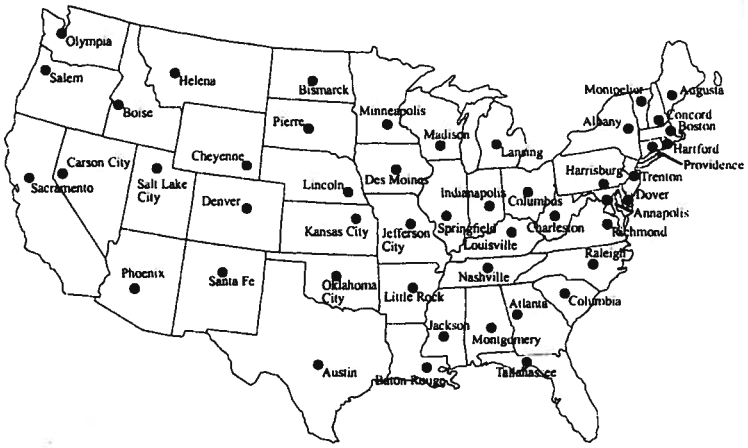
Master-Slave

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 The traveling-salesman problem is well-known in graph theory. 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—there are approximately $6.0828 * 10^{62}$ different trips that connect the state capitals of the United States! Generally, the solution to the traveling-salesman problem with n locations is the best of $(n-1)!$ possible routes. Since the traveling-salesman problem is NP-complete [GJ79], there is no way to circumvent this high complexity if the optimal solution must be found.

Most existing implementations of the traveling-salesman problem therefore approximate the optimal solution by only comparing a fixed number of routes. One of the simplest approaches is to select routes to compare at random, and hope that the best route found approximates the optimal route sufficiently. We should make sure

however that the routes to be investigated are chosen in a random and independent fashion, and that the number of selected routes is sufficiently large.

Context Partitioning work into semantically-identical sub-tasks.

Problem 'Divide and conquer' is a common principle for solving many kinds of problems. Work is partitioned into several equal sub-tasks 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 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, for example to increase computational accuracy.
- 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.* The execution of a service is delegated to several replicated implementations. 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. The final result is built with the help of the results obtained from processing these sub-tasks.

- *Computational accuracy.* The execution of a service is delegated to several different implementations. Inaccurate results can be detected and handled.

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. This simple-minded implementation uses an early version of the object-oriented parallel programming language pSather [MFL93]. 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. Each slave takes a number of trips to be compared as input, randomly selects these trips and returns the shortest trip found. 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. In other words, the slaves provide local optima that the master resolves to a global optimum. ❑

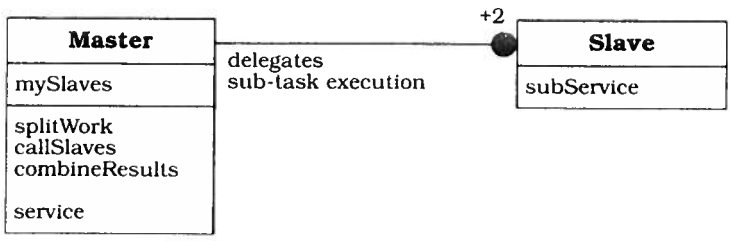
Structure The *master* component provides a service that can be solved by applying the 'divide and conquer' principle. It offers an interface that allows clients to access this service. Internally, the master implements functions for partitioning work into several equal sub-tasks, starting and controlling 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.	

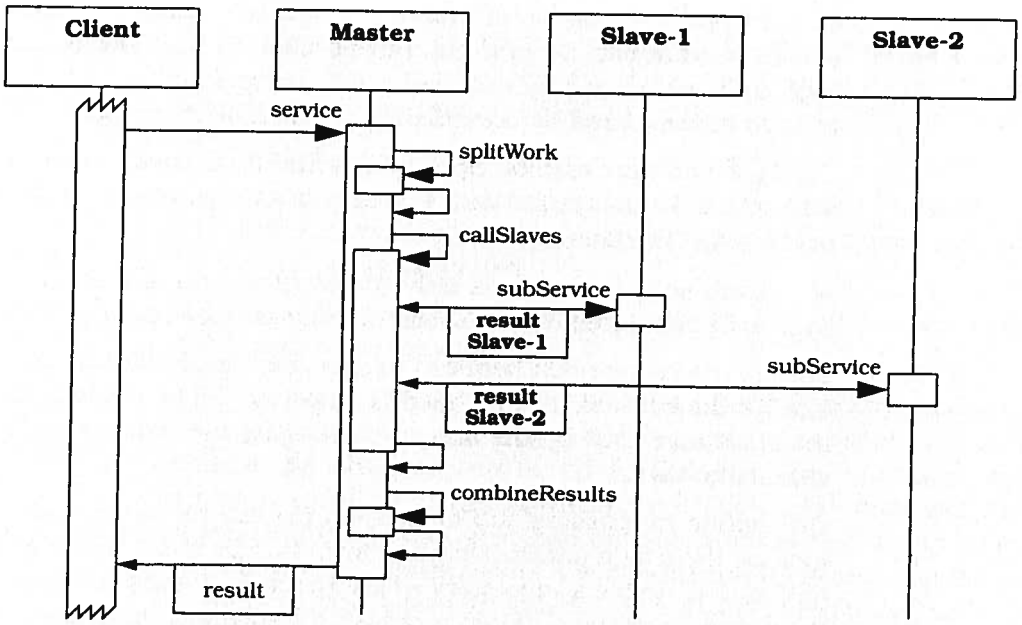
Class Slave	*Collaborators* -
Responsibility • Implements the sub-service used by the master.	

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



Dynamics In the following scenario we assume, for simplicity, that slaves are called one after the other. However, the Master-Slave pattern unleashes its full power when slaves are called concurrently, for example by assigning them to several separate threads of control. 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 The implementation of the Master-Slave pattern follows five steps. Note that these steps abstract from specific issues that need to be considered when supporting the application of the pattern to the special cases of fault tolerance, parallel computation, and computational accuracy, or when distributing slaves to several processes or threads. These aspects are addressed in the Variants section.

1 *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.

➥ For our parallel traveling-salesman program we could 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. ❑

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.

➥ 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.❑

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.

One option for passing sub-tasks from the master to the slaves is to include them as a parameter when invoking the sub-service. Another option is to 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.

Which of these options are best depends on many factors; for example, the costs of passing sub-tasks to slaves, of duplicating data structures, and of operating on a shared data structure with several slaves. The original problem also influences the decisions to be made. When slaves modify the data on which they operate, you need to provide each slave with its own copy of the original data structure. If they do not modify data, all slaves can work on a shared data structure, for example when implementing matrix multiplication.

➥ For the traveling-salesman program we let each slave operate on its own copy of the graph that represents all cities and their connections. We will create these copies when instantiating the slaves. The alternative—having the slaves read from one shared graph representation—was not chosen since such a communication load on the CM5 internal network would reduce the performance of our application considerably.

The interface of the slave to the master is defined by a function that takes the number of random routes to be evaluated as an input parameter. The function returns the optimal route found, which is represented by an instance of class TOUR.

```
random_perms(numberPerms : INT) : TOUR
```

The term perms in random_perms() stands for permutations, since we represent round trips as permutations of the n nodes that stand for the n cities to be visited. ❑

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

➥ The class TSP is the design center of our small applications. It includes a constructor, functions to create a random trip and to update the shortest trip found so far, and the random_perms() function specified in the previous step. The class COMPLETE_GRAPH represents the graph structure on which instances of TSP operate. The class RANDOM represents a random number generator. The code is not complete, but is an excerpt from a working application.

```
class TSP is
    -- Data structures
    best_tour, current_tour     : TOUR;
    graph                       : COMPLETE_GRAPH;
    random                      : RANDOM;
    -- Constructor for the slave that initializes
    -- the return value, creates the graph structure,
    -- and creates the random number generator.
    create() : TSP is
        res             := new;
        res.graph       := COMPLETE_GRAPH::create;
        res.random      := RANDOM::create;
    end; -- create
    -- Construct a number of randomly selected tours and
    -- return the tour with the lowest costs
    random_perms(numberPerms : INT) : TOUR is
        i : INT := 1;
        while i <= numberPerms loop
            construct_random_tour;
            update_optimum;
            i := i+1;
        end;  -- loop
        res := best_tour;
    end; -- random_perms
    -- Construct a new random tour and calculate its costs
    construct_random_tour is -- not shown here
    end; -- construct_random_tour
```

```
        -- Update the optimal tour if the currently evaluated
        -- tour is better than the current optimum
        update_optimum is
            if current_tour.cost < best_tour.cost then
                best_tour     := current_tour;
            end; -- if
        end; -- update_optimum
    end; -- class TSP
```

Note that the assignment in `update_optimum` assumes either deep-copy semantics, or that `current_tour` will refer to a new `TOUR` object after the assignment. Otherwise, `construct_random_tour()` corrupts `best_tour` when modifying `current_tour`. The original program solved the problem by swapping the two `TOUR` objects to which `best_tour` and `current_tour` referred. ❑

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. This is 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, or if the amount of parallel work is always fixed and known a priori. 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 exchange of algorithms for subdividing a task can be supported by applying the Strategy pattern [GHJV95]. We discuss further issues you should consider in the Variants section.

The code for launching the slaves, controlling their execution and collecting their results depends on many factors. Are the slaves executed sequentially, or do they run concurrently in different processes or threads? Are slaves independent of each other, or do they need coordination? We give more details about this in the Variants section.

The master computes a final result with help of the results collected from the slaves. This algorithm may follow different strategies, as described in the Variants section. To support its dynamic exchange and variation, you can again apply the Strategy pattern [GHJV95].

You also must deal with possible errors, such as failure of slave execution or failure to launch a thread. Details are discussed in the Variants section.

There is only one master component within a Master-Slave structure. You can apply the Singleton pattern [GHJV95] to ensure this property.

➥ In the traveling salesman program we represent the master with an object of class `CM5_TSP`. It offers a function `best_tour()` to its clients which returns the best round trip visited by the whole Master-Slave structure. The `best_tour()` function takes the number of routes to be generated and the number of processors to use as parameters.

The function `distribute()` copies the graph and some additional data structures to all processors. The implementation we show works sequentially. '`@j`' means 'do this operation on processor j'. The function `distribute()` creates as many new slaves as there are processors available. The function `random_perms()` launches the slaves. The function `update_optimum()` selects the optimal route from the local optima returned by the slaves.

Our strategy for coordinating the slaves is to start them asynchronously and to synchronize them later, in particular when we want to select the best trip found. To implement this behavior we use the 'future' principle. A future is a variable that defines a value that is computed asynchronously in a different process or thread of control. Synchronization is achieved when the variable is accessed later. Since pSather supports futures, we use an array of futures for slaves to coordinate their parallel execution. For reasons of brevity we do not illustrate object creation. For more details on the pSather version we use in our example, see [Lim93].

```
class CM5_TSP is
     -- Data structures. Shared variables in pSather
     -- correspond to static members in C++
     shared n : INT              -- Number of Cities
     shared P : INT;             -- Number of processors
     shared T : ARRAY{TSP};      -- The slave array
     shared best_tour : TOUR     -- The best round trip
```

```
    -- Assign a slave to each available processor
    distribute is
        -- Create the slave instances
        i : INT := 1;
        while i <= P loop
            -- initializes T[j];
            copy_graph()@j;
            i := i+1;
        end; -- loop
    end; -- distribute
    -- Launch the slaves
    random_perms(t : INT) is
        i, j, jobs_per_proc : INT;
        -- Calculate how many tours each slave must visit
        -- Assume that P divides t
        jobs_per_proc     := t/P;
        -- Define a monitor
        m := MONITOR{TOUR}:=MONITOR{TOUR}::new;
        -- Launch each slave at its processor
        i := 1;
        while i <= P loop
            m :- T[i].random_perms(jobs_per_proc)@i;
            i := i + 1;
        end; -- loop
        -- wait until the slaves finish with their
        -- computation and take the results of the slaves
        -- in whatever order they are returned
        j := 1;
        while j <= P loop
            current_tour := m.take;
            update_optimum();
            j := j + 1;
        end; -- loop
    end; -- random_perms
    -- Select the optimal tours from the trips the slaves
    -- returned
    update_optimum is -- not shown here
    end; -- update_optimum
    -- Return the optimal tour from t randomly created
    -- ones with help of P slaves.
    best_tour(t, p : INT): TOUR is
        P := p;
        -- Create the slaves, launch them, determine
        -- the best trip visited, and return this tour
        -- to the client calling the master
        distribute;
        random_perms(t);
        update_optimum;
        res := best_tour;
    end; -- best_tour
end; -- class CM5_TSP                                    ❑
```

Variants There are three application areas for the 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. As soon as the first slave terminates, the result produced is returned to the client of the master. Fault tolerance is supported by the fact that as long as at least one slave does not fail, the client can be provided with a valid result. The master can handle the situation in which all slaves fail, for example by raising an exception or by returning a special 'Exceptional Value' [Cun94] with which the client can operate. The master may use time-outs to detect slave failure. However, this variant does not help with the situation in which the master itself fails—it is the critical component that must 'stay alive' to make this structure work.

Master-Slave for parallel computation. The most common use of the Master-Slave pattern is for the support of 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. The master builds the final result from the results obtained from the slaves. The master contains the strategies for dividing the overall task and for computing the final result.

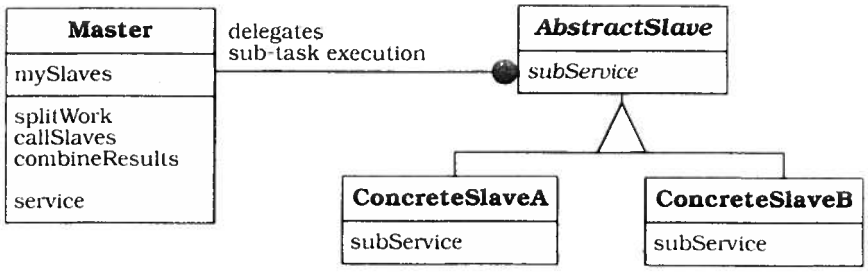
The algorithm for sub-dividing the task and for coordinating the slaves is strongly dependent on the hardware architecture of the machine on which the program runs. On distributed memory machines with general-purpose processors, for example, the granularity is usually larger than on SIMD (single instruction multiple data) machines. Other aspects that govern the algorithm are the machine's topology and the speed of its processor interconnections. The cooperation between the master and the slaves also depends on aspects such as the existence of shared or distributed memory for machines. The division of work is further influenced by issues listed in the *Slave as Threads variant* (see below), and the cooperation between master and slaves by issues listed in step 3 of the Implementation section.

Before the master can compute the final result it must wait for all slaves to finish executing their sub-tasks. To free the master from the task of synchronizing each slave individually, [KSS96] introduces the concept of a *barrier*. A barrier is initialized with the slaves on whose termination the master waits. It then suspends the execution of the master until all the slaves it controls have terminated. Our pSather

example, in contrast, works in an incremental fashion—whenever a slave terminates the `random_perms()` method takes its result.

Master-Slave for computational accuracy. In this variant the execution of a service is delegated to at least three different implementations, each of which is a separate slave. The master waits for all slaves to complete, and votes on their results to detect and handle inaccuracies. This voting may follow different strategies. Examples include that in which the master selects the result that is returned by the greatest number of slaves, the average of all results, or the use of an Exceptional Value [Cun94] in the case in which all slaves produce different results.

To provide different slave implementations, we can extend the structure of the Master-Slave pattern with an additional abstract class. This defines an interface common to all slave implementations. Different slave implementations are then derived from this abstract base.



Further variants exist for implementing slaves:

Slaves as Processes. To handle slaves located in separate processes, you can extend the original Master-Slave structure with two additional components [Bro96]. The master includes a *top component* that keeps track of all slaves working for the master. To keep the master and the top component independent of the physical location of distributed slaves, remote proxies (263) represent each slave in the master process. You can apply the Forwarder-Receiver (307) or Client-Dispatcher-Server pattern (323) to implement the interprocess communication.

Slaves as Threads. In this variant, every slave is implemented within its own thread of control [KSS96]. In this variant the master creates the threads, launches the slaves, and waits for all threads to complete before continuing with its own computation. The Active Object pattern [Sch95] helps in implementing such a structure.

In this variant the master must deal with two problems: what happens if a thread cannot be created, and how many threads should be created? A solution to the first problem is to call the slave's services directly, without launching them in a separate thread. Performance will suffer, but the result will be correct. The optimal number of threads depends on the number of processors available and on the amount of work required from each thread. Too many threads incur overheads in their creation and destruction, as well as in memory consumption. [KSS96] suggests experimenting with different strategies, starting with 'a few more threads than the number of processors'.

Master-Slave with slave coordination. The computation of a slave may depend on the state of computation of other slaves, for example when performing simulation with finite elements. 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.

There are two ways of implementing such a behavior. Firstly, you can include the control logic for slave coordination within the slaves themselves. This frees the master from the task of implementing this coordination, but may decrease the performance of the overall structure. Slaves will stop their execution independently and may idle until the slaves on which they depend are ready for coordination.

The second option is to let the master maintain dependencies between slaves and to control slave coordination. At regular time intervals the master suspends all slaves, retrieves the current state of their computation, forwards this data to all slaves that depend on this data, and resumes the execution of all slaves.

Known Uses [KSS96] lists three concrete examples of the application of the Master-Slave pattern for parallel computation:

- 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 8×8 pixel block in an image. Each block can be computed by a separate slave.
- Computing the cross-correlation of two signals. This is done by iterating over all samples in the signal, computing the mean-square distance between the sample and its correlate, and summing the distances. We can partition the iteration over the samples into several parts and compute the square distance and its sums separately for each partition. The final sum is computed by summing all sums from these partitions. Each partial summing can be performed by a separate slave. A master component defines the partitions, launches the slaves, and computes the final sum.

The **workpool model** described in [KR96] applies the Master-Slave pattern to implement process control for parallel computing, based on the principles of Linda [Gel85]. A programmer can assign a number of so-called workers to a workpool. Each worker offers the same services and is implemented in a separate process or thread. Clients send requests to the workpool, which handles these requests with help of its associated workers. The request itself is a function whose execution should be parallelized with help of the workers, such as matrix multiplication. This function corresponds to the master component in the Master-Slave pattern.

The concept of **Gaggles** [Bl93] 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. When receiving a service request from a client, the gaggle forwards this request to one of the service objects it includes. Each of these service objects can be atomic, which means it executes the service and delivers a result, or another gaggle which itself represents a set of replicated service objects.

[Bro96] lists several applications of the Master-Slave design pattern, all of which focus on distributed slaves. These include the distributed design rule checking system **Calibre™ DRC-MP** and the **CheckMate** IC verification tool, both from Mentor Graphics.

Factoring large numbers into *prime factors* can also be done in a 'divide and conquer' fashion. As this problem is central to cryptography, of great interest to governments, and requires vast computing resources, it has been carried out over the Internet. One site did the subdivision and sent sub-tasks to people willing to provide computing time and the use of their machines.

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. Clients are not affected by such changes. If they are implemented with the Strategy pattern [GHJV95], the same holds true when changing the algorithms for allocating sub-tasks to slaves and for computing the final result.

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. However, you must always consider the costs of parallel computation (see below).

The Master-Slave pattern suffers from three **liabilities**:

Feasibility. A 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—see the Variants section for details. This may decrease the changeability and portability of a Master-Slave structure.

Hard to implement. Implementing Master-Slave is not easy, especially for parallel computation. Many different aspects must be considered and carefully implemented, such as how work is subdivided, how master and slaves should collaborate, and how the final result should be computed. You also must deal with errors such as the failure of slave execution, failure of communication between the master and slaves, or failure to launch a parallel slave. Implementing the Master-Slave pattern for parallel computation usually requires sound knowledge about the architecture of the target machine for the system under development.

Portability. Because of the potential dependency on underlying hardware architectures, Master-Slave structures are difficult or impossible to transfer to other machines. This is especially true for the Master-Slave pattern for parallel computation, and similarly for our simple traveling-salesman program tuned for the CM5 computer.

See also An earlier version of this pattern appeared in [PLoP94].

The *Master-Slave Pattern for Parallel Compute Services* [Bro96] provides additional insights for implementing a Master-Slave structure. It differs from the structure described here, as it concentrates on describing the *Slaves as Processes* variant.

The book *Programming with Threads* [KSS96] describes the Slaves as Threads variant in detail.

Object Group [Mat96] is a pattern for group communication and support of fault tolerance in distributed applications. It corresponds to the *Master-Slave for fault tolerance* variant and provides additional details for its implementation. The Object Group pattern provides a local surrogate for a group of replicated objects distributed across networked machines. A request is broadcast to all objects of the group. The request will succeed as long as one group member terminates successfully.

Credits We thank Ken Auer, Norbert Portner, Douglas C. Schmidt, Jiri Soukup, and John Vlissides for their valuable criticism and suggestions for improvement of the [PLoP94] version of this pattern. Special thanks go to Phil Brooks and Jürgen Knopp for their contribution to this new version.

3.4 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.

In this section we describe one design pattern that helps to protect access to a particular component:

- The *Proxy* design pattern (263) 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.

[GHJV95] also describes the Proxy pattern. Our description differs in that it separates the general principle that underlies the pattern from its concrete application cases, which we describe as variants. We also provide several new variants of Proxy that are not covered by the Gang-of-Four version.

The Proxy pattern is widely applicable. Almost every distributed system or infrastructure for distributed systems uses the pattern to represent remote components locally, for example OMG-Corba [OMG92]. A more recent application of Proxy is the World Wide Web [LA94], where it is used to implement the proxy servers.

Two other patterns described in [GHJV95] also belong to this category—Facade and Iterator:

- 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.

Like the Proxy pattern, both the Facade and Iterator patterns are widely applicable.

Facade shields the components of a subsystem from direct access by their clients. Vice-versa, clients do not depend on the internal structure of the subsystem. A facade component routes incoming service requests to the subsystem component that implements the service. Facade is therefore of larger granularity than Proxy, which guards access to single component.

Iterators are offered by almost every container class in an object-oriented program or class library. An iterator defines the order in which clients can traverse and access the elements of a container. For example, to access all elements in a binary tree, you can define iterators for pre-order, in-order and post-order traversal.