects
  - Encapsulation of parts within a whole prevents clients from 'scattering' the use of part objects all over its source code.

The whole-part pattern suffers from the following *Liabilities:*

- **Lower efficiency through indirection**
  Since the Whole builds a wrapper around its Parts, it introduces an additional level of indirection between a client request and the Part that fulfils it.
- **Complexity of decomposition into parts.**
  An appropriate composition of a Whole from different Parts is often hard to find, especially when a bottom-up approach is applied.

ORGANIZATION OF WORK

The implementation of complex services is often solved by several components in co operation. To organize work optimally within such structures you need to consider several aspects.

We describe one pattern for organizing work within a system → maser-slave pattern.

MASTER-SLAVE

*The **Master-Slave** design pattern supports fault tolerance, parallel computation and computational accuracy. A master component distributes work to identical slave components and computes a final result from the results these slaves return.*

Example:

Travelling salesman problem

Context:

Portioning work into semantically identical subtasks

Problem:

Divide and conquer: here work is partitioned into several equal subtasks that are processed independently. The result of the whole calculation is computed from the results provided by each partial process.

Several forces arise when implementing such a structure

- Clients should not be aware that the calculation is based on the 'divide and conquer' principle.
- Neither clients nor the processing of subtasks should depend on the algorithms for partitioning work and assembling the final result.
- It can be helpful to use different but semantically identical implementations for processing subtasks.
- Processing of subtasks sometimes need co ordination for ex. In simulation applications using the finite element method.

Solution:

- Introduce a co ordination instance b/w clients of the service and the processing of individual subtasks.
- A master component divides work into equal subtasks, delegates these subtasks to several independent but semantically identical slave components and computes a final result from the partial results the slaves return.
- The general principle is found in three application areas
  - Fault tolerance → Failure of service executions can be detected and handled
  - Parallel computing → A complex task is divided into a fixed number of identical sub-tasks that are executed in parallel.
  - Computational accuracy → Inaccurate results can be detected and handled.

Structure:

- **Master component:**
  - ✓ Provides the service that can be solved by applying the 'divide and conquer' principle.
  - ✓ It implements functions for partitioning work into several equal subtasks, starting and controlling their processing and computing a final result from all the results obtained.

✓ It also maintains references to all slaves instances to which it delegates the processing of subtasks.

❖ **Slave component:**

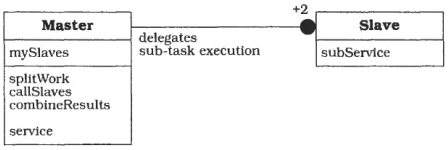
✓ Provides a sub-service that can process the subtasks defined by the master

✓ There are at least two instances of the slave component connected to the master.

Class Master	***Collaborators***
Responsibility • Partitions work among several slave components • Starts the execution of slaves • Computes a result from the sub-results the slaves return.	• Slave

Class Slave	***Collaborators***
Responsibility • Implements the sub-service used by the master.	-

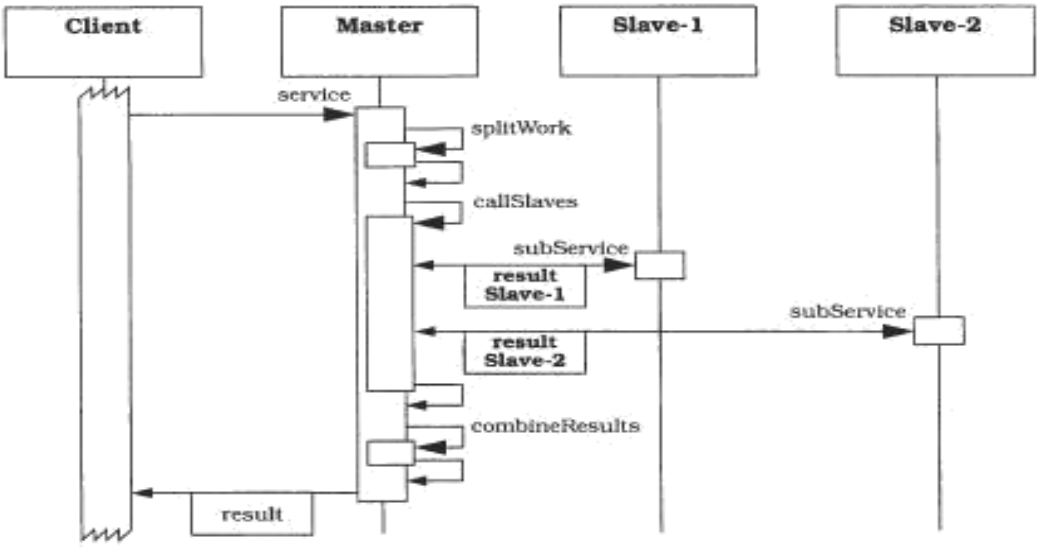
The structure defined by the Master-Slave pattern is illustrated by the following OMT diagram.



Dynamics:

The scenario comprises six phases:

- A client requests a service from the master.
- The master partitions the task into several equal sub-tasks.
- The master delegates the execution of these sub-tasks to several slave instances, starts their execution and waits for the results they return.
- The slaves perform the processing of the sub-tasks and return the results of their computation back to the master.
- The master computes a final result for the whole task from the partial results received from the slaves.
- The master returns this result to the client.



Implementation:

1. **Divide work:**
   Specify how the computation of the task can be split into a set equal sub tasks. Identify the sub services that are necessary to process a subtask.
2. **Combine sub-task results**
   Specify how the final result of the whole service can be computed with the help of the results obtained from processing individual sub-tasks.
3. **Specify co operation between master and slaves**
   - Define an interface for the subservice identified in step1 it will be implemented by the slave and used by the master to delegate the processing of individual subtask.
   - One option for passing subtasks from the master to the slaves is to include them as a parameter when invoking the subservice.
     Another option is to define a repository where the master puts sub tasks and the slaves fetch them.
4. **Implement the slave component**s according to the specifications developed in previous step.
5. **Implement the master** according to the specifications developed in step 1 to 3
   - There are two options for dividing a task into subtasks.
     - The first is to split work into a fixed number of subtasks.
     - The second option is to define as many subtasks as necessary or possible.
   - Use strategy pattern to support dynamic exchange and variations of algorithms for subdividing a task.

Variants:

- There are 3 application areas for master slave pattern.
  - **Master-slave for fault tolerance**
    In this variant the master just delegates the execution of a service to a fixed number of replicated implementations, each represented by a slave.
  - **Master-slave for parallel computation**
    In this variant the master divides a complex task into a number of identical sub-tasks, each of which is executed in parallel by a separate slave.
  - **Master-slave for computational concurrency.**
    In this variant the execution of a service is delegated to at least three different implementations, each of which is a separate slave.

- Other variants
  - **Slaves as processes**
    To handle slaves located in separate processes, you can extend the original Master-Slave structure with two additional components
  - **Slaves as threads**
    In this variant the master creates the threads, launches the slaves, and waits for all threads to complete before continuing with its own computation.
  - **Master-slave with slave co ordination**
    In this case the computation of all slaves must be regularly suspended for each slave to coordinate itself with the slaves on which it depends, after which the slaves resume their individual computation.

Known uses:

- **Matrix multiplication**. Each row in the product matrix can be computed by a separate slave.
- **Transform-coding** an image, for example in computing the discrete cosine transform

(DCT) of every 8 x 8 pixel block in an image. Each block can be computed by a separate slave.

- Computing the **cross-correlation** of two signals
- The **Workpool model** applies the master-slave pattern to implement process control for parallel computing
- The concept of **Gaggles** builds upon the principles of the Master-Slave pattern to handle 'plurality' in an object-oriented software system. **A *gaggle*** represents a set of replicated service objects.

Consequences:

The Master-Slave design pattern provides several *Benefits:*

- **Exchangeability and extensibility**
  By providing an abstract slave class, it is possible to exchange existing slave implementations or add new ones without major changes to the master.
- **Separation of concerns**
  The introduction of the master separates slave and client code from the code for partitioning work, delegating work to slaves, collecting the results from the slaves, computing the final result and handling slave failure or inaccurate slave results.
- **Efficiency**
  The Master-Slave pattern for parallel computation enables you to speed up the performance of computing a particular service when implemented carefully

The Master-Slave design pattern has certain *Liabilities:*

- **Feasibility**
  It is not always feasible
- **Machine dependency**
  It depends on the architecture of the m/c on which the program runs. This may decrease the changeability and portability.
- **Hard to implement**
  Implementing Master-Slave is not easy, especially for parallel computation.
- **Portability**
  Master-Slave structures are difficult or impossible to transfer to other machines

ACCESS CONTROL

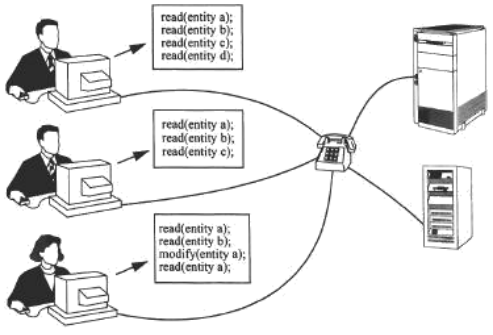
Sometimes a component or even a whole subsystem cannot or should not be accessible directly by its clients. Here we describe one design pattern that helps to protect access to a particular component: → *The proxy design pattern*

PROXY

Proxy design pattern makes the clients of a component communicate with a representative rather than to the component itself. Introducing such a place holder can serve many purposes, including enhanced efficiency, easier access and protection from unauthorized access.

Example:

Company engineering staff regularly consults databases for information about material providers, available parts, blueprints, and so on. Every remote access may be costly, while many accesses are similar or identical and are repeated often. This situation clearly offers scope for optimization of access time and cost.



Context:

A client needs access to the services of another component direct access is technically possible, but may not be the best approach.

Problem:

It is often inappropriate to access a component directly.

A solution to such a design problem has to balance the following forces.

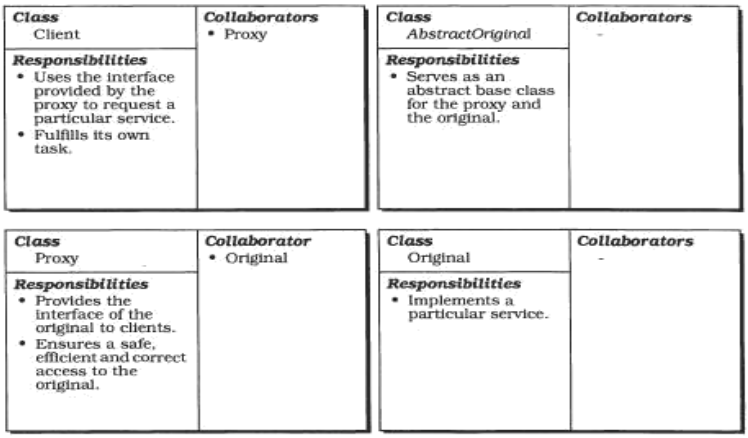
- Accessing the component should be runtime efficient, cost effective and safe for both the client and the component
- Access to component should be transparent and simple for the client. The client should particularly not have to change its calling behavior and syntax from that used to call any other direct access component.
- The client should be well aware of possible performance or financial penalties for accessing remote clients. Full transparency can obscure cost differences between services.

Solution:

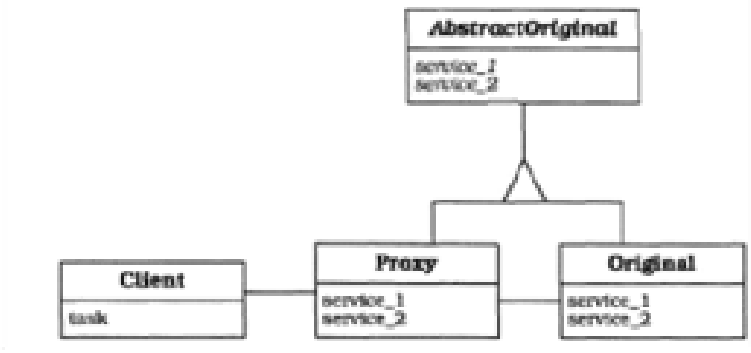
- Let the client communicate with a representative rather than the component itself.
- This representative called a 'proxy' offers the interface of the component but performs additional pre and post processing such as access control checking or making read only copies of the 'original'.

Structure:

- **Original**
  - Implements a particular service
- **Client**
  - Responsible for specific task
  - To do this, it involves the functionality of the original in an indirect way by accessing the proxy.
- **Proxy**
  - Offers same interface as the original, and ensures correct access to the original.
  - To achieve this, the proxy maintains a reference to the original it represents.
  - Usually there is one-to-one relationship b/w the proxy and the original.
- **Abstract original**
  - Provides the interface implemented by the proxy and the original. i.e, serves as abstract base class for the proxy and the original.



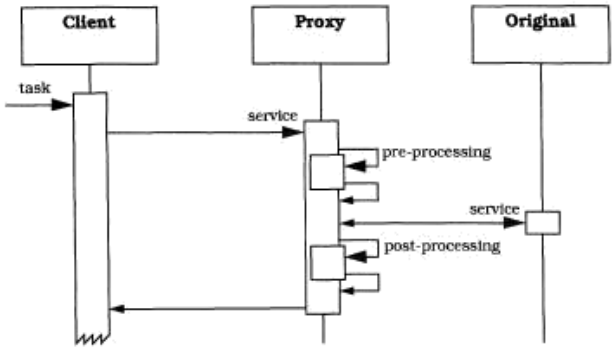
The following **OMT** diagram shows the relationships between the classes graphically:



Dynamics:

The following diagram shows a typical dynamic scenario of a Proxy structure.

- ♣ While working on its task the client asks the proxy to carry out a service.
- ♣ The proxy receives the incoming service request and pre-processes it.
- ♣ If the proxy has to consult the original to fulfill the request, it forwards the request to the original using the proper communication protocols and security measures.
- ♣ The original accepts the request and fulfills it. It sends the response back to the proxy.
- ♣ The proxy receives the response. Before or after transferring it to the client it may carry out additional post-processing actions such as caching the result, calling the destructor of the original or releasing a lock on a resource.



Implementation:

1. **Identify all responsibilities for dealing with access control to a component**
   Attach these responsibilities to a separate component the proxy.
2. **If possible introduce an abstract base class that specifies the common parts of the interfaces of both the proxy and the original.**
   Derive the proxy and the original from this abstract base.
3. **Implement the proxy's functions**
   To this end check the roles specified in the first step
4. **Free the original and its client** from responsibilities that have migrated into the proxy.
5. **Associate the proxy and the original** by giving the proxy a handle to the original. This handle may be a pointer a reference an address an identifier, a socket, a port, and so on.
6. **Remove all direct relationships between the original and its client**
   Replace them by analogous relationships to the proxy.

Variants:

- **Remote proxy:**
  Clients of remote components should be scheduled from network addresses and IPC

protocols.

- **Protection proxy:**
  Components must be protected from unauthorized access.
- **Cache proxy:**
  Multiple local clients can share results from remote components.
- **Synchronization proxy:**
  Multiple simultaneous accesses to a component must be synchronized.
- **Counting proxy:**
  Accidental deletion of components must be prevented or usage statistics collected
- **Virtual proxy:**
  Processing or loading a component is costly while partial information about the component may be sufficient.
- **Firewall proxy:**
  Local clients should be protected from the outside world.

Known uses:

- **NeXT STEP**
  The Proxy pattern is used in the NeXTSTEP operating system to provide local stubs for remote objects. Proxies are created by a special server on the first access to the remote object.
- **OMG-COBRA**
  It uses the Proxy pattern for two purposes. So called 'client-stubs', or IDL-stubs, guard clients against the concrete implementation of their servers and the Object Request Broker.
- **OLE**
  In Microsoft OLE servers may be implemented as libraries dynamically linked to the address space of the client, or as separate processes. Proxies are used to hide whether a particular server is local or remote from a client.
- **WWW proxy**
  It gives people inside the firewall concurrent access to the outside world. Efficiency is increased by caching recently transferred files.
- **Orbix**
  It is a concrete OMG-CORBA implementation, uses remote proxies. A client can bind to an original by specifying its unique identifier.

Consequences:

The Proxy pattern provides the following *Benefits:*

- **Enhanced efficiency and lower cost**
  The Virtual Proxy variant helps to implement a 'load-on-demand' strategy. This allows you to avoid unnecessary loads from disk and usually speeds up your application
- **Decoupling clients from the location of server components**
  By putting all location information and addressing functionality into a Remote Proxy variant, clients are not affected by migration of servers or changes in the networking infrastructure. This allows client code to become more stable and reusable.
- **Separation of housekeeping code from functionality.**
  A proxy relieves the client of burdens that do not inherently belong to the task the client is to perform.

The Proxy pattern has the following *Liabilities:*

- **Less efficiency due to indirection**
  All proxies introduce an additional layer of indirection.
- **Over kill via sophisticated strategies**
  Be careful with intricate strategies for caching or loading on demand they do not always pay.