UNIT 7

Design Pattern: Whole-Part

This design pattern belongs to the category of Structural Decomposition, which support a suitable decomposition of subsystems and complex components into cooperating parts.

The context in which you will consider this design pattern is when you are implementing aggregate objects. There are a few forces to balance when using this pattern. Parts might have some simple behavior, but combining the Parts might give some emergent behavior. A complex object should either be decomposed into smaller objects, or composed of existing objects. Clients should see the whole object as just one object, and allow no direct access to its constituent Parts.

Solution

The Whole-part pattern introduces a component that encapsulates several smaller objects. This aggregate object will appear as a single semantic unit to other objects in the system. The Whole object defines an interface to allow other objects to access the functionality of the subsystem, as they cannot access constituent Parts directly.

There are three types of relationships that can exist between the Whole object and the various Part objects in the system.

1. Assembly-Parts relationship: Here there is tight coupling between Parts and whole. The Aggregate has a structure which constrains Parts. Types of Parts, number of each Part is predefined.

2. Container-Contents relationship is less-tightly coupled and it allows for dynamically adding or removing Parts.

3. Collection-Member relationship is when all Part objects are identical or similar in behavior. All Parts are treated equally and the Aggregate provides functionality to access members.

Structure

Whole object represents an aggregation of smaller objects or Parts. Whole coordinates and organizes the collaboration of Parts. By utilizing the functionality of a single part or multiple cooperating Parts, the Whole object provides services to external world. The Whole may additionally provide functionality without having to invoke any Part object.

External clients can access only the interfaces exposed by the Whole, as the Part objects are out of bounds for the clients. So, Whole acts as a wrapper around its constituent Parts. The

structure of whole can impose some constraint on Part objects. The Whole object also protects Parts from unauthorized access.

Each Part object can belong to only one Whole, i.e., sharing of Parts is not allowed. Whole creates the Part objects and when Whole exits, Part objects cannot persist.

Implementation

The first step is to design the public interface of the Whole. So, first we need to analyze the functionality the Whole must offer to its clients and decide on the interface to give access to the functionalities for the clients. When doing this, we look at the services only from the client's viewpoint.

The next step is to separate the Whole into Parts. There could be situations where we might have to synthesize a Whole from existing ones, i.e., take a bottom-up approach. In other situations we might need to take a Top-down approach, where we have to decompose the functionality of the subsystem into constituent parts.

The bottom-up approach composes Wholes from loosely-coupled Parts, which can be reused when building other types of Whole. The problem sometimes is that the Parts might not provide complete functionality, so, we might need to write some glue-code to tailor the part to fit the requirements

In the top-down approach, functionality of Whole is Partitioned and allocated to different Part objects. The advantage is that there is no awkward glue-code, but we will have tight coupling which makes reuse difficult.

If using bottom-up approach and using existing components, we need to specify relationships between the Parts and between Parts and Whole. If system functionality is not covered by existing Parts, we need to specify Parts that need to be built, probably using a top-down approach.

If using top-down approach, we need to partition the Whole's services into smaller collaborating services and map these collaborating services to separate Parts. The decomposition strategy selected should provide an easy way of implementing the services of the Whole object.

The next step is to specify the services of the Whole in terms of services of the Parts. We can categorize services in two ways: services provided by using Part objects and the services the Whole performs on its own without invoking any Part objects.

If the services are provided by using Part objects we should strive to ensure that the Part objects need no context information, resulting in loose-coupling. But if Part objects need context information, it will result in tight coupling.

A key decision to be made at this stage is if Parts can talk to each other or if all communication has to be mediated through Whole.

The next step will be to implement the Parts and the Whole. Whole should manage life cycle of Parts. We have to decide and implement how Part objects will be created and destroyed.

Variants

1. Shared Parts variant is one where several Wholes can share one Part. Here the life-span of a shared Part is decoupled from that of its Whole.
2. Assembly-Parts variant is one where the Whole is an assembly of smaller objects. Note that the Part Objects can itself be composite objects, so the subsystem has a Tree or a hierarchical structure. The Assembly-Parts structures are fixed, so no addition or removal of Parts at run-time is allowed. But, Exchange of Parts with other Parts of the same type might be allowed.
3. In the Container-Contents variant, the Whole acts as a container for different Part objects. Here, we can add or remove Part objects dynamically.
4. Collection-Members is a specialization of Container-Contents variant, where the Part objects are of same type. The Parts are usually not coupled to or dependent on each other. Here, too, we can add or remove Part objects dynamically. The Whole should provide some functionality for iterating over some or all members.
5. In the Composite pattern the Whole-Part is represented as Tree. But the important feature is that there is a common interface to composite/simple objects, so we can treat objects consistently as though they were simple.

Benefits

Changeability of Parts: Since, Parts are not exposed to clients, we can replace Parts as long as system can provide same functionality

Separation of concerns: Each concern is implemented by a separate Part and we can implement complex strategies by composing them from simpler services

Reusability: Parts of a Whole can be reused in other aggregate objects and Wholes can be reused as well, because Clients are agnostic about the Parts.

Liabilities

Lower efficiency through indirection: The Clients have to go through Whole, which is an overhead not present in monolithic systems. The performance degradation is especially more when Parts are composite objects

Complexity of decomposition into Parts: Partitioning solution space is often not straightforward.

Organization of Work

- The implementation of complex services is often solved by several components in cooperation.

- To organize work optimally within such structures we need to consider several aspects.
- Several general principles apply when organizing the implementation of complex services.
  - separation of concerns
  - divide and conquer
- Master-Slave applies the 'divide and conquer' principle
- Widely used in Parallel and Distributed Computing
- Also, in implementing triple modular redundancy

Types:

- The ***Chain of Responsibility*** pattern avoids coupling the sender of a request to its receiver by giving more than one object the chance to handle the request.
- The ***Command*** pattern encapsulates a request as an object, allowing you to parameterize clients with different requests, to queue or log requests and to support undoable operations.
- The ***Mediator*** pattern defines an object that encapsulates the way in which a set of objects interact.

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

- The task is to find an optimal round trip between a given set of locations such as the shortest trip that visits each location exactly once.
- The solution to this problem is of high computational complexity.
- Generally, the solution to the traveling-salesman problem with n locations is the best of (n-l)! possible routes.

Context: Partitioning work into semantically-identical sub-tasks.

Problem:

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 sub-tasks 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 sub-tasks.
- Processing of sub-tasks sometimes needs coordination, for example in simulation applications using the finite element method.

Solution

- Introduce a coordination instance between clients of the service and the processing of individual sub-tasks.
- A master component divides work into equal sub-tasks
- Delegates these sub-tasks to several independent but semantically-identical slave components
- And computes a final result from the partial results the slaves return.
- This general principle is found in three application areas:
  - Fault Tolerance

  - Parallel Computing
  - Computational Accuracy
- Provide all slaves with a common interface. Let clients of the overall service communicate only with the master.
- We decide to approximate the solution to the traveling-salesman problem by comparing a fixed number of trips.
- Our strategy for selecting trips is simple-we just pick them randomly.
- The program is tuned for a CM-5 computer from Thinking Machines Corporation with sixty-four processors.
- To take advantage of the CM-5 multi-processor architecture, the lengths of different trips are calculated in parallel.
- We therefore implement the trip length calculation as a slave.
- A master determines a priori the number of slaves that are to be instantiated, specifies how many trips each slave instance should compare, launches the slave instances, and selects the shortest trip from all trips returned.

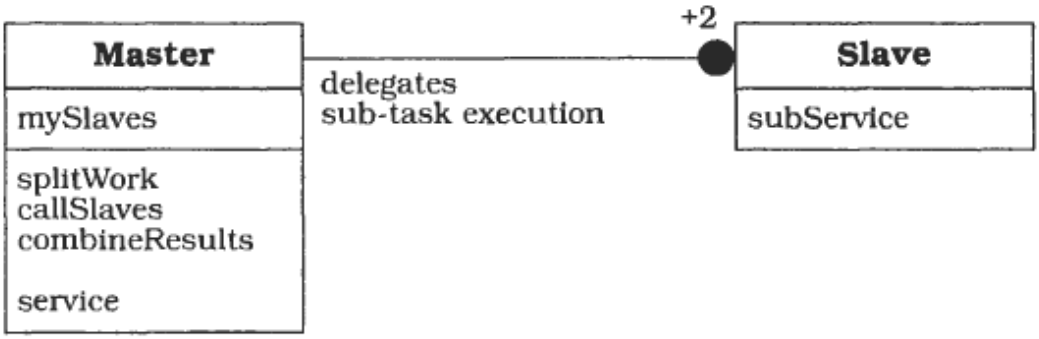
Structure:

- The master component provides a service that can be solved by applying the 'divide and conquer' principle.
- It offers an interface that allows clients to access this service.
- Internally, the master implements functions for partitioning work into several equal sub-tasks, starting and controlling their processing, and computing a final result from all the results obtained.
- The master also maintains references to all slave instances to which it delegates the processing of sub-tasks.
- The slave component provides a sub-service that can process the sub-tasks defined by the master. Within a Master-Slave structure, there are at least two instances of the slave component connected to the master.

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

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

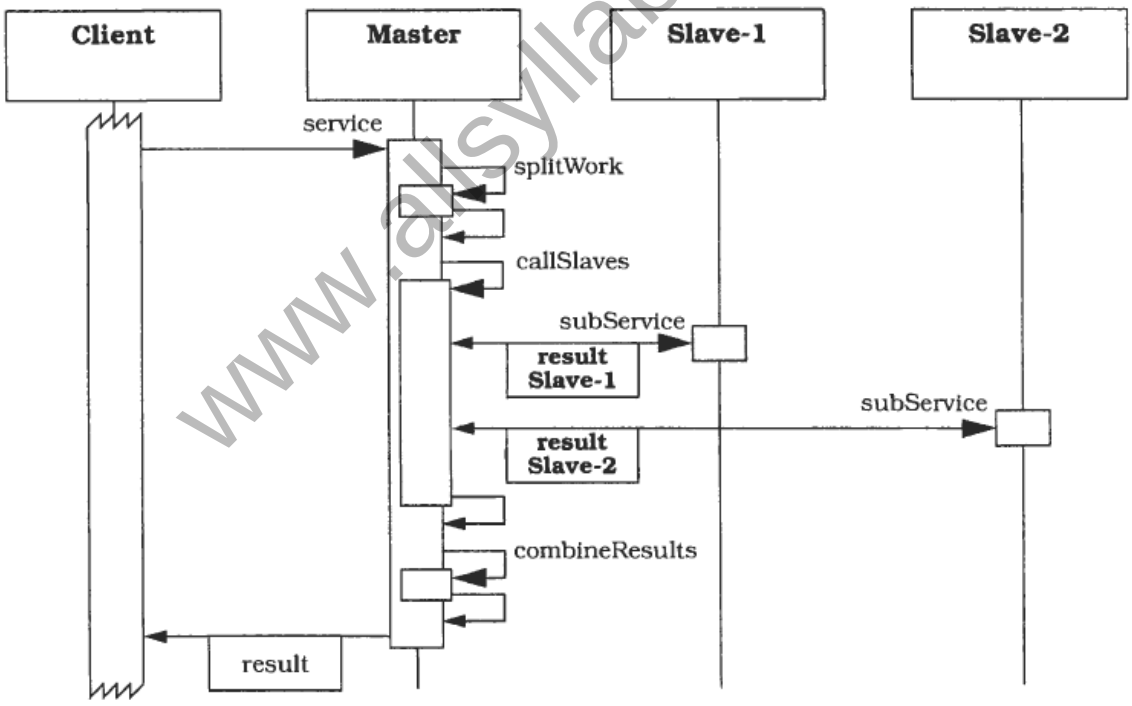
Relation between Master and Slave objects:



Dynamics

Scenario: Slaves are called one after the other

- The scenario comprises of ***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

Step1: ***Divide work***

- Specify how the computation of the task can be split into a set of equal sub-tasks.
- Identify the sub-services that are necessary to process a sub-task.
- In the travelling-salesman problem

  - Partition the problem so that a slave is provided with one round trip at time and computes its cost.
  - However, for a machine like the CM5 with SPARC node processors, such a partitioning might be too fine grained.
  - The costs for monitoring these parallel executions and for passing parameters to them decreases the overall performance of the algorithm instead of speeding it up.
- A more efficient solution is to define sub-tasks that identify the shortest trip of a particular subset of all trips.
- This solution also takes account of the fact that there are only sixty-four processors available on our CM5.
- The number of available processors limits the number of sub-tasks that can be processed in parallel.
- To find the number of trips to be compared by each sub-task, we divide the number of all trips to be compared by the number of available processors.

Step2: ***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.
- In our example, each sub-task returns only the shortest trip of a subset of all trips to be compared.
- We must still identify the shortest trip of these

Step 3: **Specify the cooperation between master and slaves**

- Define an interface for the sub-service identified in step 1
- It will be implemented by the slave and used by the master to delegate the processing of individual sub-tasks.
- There are two options for passing subtasks from master to slaves
  - include them as a parameter when invoking the sub-service, or
  - define a repository where the master puts sub-tasks and the slaves fetch them.

When processing a sub-task:

- individual slaves can work on separate data structures or
- all slaves can share a single data structure

Slaves may return the result of their processing:

- explicitly as a return parameter, or
- they may write it to a separate repository from which the master retrieves it.
- For the traveling-salesman program we let each slave operate on its own copy of the graph that represents all cities and their connections
- The copies are created when instantiating slaves

Step 4: **Implement the *slave components according to the specifications developed in the previous step***

Step 5: ***Implement the master according to the specifications developed in step 1 to 3.***

There are two options for dividing a task into sub-tasks.

- The first is to split work into a fixed number of sub-tasks.
- Most applicable if the master delegates the execution of the complete task to the slaves.
- This might typically occur when the Master-Slave pattern is used to support fault tolerance or computational accuracy applications.
- The second option is to define as many sub-tasks as necessary, or possible.
- For example, the master component in our traveling-salesman program could define as many sub-tasks as there are processors available.

- The code for launching the slaves, controlling their execution and collecting their results depends on many factors.
- The master computes a final result with help of the results collected from the slaves.

Consequences – Advantage

The Master-Slave design pattern has some important **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. Clients are not affected by such changes.
- *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.

Liabilities

- ***Feasibility.*** Master-Slave architecture is not always feasible. You must partition work, copy data, launch slaves, control their execution, wait for the slave's results and compute the final result. All these activities consume processing time and storage space.
- ***Machine dependency.*** The Master-Slave pattern for parallel computation strongly depends on the architecture of the machine on which the program runs.
- ***Hard to implement.*** Implementing Master-Slave is not easy, especially for parallel computation.
- ***Portability.*** Because of the potential dependency on underlying hardware architectures, 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.
- For example, not all clients may be authorized to use the services of a component, or to retrieve particular information that a component supplies.
- And, sometimes we want to use the services of a remote service provider
- This should be transparent to the client who requested the service

Types:

- The ***Proxy design pattern*** makes the clients of a component communicate with a representative rather than to the component itself.
- The ***Facade pattern*** provides a uniform interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.
- The ***Iterator pattern*** provides a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

Design Pattern: Master-Slave

Alternative view

This pattern belongs to the category of Organization of Work. Work is partitioned into several identical subtasks. Each subtask is allocated to independent components (slaves) by a special component (master). Master collates all the results returned by the slave components.

Context and Forces to Balance

The context here is when we need to partition work into semantically-identical sub-tasks.

Master should hide the presence of slaves from Clients. The algorithm for partitioning the task should not affect Clients or Slaves. Similarly, computing the final result by collating all the results from the Clients should be transparent to Clients and Slaves.

In some cases where computational accuracy is mission-critical, Slaves must be implemented differently, but with identical functionality.

Also, Slaves need to be coordinated to achieve system behavior.

Solution

A master component distributes work to identical slave components and computes a final result from the results these slaves return.

There are three application areas in which this can be applied.

1. Fault tolerance, where the execution of a service is delegated to several replicated implementations
2. Parallel computing, where a complex task is divided into a fixed number of identical sub-tasks that are executed concurrently
3. Computational accuracy, where the execution of a service is delegated to several different implementations

Structure

Master component provides an interface that allows clients to access the service achieved by utilizing the Slave components. The responsibility of the Master component is to Partition the work among several slave components. It should then start the execution of slaves. After all slaves return their results it should compute a result and return it to the Client.

The Slave component implements the sub-service used by the master.

Implementation

The first step is to divide work by partitioning the task to multiple subtasks. Then we need to identify sub-services needed to accomplish subtasks.

The next step is computation of final result to send to client, by combining the results returned by the various slaves.

Next we will need to specify the cooperation between master and slaves. We have to specify the interface for slaves. We have to decide how data will be passed to the slaves and how the slaves will return the results to the Master. There are two options here: either we can use a shared repository or we can pass the data as parameter.

The next step is to implement the slave components. Here, if you want to achieve concurrency, all clients are identical, but exist in their own thread/process/hardware. But if you are using voting tactic, different implementation for each slave will be required.

The last step is to implement the master. First functionality is partitioning the service to subtasks. Here we could have a fixed number of subtasks, or it can be dynamic, depending on the availability of slaves. Another functionality of the Master component is to manage the lifecycle of slaves. We have to add functionality to launch slaves and coordinate operations of slaves. Then the Master has to collect results and compute final result to ship to client. Last important functionality to implement is error handling.

Variant

1. Master-Slave for fault tolerance: here the Master delegates entire task to all slaves, so no need for partitioning of tasks. The Slaves are usually identical . The result of first slave is sent to client, and typically the Master ignores results of other slaves. There has to be a mechanism for detection of failure of slaves, e.g., through timeouts.
2. Master-Slave for parallel computation: Here task is partitioned by master and sent to identical slaves. All the slaves work on subtasks in parallel but with different part of the data. Sometimes, they can work with shared data. The Master collates the results from the all the slaves and sends it to client. This is the most common use of the Master-Slave pattern.
3. Master-Slave for computational accuracy: The main difference here is that we need different implementation on each slave, to implement the voting tactic. The final result can be average of all results from slave, or the maximum, or the most common value
4. Slaves as Processes: Here Slaves will run separately in their own process space, and possibly in multiple hardware
5. Master-Slave with slave coordination: If computation is in multiple stages and requires slaves to exchange huge volumes of data, the control logic for coordination can be placed in the slave, and the Master has coordination responsibilities

Benefits

1. Exchangeability and extensibility: By using an abstract slave class, we can insulate any changes to the Master, when slave implementation has to be modified. Clients, of course, are not affected as they do not directly deal with the Slaves.

2. Separation of concerns: We place the core functionality with slaves, and the Master performs all tertiary duties, like, task partitioning, delegating work to slaves, collecting results from slaves, collating result and ship to client and also handle errors
3. Efficiency: The pattern exploits concurrency, so performance can be enhanced

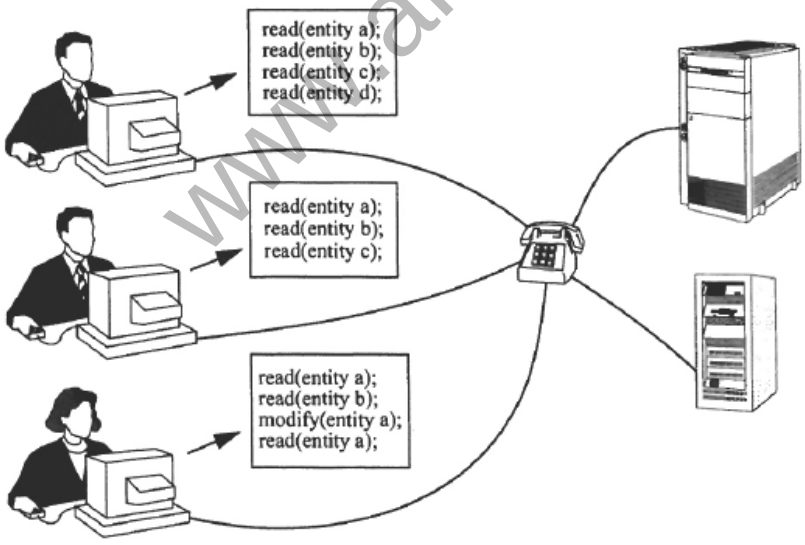
Liabilities

1. Feasibility: Not all tasks can be partitioned, so the applicability of this design pattern is limited.
2. System dependency affects portability: This is especially true in parallel computation, where the hardware architecture and topology dictates design.
3. Implementation complexity: The added task of partitioning of tasks, and the coordination of slaves could be complex. Also error-handling might not be straight-forward.
4. Master-slave communication latency can be an issue, especially in real-time systems.

Proxy Design Pattern

The ***Proxy design pattern*** makes the clients of a component communicate with a representative rather than to the component itself. Introducing such a placeholder 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. However, we do not want to burden the engineer's application code with such optimization. The presence of optimization and the type used should be largely transparent to the application user and programmer.

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

Problem*:*

A solution to such a design problem has to balance some or all of the following forces:

- Accessing the component should be run-time-efficient, cost-effective, and safe for both the client and the component.
- Access to the component should be transparent and simple for the client.
- The client should be well aware of possible performance or financial penalties for accessing remote clients.

Solution

- Let the client communicate with a representative rather than the component itself.
- This representative-called a proxy
- 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

- The original implements a particular service. Such a service may range from simple actions like returning or displaying data to complex data-retrieval functions or computations involving further components.

Class Original	***Collaborators*** -
Responsibilities • Implements a particular service.	

- The client is responsible for a specific task. To do its job, it invokes the functionality of the original in an indirect way by accessing the proxy.
- The client does not have to change its calling behavior and syntax from that which it uses to call local components.

Class Client	***Collaborators*** • Proxy
Responsibilities • Uses the interface provided by the proxy to request a particular service. • Fulfills its own task.	

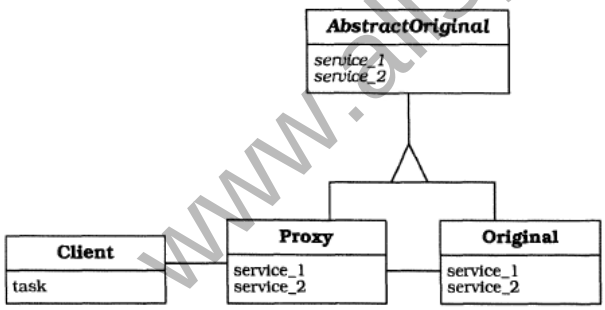
- the proxy offers the same interface as the original, and ensures correct access to the original.
- To achieve this proxy maintains a reference to the original it represents.

Class Proxy	Collaborator • Original
Responsibilities • Provides the interface of the original to clients. • Ensures a safe, efficient and correct access to the original.	

- The abstract original provides the interface implemented by the proxy and the original.
- In a language like C++, with no notable difference between subtyping and inheritance, both the proxy and the original inherit from the abstract original.

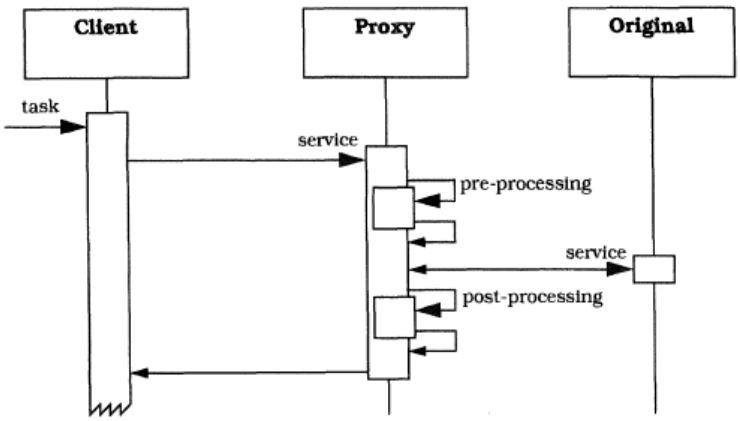
Class *AbstractOriginal*	Collaborators -
Responsibilities • Serves as an abstract base class for the proxy and the original.	

The Relationship between the classes



Dynamics

Scenario: Client with the help of proxy getting service of the original



- 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. This pre-processing involves actions such as looking up the address of the original, or checking a local cache to see if the requested information is already available.
- 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

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.

If identical interfaces for the proxy and the original are not feasible you can use an adapter for interface adaptation.

Adapting the proxy to the original's interface retains the client with the illusion of identical interfaces, and a common base class for the adapter and the original may be possible again.

3. ***Implement the proxy's functions.***
4. ***Free the original and its clients 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 clients.***

Replace them by analogous relationships to the proxy.

Consequences - Advantages

The Proxy design pattern has some important **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.

Note however that a straightforward implementation of a remote proxy still has the location of the original hard-wired into its code.

- ***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.

Liabilities

- ***Less efficiency due to indirection.***

All proxies introduce an additional layer of indirection. But, usually negligible compared to the cleaner structure of clients and gain of efficiency through caching.

- ***Overkill via sophisticated strategies.***

Be careful with intricate strategies for caching or loading on demand-they do not always pay.

This occurs when originals are highly dynamic, for example in an airline reservation or other ticket booking system.