

Client-Server Interlock State Machine Pattern

Contributed by Leon Starr

One object, let's call it the *server*, performs a task or makes a resource available to one or more *client* objects. Synchronization is required as follows: 1) A client must wait if the server is busy at the time of request 2) The server must wait if there are no clients requesting service 3) If the server is idle and a client requests service, the server must be activated 4) If multiple clients request service simultaneously, service will be provided serially based on some criteria such as "first come - first serve". Since a single server is dedicated to a pool of one or more clients, there is no need for the assigner mechanism described in [Mellor 1].

Here's the key trouble to avoid: 1) Clients don't hang: every client is serviced as soon as possible. 2) The server doesn't hang: it never goes idle leaving clients to wait pointlessly.

The terms *server* and *client* refer to individual objects in the context of this document. Class references are made using initial caps like this: "The Server class has three states"

Applications

Don't let the phrase "Client-Server" lull you into thinking that this pattern is relevant only to data processing. Mechanical collision avoidance (safety interlock) is a key application. A single device in the server role, for example, might be locked in a safe zone until one or more other client devices have cleared the area.

In general, though, anytime you have one object that must serve or otherwise coordinate an activity with another single object or pool of objects, this pattern should be considered.

Solving the problem

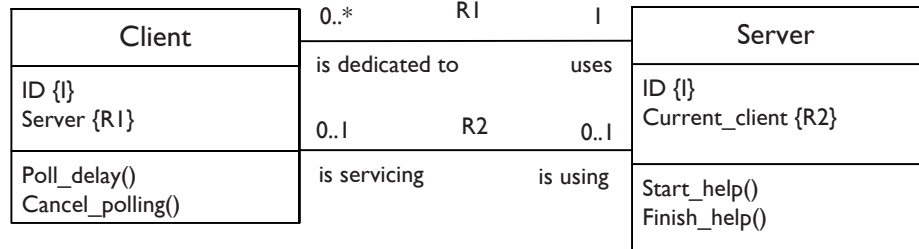
The final pattern is defined at the end of this note followed by an example application and distillation of the principles. Skip ahead if you like, but to fully appreciate the solution, it might be interesting to compare some antipatterns.

Antipattern #1 - Client polls the server

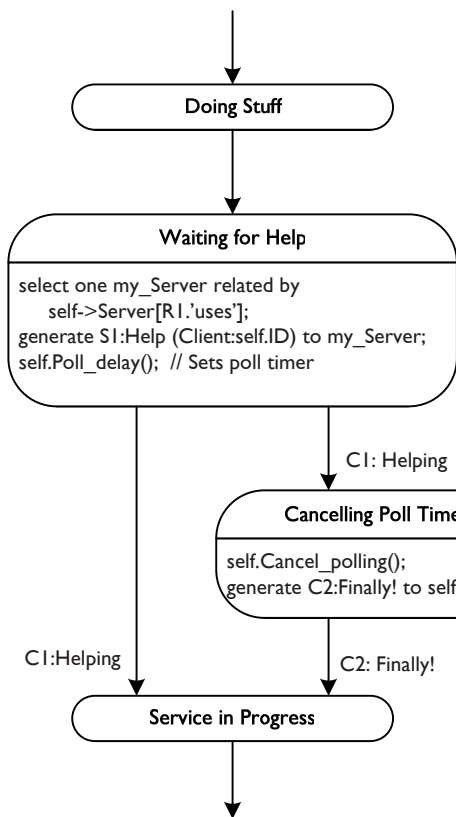
With the polling antipattern each client attempts to get its server's attention. If the server is busy, a timer is set and the client tries again later. Pretty poor service if you ask me!

Class diagram

The R1 association establishes a pool of potential clients for each server. R2 remembers the client, if any, currently being serviced.



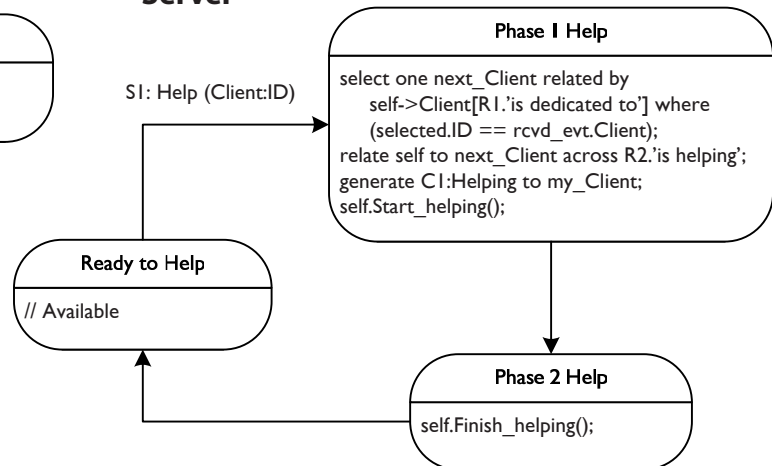
Client



How it works

When a client enters **WAITING FOR HELP**, it signals its server with S1:Help including the client object ID. If the server is available, it links itself to the client on R2, responds with C1:Helping, and begins processing. The client proceeds to **SERVICE IN PROGRESS**. The server moves through its work states, signals completion to the client and then waits in **READY TO HELP** for another client to signal. If the server is not in **READY TO HELP** when a client signals, the signal is ignored (defined in the Server class state table for **PHASE 1 HELP** and **PHASE 2 HELP**). Note that as this is xtUML all state actions are executed upon entry.

Server



Anti-pattern #1 - Why it's a bad solution

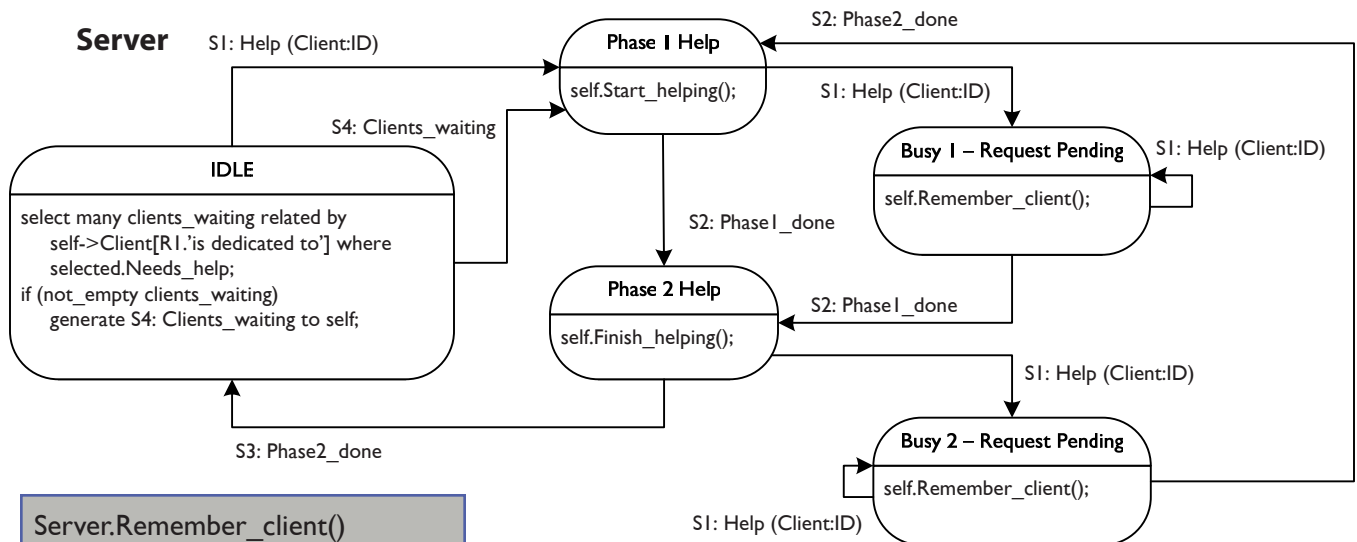
Two key problems with this approach are 1) unnecessary polling and 2) reliance on event handling to choose the next client.

Polling requires extra model gunk in the Client statechart for setting and canceling timers. The runtime activity can be excessive, especially if you have a large number of clients — each with its own timer. Polling is appropriate when you want to model an inherently periodic task like sampling or feedback control. As an implementation trick, however, polling is unnecessary in the asynchronous world of xtUML.

The other problem with this antipattern is the way the server chooses the next client to service. In fact, the server isn't choosing anything. The model architecture simply delivers signals based on its own internal event prioritization scheme and the order of timer expiration. If requirements dictate a clear first in first out or some other prioritization scheme we are out of luck with this particular antipattern. So let's try something else.

Antipattern #2 - Interruptible Server

Instead of polling we can arrange for the server to react immediately to client requests regardless of current server state.



Server.Remember_client()

```

select one current_Client related by
self->R2.'is helping';
current_Client.Needs_help = true;
  
```

Server.Help_client()

```

select one my_Client related by
self->Client[R2.'is servicing'];
my_Client.Needs_help = false;
self.Start_help();
  
```

Server.Choose_client()

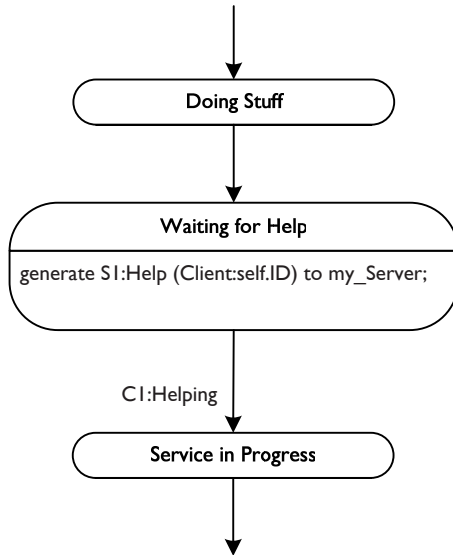
```

// Insert any algorithm here that selects all of the
// clients across R1 where selected.Needs_help is true and
// then picks a single chosen_Client based on some criteria.

relate chosen_Client to self across R2.'is servicing';
  
```

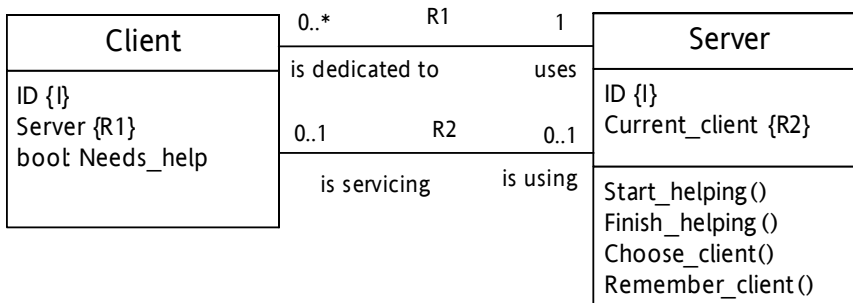
Antipattern #2 - Continued

Client



How it works

Here the client simply asks for help and lets its server do all of the work. And a lot of work it is! The server not only performs its help service, it must also continually note any waiting clients. To do this, the server manages the boolean `Client.Needs_help` attribute on the relevant client object. A server must respond to the help signal in every one of its states. When it is in **PHASE I HELP** of processing, it must move to **BUSY I - REQUEST PENDING**. Why not just return to **PHASE I HELP**? The `Start_helping()` operation would be re-initiated upon entry, so we can't go back.

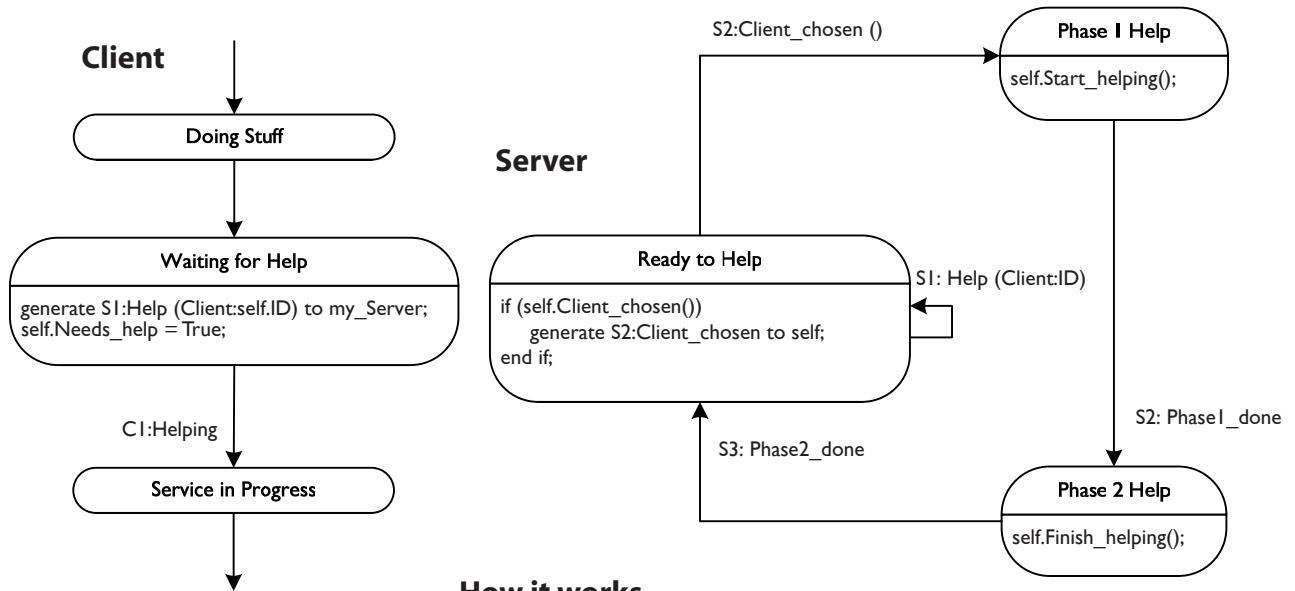


Antipattern #2 - Why it's a bad solution

All those extra `S1:Help` transitions are bad because of all the excess graphics gunk and because we lose site of the Server lifecycle. Every class should have a straightforward lifecycle with a tight scope. If a statechart fields the same event in every state, you know something isn't right.

Final - The Client-Server Interlock Pattern

When the server is busy, it should not be interrupted by client requests. The server can't do anything for a client anyway if it is busy. The server should focus on the help task and check for waiting clients only when finished servicing the current client. Each client is responsible for simultaneously signaling and posting persistent data that the server can consult later if the signal is ignored in a busy state.



How it works

When the client signals for help it also sets its Needs_help boolean attribute. If the server is not waiting in **READY TO HELP**, the signal will be ignored. The loss of this signal is not a problem since, when the server enters **READY TO HELP**, it invokes `Server.Client_chosen()` anyway. This operation searches all related clients with the Needs_help boolean set to true. If none are found, the server waits in **READY TO HELP** for the next `S1:Help` signal to arrive and trigger another search.

```

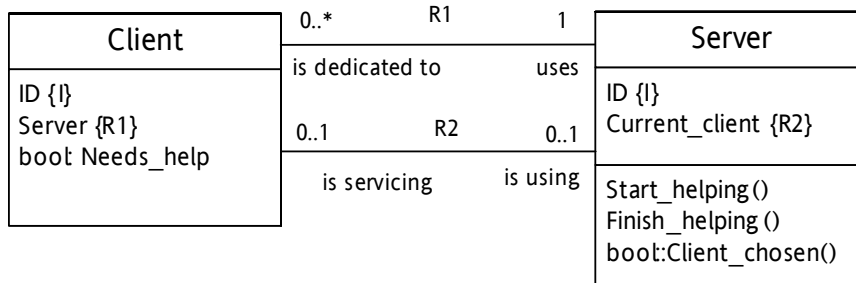
bool::Server.Client_chosen ()
    // Insert any algorithm here that selects all of the
    // clients across R1 where selected.Needs_help is true and
    // then picks a single chosen_Client based on some criteria.

    if (not_empty chosen_Client)
        relate chosen_Client to self across R2.'is serving';
        return true;
    else
        return false;
    end if;

```

Key constraint: The `Client_chosen` algorithm must never report false when there is at least one client waiting for help!

Final pattern class diagram



Why this is a good solution

All bases are covered with the final pattern signal-set technique. The server can be idle only if there are no requesting clients. As soon as one or more clients make a request, the server will choose one and begin processing. If multiple clients are pending, the server will rotate through all of its states until the call to Chosen_client() returns false.

Isn't the Server interrupted when an S1:Help signal arrives during the Phase 1 or 2 Help states? No. The model architecture does execute the ignore option for each unheeded S1:Help signal (assuming the Server state table is defined properly), but that does not concern the Server object. The architecture does its job by making the unwanted signals transparent.

This pattern does a great job of highlighting the problem analysis. We see clean Server and Client lifecycles. The Client selection algorithm can be as complex or as simple as it needs to be. And the algorithm is nicely encapsulated in a Server operation, so it doesn't weigh down the Server statechart.

An application of the Server-Client pattern

Now let's take a look at a less abstract application of the Server-Client Interlock pattern. Our example is taken from the Elevator Case Study [Starr 1, 2]. Our goal is to interlock the motion of an elevator cabin with the opening and closing of the cabin's door.

Imagine that you are inside of an elevator cabin at the hotel lobby level and you press the <8> button to go up to your room on the 8th floor. What should happen? The doors should eventually close and the cabin should begin moving upward. But let's say that at the last second you hit the <open> button because you decided that you want a drink instead. Now what should happen? If the cabin hasn't begun moving, the doors should open. But what if you are too late and the cabin begins to move upward? The doors stay closed - no party for you tonight!

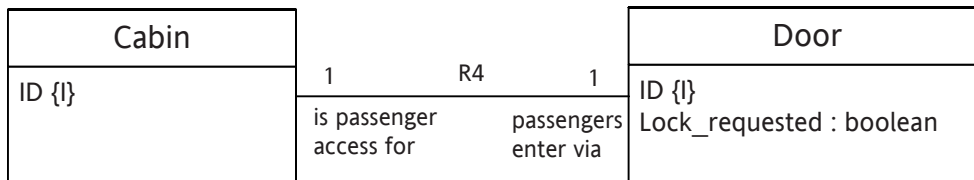
Okay, but what about that point in time where you push the <open> button in the split millisecond before the cabin starts to move? The requirement is what we need to model and it is very clear. Under no circumstances may the doors open while the cabin is in motion.

When the door is ready to close it could signal the cabin that it is ready to go. But if the door responds to an <open> press immediately after it sends the ready-to-go signal, the Cabin will start moving while the doors re-open. The cabin could somehow verify that the door is still closed before moving, but there is no guarantee that the door isn't *about* to exit the Closed state and reopen.

Modeling the requirement

Our solution inserts a LOCKED state in the Door statechart where the passenger <open> button (Door6: Open) will be ignored. We must then ensure that the cabin never moves until its door enters the LOCKED state. Furthermore, the door must remain LOCKED until after the Cabin has stopped.

The server-client interlock pattern can then be employed to synchronize the two state machines. Treat the Door class as the Server (Closed corresponds to the Ready to Help state and then we use the Door2: Lock signal in conjunction with the Lock_requested boolean).



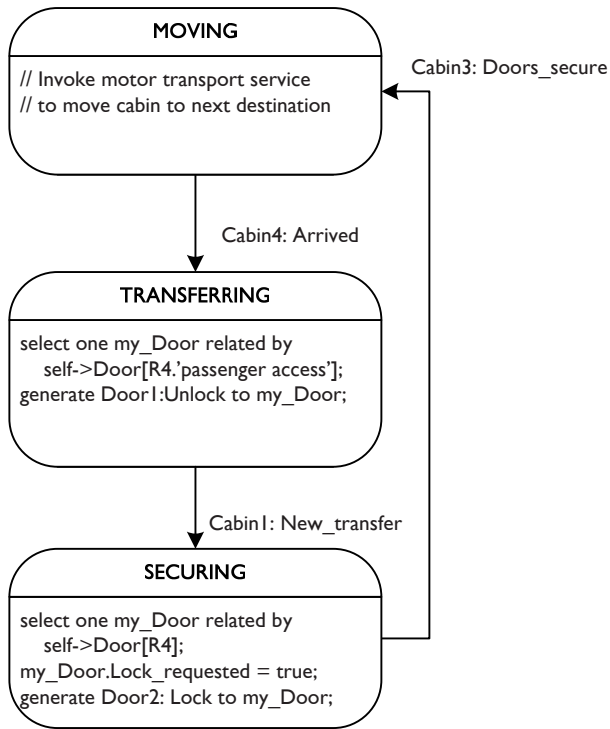
How it works

When the cabin is stationary it is transferring passengers and thus in TRANSFERRING. Now if the dispatching algorithm (elsewhere) finds a destination, the Cabin1: New_transfer event will be detected. The cabin must now get the door into LOCKED so the cabin can proceed to MOVING. At this point the door may or may not be closed. Either way, the cabin both sets the door's Lock_requested to true and fires off a Door2: Lock signal.

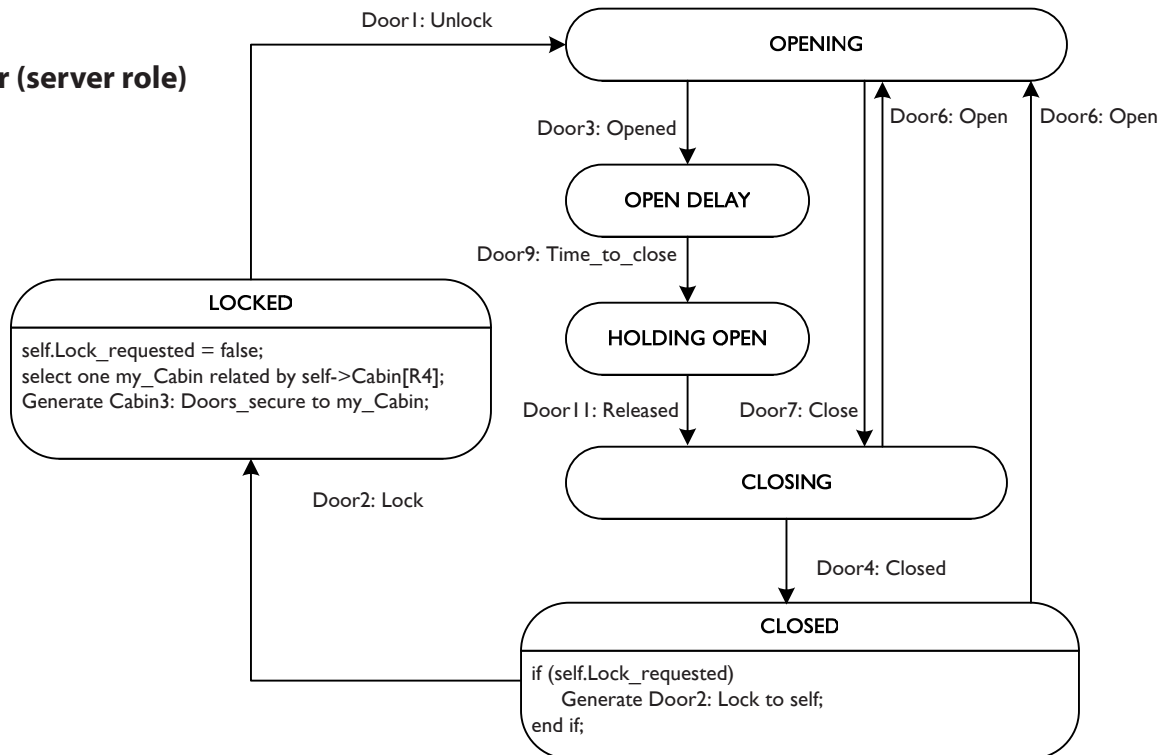
If the door is presently closed, it simply advances to LOCKED, resets its Lock_requested attribute and signals the cabin that it is okay to move with Cabin3: Door_secure.

On the other hand if the door is in OPENING, OPEN DELAY, HOLDING OPEN or CLOSING, the Door2: Lock signal will be ignored and lost forever. But that's okay because there is nothing that the door could have done with the lock signal anyway. Eventually the door will enter CLOSED. At this point the door will regenerate Door2: Lock as a self directed signal since the Lock_requested attribute had been set to true by the cabin.

Cabin (client role)



Door (server role)



Summary

Here is a distillation of the client-server interlock pattern.

Function and roles

There are two lifecycle statecharts. One plays the role of Server and the other is the Client. Each server is dedicated to one or more clients. Each client uses only one server. The client has a state where it waits for its server to do something. The server might free a resource, get out of the way, secure itself, process a transaction or do any other task.

Requirement

There is no guarantee that the server is available when its client is ready. When a server is ready, its clients may or may not be waiting. The server processes its task for only one client at a time. If a server can have multiple clients, any weighting criteria may be applied to determine which client to serve next. When a server finishes its task, it must immediately process any waiting client. If no client is waiting, the server may either wait or resume processing. In other words a server may not wait and leave one or more clients hanging or forgotten.

Mechanism

The Client-Server interlock pattern calls for a wait-for-server state on the Client, a boolean help-me attribute to mark a waiting client and a check-for-work state on the Server. If each server is dedicated to exactly one client, the help-me attribute may be placed on either the Server or Client.

When a client enters its wait-for-server state, it both sends a signal to its server, sets the help-me attribute to true and waits.

If a server is in its check-for-work state it will detect the next client waiting signal, chose a client to process, if there are multiple found, and begin performing the server task. If the server is busy, the client waiting signal will be ignored, but when the server eventually enters the check-for-work state it will see one or more client help-me attributes set to true, choose a client to serve and begin the server task. A client's help-me attribute is always reset to false by the client's server when service is rendered.

References

- [Mellor 1] Executable UML, A Foundation for Model-Driven Architecture, ISBN 0201748045, Steve Mellor and Marc Balcer, 2002, Addison-Wesley
- [Starr 1] Executable UML: A Case Study, ISBN 970804407, Leon Starr, 2001, Model Integration, LLC (out of print)
- [Starr 2] Elevator 2.0 xtUML Models, available as download at modelint.com