

Colorado Technical University



Technical Report Computer Science

Choosing a thread architecture

Bo I. Sandén
Professor of Computer Science
Colorado Technical University
bsanden@coloradotech.edu

Technical Report Number
CTU-CS-2001-004

Choosing a thread architecture

Bo I. Sandén
Colorado Technical University
Colorado Springs
bsanden@acm.org
<http://iis-web.coloradotech.edu/bsanden>

Abstract

A software architect should consciously choose between architectural alternatives for a given problem. In multi-threaded software, the architectural choices for resource sharing problems can be expressed as two patterns, the *resource-user thread* and the *resource thread*. A resource-user thread acquires and releases exclusive access to shared resources. A resource thread has permanent exclusive access to a resource and performs all the operations on it. The article discusses the choice between the two patterns.

Keywords: concurrency, software architecture, resource sharing, threads, design pattern

1. Introduction

The current interest in software architecture suggests that we put more emphasis on the choice of architecture for a given problem. As software designers we often tend to reuse a favorite architecture or follow some stepwise design method to whatever architecture it produces. We should instead consciously and explicitly choose between different architectural solutions. To make this possible, we need to catalog standard problem types and standard architectures for each type. As architects we must identify the problem at hand with a standard type and consider the solution choices. This is similar to many sciences, where we learn to recognize type problems with standard solutions. I illustrate this for the specific domain of multi-threaded software and discuss the architectural choices that exist for the common problem of resource sharing.

A system's architecture can be viewed in different ways. I use the term *thread architecture* for a view focused on concurrency. Clearly, not all systems exhibit a thread architecture, but in those that do, it is often fundamental. For example, a threaded kernel is quite different from a monolithic one. Just as software objects can often be modeled directly on objects in the problem domain, a thread architecture can capture problem-domain relationships. This is often an important starting point for the structuring of the software.

I discuss two patterns that expose relationships between resource users and shared resources in a problem domain, namely the *resource thread* and the *resource-user thread*. A resource-user thread acquires and releases exclusive access to shared resources. A resource thread has permanent exclusive access to a resource and performs all the operations on it. Both patterns are universally known and used all the time in concurrent systems. My point is that the patterns are *dual* in the sense that either one can often be used in a given problem. As a software architect and programmer of concurrent systems, I used to be unaware of this duality and would settle for one pattern as the solution in a given problem without considering the other. Typically, someone would later suggest the dual solution, which often turned out to be better. I illustrate this with examples that I have used and published over the years.

Resource-threads often provide the simpler solution, unless each resource user already has a thread that performs other processing that is independent of shared resources. Sometimes, a resource-user thread solution is preferred simply because it is more intuitive. This is important because a good architecture must be easy to describe and understand. It should give a clear mental picture of the software in problem-domain terms and have a simple rationale that can be described easily and grasped immediately, such as one thread per call in a telephone switch application, one thread per travel agent in a booking system, etc. Resource-user threads are also necessary for simultaneous exclusive access to multiple resources.

A multi-thread solution may well be less efficient than a sequential one because of the overhead for context switching. To avoid unnecessary context switching, I adhere to a restrictive philosophy of multi-threading that I have described elsewhere as *entity-life modeling (ELM)* [1, 6 - 9]. It applies to concurrent systems that respond to events in the problem environment.

In ELM, each thread in the software is patterned on an *event thread* in the problem domain. An event thread is a sequence of events. Conceptually, threads are identified through the following process: First create a *trace* by putting those events in a problem domain that the software system has to serve along a time line. Then partition the trace into event threads such that each event belongs to exactly one thread, and the events in each thread are separated by sufficient time for the processing of each event. The resulting set of threads is a *thread model* of the problem domain. Each event thread generally becomes a control thread in the software, and each thread model of a problem domain corresponds to a thread architecture.

As an example, assume that a simple problem domain consists only of events where resource users access resources. One possible thread model of this domain has one thread per resource user, which contains those events where that user accesses a resource. The software based on this thread model will exhibit the resource-user thread pattern. An alternative model of the same domain has one thread per resource, which contains the events where different users access that resource. This is the resource thread pattern.

To eliminate unnecessary threads, ELM further recommends that a thread model be *optimal*. A model is optimal if there are times when every thread is dealing with an event. An example of a non-optimal model is where there are very many threads, all of which are dependent on one or a few resources and spend most of their time waiting for access.

ELM is concerned with choosing the right threads. Other authors on the design of concurrent software [2, 5] describe strategies and idioms for mutual exclusion, etc.

2. Resource sharing patterns

The purpose of the resource-user thread and the resource thread patterns is to ensure that operations on each shared resource are completed in a serial fashion, without interfering with each other. In the resource-user thread pattern, synchronized objects represent the shared resources, and each resource-user thread obtains and releases exclusive access to them as needed. In the resource thread pattern, one thread permanently owns each resource and performs all operations on it.

A jukebox in a diner provides a textbook example of the duality between the two patterns. There is a panel at each table, where customers can request one or more songs, which are subsequently played by a single player. The player is shared since only one song at a time can be played. A resource-user thread solution and a resource thread solution are both viable.

The resource-user thread solution includes a *Customer* thread type, which represents the events associated with one user of the jukebox. Each panel initially has an associated *Customer* thread, which interacts with the first customer that enters money and requests songs. This thread has exclusive access to the panel until the customer completes the selection. It then instantiates a thread for the next customer, and requests exclusive access to the player for the first customer's songs by calling the operation *Play_Songs* on a *Player* object. The threads waiting for exclusive access to the player are kept in a FIFO queue.

Fig. 1 shows the resource-user thread solution in UML class and collaboration diagrams. Since threads and shared objects play a central role, I use parallelograms and trapezoids, respectively, as their special icons [2].

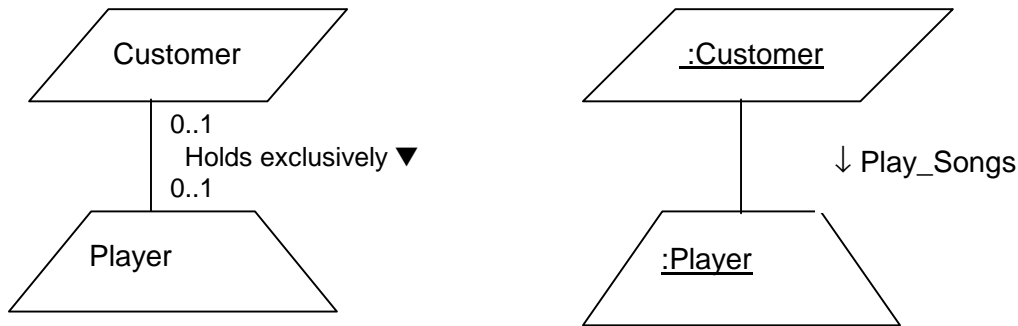


Fig. 1. *Customer* is a resource-user thread that acquires exclusive access to the player by calling the *Play_Songs* operation on the *Player* object. The class and collaboration diagrams use a parallelogram and a trapezoid as special icons for threads and shared objects, respectively.

Fig 2. shows the dual, resource thread solution. A *Panel* thread is associated with each panel. It owns the panel resource and allows customers to enter money and select songs, which it inserts in the queue. The *Player* thread owns the physical player and successively extracts and plays song requests.

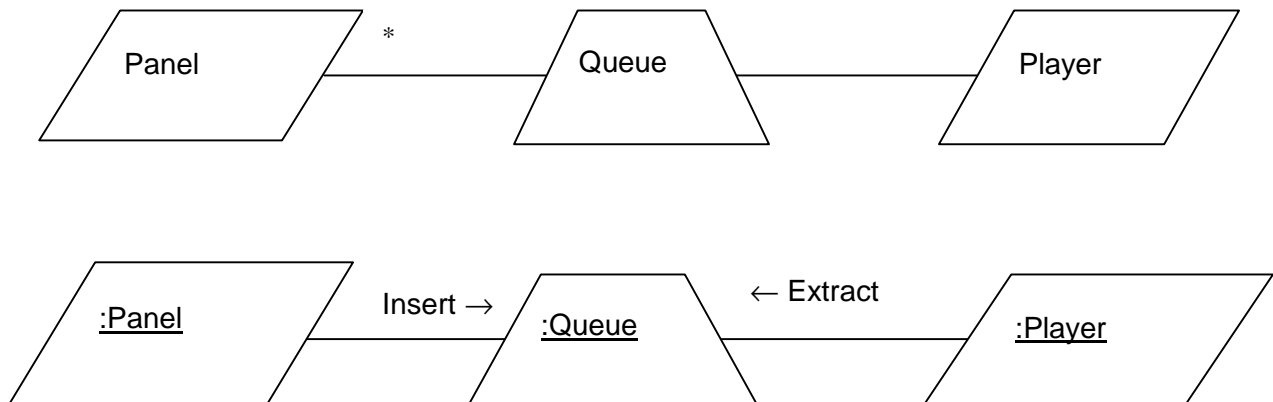


Fig. 2. *Panel* and *Player* are resource threads connected into an assembly line by means of a queue.

The resource-user thread solution in Fig. 1 generates many *Customer* threads, although at most one thread per panel plus one whose request is currently being played are active at any one time. In contrast, the resource thread solution in Fig. 2 is optimal because there are conceivably times when the

Player thread and all the *Panel* threads are active. Furthermore, *Panel* and *Player* are not created and destroyed as the *Customer* threads, but live as long as the system is running. This reduces overhead, which is important in some systems.

The resource thread solution seems obviously superior in this example, and is probably the one most architects of a multi-threaded design would choose. It has the *assembly-line* character common to many resource thread designs. The *Panel* threads form a multi-thread *station* on the line. They put requests into a common queue, which feeds the *Player* station.

Even though *Panel* and *Player* are resource threads, they are also the users of the queue resource, which illustrates how patterns in general have a tendency to "dovetail and intertwine to produce a greater whole" [3]. I next expand further on each pattern and then apply them to a few examples.

3. The resource-user thread pattern

A resource-user thread acquires and releases exclusive access to one or more resources. If the exclusive access has a duration that is noticeable in the problem environment it is called *condition* synchronization. The idea is that a thread is waiting for some condition to change rather than just for access to a synchronized object. In that situation, waiting threads must often be kept in an organized, FIFO queue. For example, *Customer's* exclusive access to the jukebox player may last several minutes.

Exclusive access that is of negligible duration in problem-domain terms is called *exclusion* synchronization and only prevents threads from accessing a shared data item at the same time and thereby jeopardizing data integrity. The probability that a thread will have to wait for access to the resource is usually very low. This is the case when *Panel* and *Player* access *Queue* in Fig. 2.

In Fig. 1, *Customer's* exclusive access to *Player* is *hidden* inside the operation *Play_Songs*. The alternative is *public* exclusive access. In that case, the resource-user thread gets exclusive access by calling an operation such as *Acquire* and relinquishes it by calling *Release*. The programmer of the resource-user thread must ensure that *Release* is actually called in all cases.

Resource-user threads that maintain dialogs with human operators are common in transaction systems such as for supermarket checkout, bank teller applications and airline seat reservation. A more recent invention is the *customer interaction system*, which is essentially a sophisticated telephone answering machine used in a variety of applications such as automated help desks. Depending on the caller's successive choices, the system plays back voice messages or makes recordings. Each call gets a thread. Shared resources such as database records are typically involved in all these systems.

4. The resource thread pattern

A resource thread performs a series of operations on a shared resource, to which it has permanent exclusive access. The resource thread pattern is applicable in problems where items (bags, parts being assembled, song requests, etc.) flow between handling stations of some kind. In some systems, there is a physical assembly line that ensures mutual exclusion and queuing. Examples of this include railway control systems and assembly plants as well as an airport baggage handling system and a toy car factory [8].

Buffering items in a queue allows the stations' processing times to vary around a common average. With no buffering, each station may hand over one item at a time to the next. The hand-over proceeds from the end of the line to the beginning. Once the hand-over is complete, all stations can operate

concurrently on their respective items, but must then await the slowest station downstream before handing over the item to the next station in line.

A *logical* assembly line is where the software receives a stream of messages and converts each message in one or more steps. Order preservation is often essential: The messages may be intentionally re-ordered, but not arbitrarily. The software itself must then ensure that the order is not changed unintentionally and must provide for the necessary queuing. This is the case in the jukebox solution in Fig. 2.

An easy mistake is to introduce a queue that no one will ever stand in. Suppose that an input data item is converted first by algorithm B, then by C. In real-time systems, it is not uncommon to implement B and C as two threads connected by a queue. (This is often the result of a dataflow analysis model.) Unless a real reason for queuing exists, B converts the item and puts it in queue. This causes an immediate context switch to C, which takes it out of the queue. An easier and cheaper solution is to let one thread perform both conversions. A classic example is where B and C are two adjacent layers in a communication protocol stack. Every message then forces a series of unnecessary context switches on its way into or out of the system.

For a queue between threads B and C to be justified, they must be on different schedules so that wait is possible, as in the following cases:

- C is associated with some resource other than the processor
- C has lower priority than B. (In this case, the queuing is for the processor.)
- B and C each has its own processor, and the processing time at each station is variable.
- C is on a set schedule. (It may run periodically.) The data item is then queued until C's next execution time.

5. Examples

Next I illustrate the duality of the resource-thread and resource-user thread patterns in different cases. More details of the solutions can be found in the references.

Remote temperature sensor (RTS). This example was first given in [10]. The RTS is a device that monitors the temperatures of a set of furnaces by means of a thermocouple connected to each furnace and a multiplexed A/D converter. The RTS sends each temperature reading as a data packet to a host computer. Several furnaces can be monitored simultaneously, each with its own frequency. The monitoring frequency for a given furnace can be changed at any time by means of a control packet from the host.

The first solution that came to my mind was based on the resource-user threads [6]. In this solution, each instance of the thread type *Furnace* periodically samples the temperature of one furnace and sends each reading in a data packet (Fig. 3). The A/D converter and the output line are resources that are shared among all the *Furnace* threads, which acquire them one at a time. The converter is exclusively held from the time a command is made to sample a particular furnace until an interrupt signals that the temperature reading is available in digital form. The output line resource is held until the data packet has been sent and acknowledged.

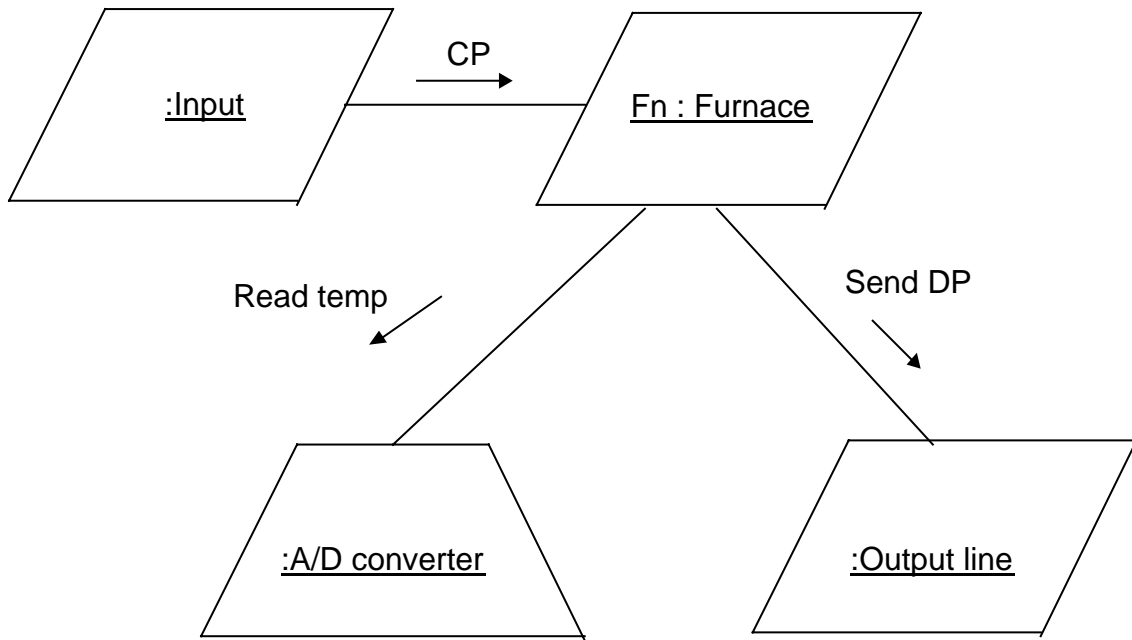


Fig 3. Resource-user thread solution to the RTS

It seemed an advantage that each *Furnace* thread could simply suspend itself until its next reading time and rely on the scheduler/dispatcher of the standard run-time system to activate it properly. But the handling of control packets became a complication. A thread, *Input*, handles all arriving control packets and communicates with the appropriate *Furnace* thread for each one. This means that a *Furnace* thread must accept a new control packet at any time. If the packet arrives while *Furnace* is waiting for its next reading time, the suspension must be broken in case the packet calls for more frequent readings.

It turned out to be easy enough to build the scheduling of all the temperature readings into a thread, *Sampler*, which also receives the control packages. *Sampler* is a resource thread that owns the A/D converter and forms a little assembly line with another resource thread, *Output_Line*. This resource thread solution [8] is similar to the jukebox solution in Fig. 2 and appears superior to the resource-user thread solution. It is simpler and has considerably fewer threads.

As in Fig. 2, the threads *Sampler* and *Output_Line* are connected by a queue. Normally, the RTS sends a data packet immediately as each temperature has been read, but a backlog of unsent packets may build up if a transmission problem develops. Should the queue become full, *Output_Line* deletes the oldest packets.

Home-heating system: In this example, first given in [4], a system heats a group of homes by keeping the difference between the actual temperature and a preset reference temperature within a certain tolerance. The heating cycle for each home involves starting a fan and letting it rev up before combustion starts, etc. A group of homes share a fuel tank. An element of resource contention is introduced by the rather curious constraint that only 4 out of 5 homes may be heated simultaneously due to undersized

fuel lines. This means that a home may be forced to discontinue heating in favor of another home. The system must ensure that no home is forcibly without heat for more than 10 minutes at a time.

In the resource-user thread solution in Fig. 4, each home has a thread, *Heater*, that takes it through the start up sequence with appropriate waits for rev-up, etc. [7, 8]. A pool of 4 heating tokens models the resource constraints. Before starting the heater, a *Heater* thread obtains a token by invoking an *Acquire* operation on *Token_Pool*. If no token is obtained immediately, the thread tries again for up to 10 minutes. It then demands a token by calling an *Insist* operation and is given one even at the price of shutting off the heat in another home, which is done by means of a shared object that serves as a logical on/off switch.

During heating, each *Heater* thread periodically senses the room temperature. It also polls the status of a main switch in case heating must be turned off. Furthermore, it determines the need to shut off its own heating for the benefit of another home by periodically querying the logical switch.

In a dual, resource thread solution, each heating token is replaced by a thread that takes a home through its heating cycle. The question is how to handle a home while it is not being heated and thus is not dependent on a resource. It is essentially a matter of polling the temperature of each home and also keeping track of how long the home is forcibly without heating. If this is implemented by means of a thread per home, the result is an assembly line with two multi-thread stations. In this case, the resource-user thread solution, where each thread is permanently associated with one home seems much simpler. But the example supports the conjecture that every resource-user thread solution has a dual.

Elevator system. A number of elevators travel in individual shafts serving requests from different floors [6, 8]. One or more polling threads collect the requests and put them in a repository. Unlike a queue, the repository has a fixed size since at most one request may exist for each button including the up and down buttons on each floor and each numbered button inside each elevator cabin. A queue is inappropriate because requests are not necessarily served in FIFO order but instead in accordance with the cabins' travel patterns. A single stop sometimes satisfies more than one request.

Each elevator has a thread that takes it through its movements, typically as far up as necessary, then as far down as necessary, etc. Whenever an elevator approaches a floor it finds out if any request requires it to stop. Whenever it leaves a floor it consults the repository to find any requests for continued travel up or down, or for a change of direction. While still maintaining exclusive access to the repository, each elevator thread also records a promise to serve a particular request, which prevents other elevators from setting out to serve it.

The solution with elevator threads is a resource thread architecture where travel requests are first handled by a sampling station, then by the elevators working in parallel to serve them. In a dual solution, each request for elevator service would have a resource-user thread that contended for access to an elevator. But since the requests do not retain their individual identity throughout it is pointless to give each of them a thread. This holds true for many systems that are based on a repository rather than a queue.

Flexible manufacturing system (FMS). This is rather a complex example with a heavy emphasis on resource contention [1, 6, 8, 9]. The FMS controls the jobs in an automated factory. Each job is concerned with the development of one part, which starts as a blank and is then worked in a series of steps at different workstations. The steps are listed in a process plan associated with each job. Each workstation includes an in-stand, a tool, an out-stand, and a robot that moves a part from in-stand to tool to out-stand. Fig. 5 shows the layout of a small FMS.

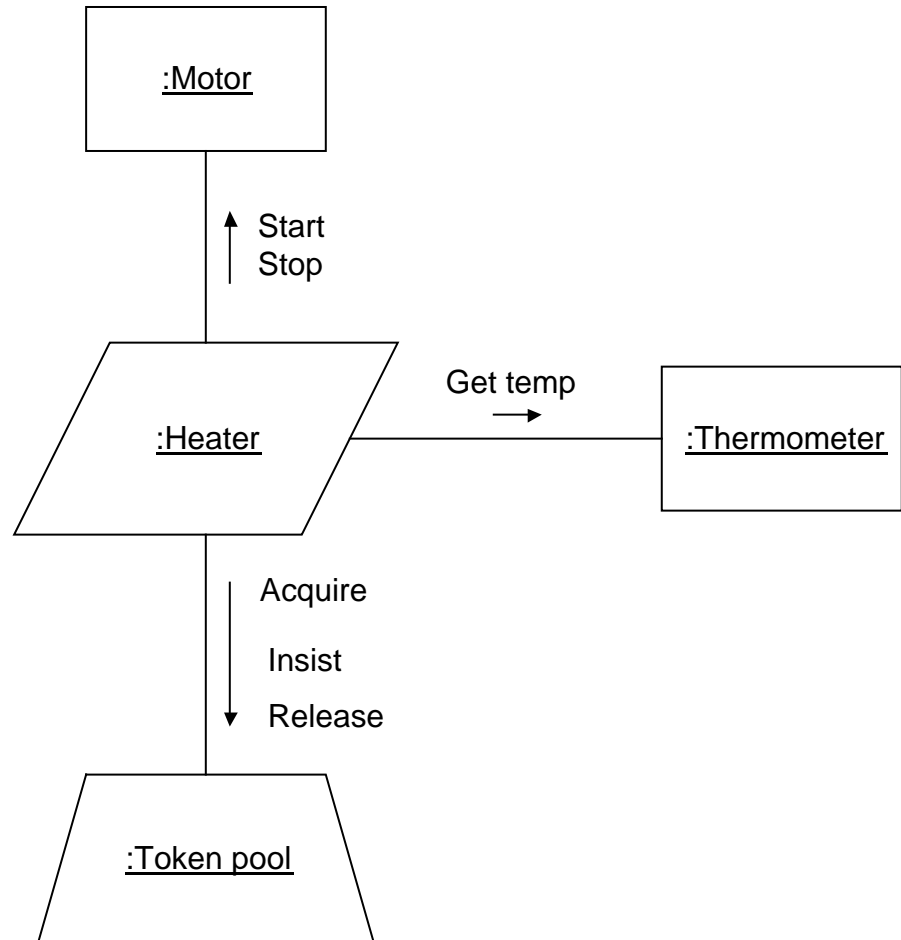


Fig. 4. Resource-user thread solution for the home heating system

Parts travel between workstations on automated guided vehicles (AGVs). A storage system houses blanks, finished parts, and parts that are staged between job steps. Each job has a dedicated storage bin. Forklift trucks carry parts between bins and storage stands, where they are accessible to the AGVs.

Resource-user threads (Fig. 6). A solution with a resource-user thread per job was first suggested by students at the Wang Institute. Once a blank is present in a bin, a Job thread is created. It attempts to acquire a workstation appropriate for the first job step and is queued if none is available. When the in-stand of a suitable workstation becomes available, the job thread acquires exclusive access first to a storage stand then to a forklift, which moves the part from bin to storage stand. After releasing the forklift, the job acquires an AGV to take the part from storage stand to workstation in-stand.

The part remains on the in-stand until the workstation tool becomes available and is then moved by the robot into the tool. When the tool has finished its operation, the robot moves the part onto the out-stand, and the job is scheduled for a workstation appropriate for its next job step. In case of contention, a part may be staged in storage until such a workstation becomes available.

In this solution, Job threads have simultaneous exclusive access to resources in various combinations. For example, when the part sets out from the bin, it has a workstation in-stand, a storage stand and a forklift.

Resource threads (Fig. 7). It is not as easy to transform the Job-thread solution into a dual, resource-thread solution as in the previous problems. This is because the Job thread has simultaneous exclusive access to multiple resources. We cannot simply give each workstation, AGV, forklift and storage stand a resource thread and let Job objects pass between them, since one and the same Job object may have exclusive access to a workstation, an AGV and a storage stand at the same time.

A resource-thread solution instead focuses on the workstations only. The workstations form a logical assembly line and are connected by queues of job objects. (The physical part associated with each job may be on a stand or in storage.) Each workstation is itself a small assembly line with three resource threads dealing with the following:

Job_to_Instand:	movement of each part to the in-stand and into the tool.
Job_in_WS:	movement from tool to out-stand.
Job_to_Storage:	movement from out-stand to storage when a part is finished or must be staged.

Whenever a workstation's in-stand becomes empty, Job_to_Instand finds the next appropriate job and transports it to the in-stand. When a job is done at a workstation, Job_in_WS inserts it into the appropriate job queue for its next job step, unless the job is done, in which case Job_to_Storage moves it to storage. The three thread types account for all events that happen to a job, since every job that is not passive in storage either is on its way to or from a workstation or is being processed at one.

The other resources, such as the forklift or the storage stand, remain represented by synchronized objects. So, although they are resource threads, Job_to_Instand and Job_to_Storage are also resource users that need simultaneous exclusive access to resources such as storage stands and AGVs in about the same combinations as Job.

The resource thread solution has the advantage that jobs in storage have no threads. So, if there are many more jobs than workstations, this solution has fewer threads than the Job thread solution. All three threads per workstation are busy when one job is traveling to the in-stand, another job is moving to the out-stand and a third job is traveling to storage. This makes the resource thread solution potentially optimal, although the number of AGVs, forklifts and storage stands limits the total number of jobs that can be in transit at any one time. Furthermore, the workstation related threads live as long as the system is running while Job threads are created and destroyed as jobs enter and finish processing. Nonetheless, many find the Job thread solution more intuitive.

6. Conclusions

The examples suggest that the thread architect often has a choice between resource-user thread and resource thread solutions. The resource thread pattern often leads to the simpler solution in problems where items do little but wait for and use shared resources. This is the case with physical assembly lines as well as the jukebox and the RTS. The resource-user thread pattern should be considered if the resource users already have threads as in the home-heating system and the transaction systems. In other cases, such as the FMS, some may find the resource-user thread solution more intuitive. A more general conclusion is that an architect should always be on the lookout for alternative solutions and make a habit of weighing the pros and cons of each before settling for one or the other.

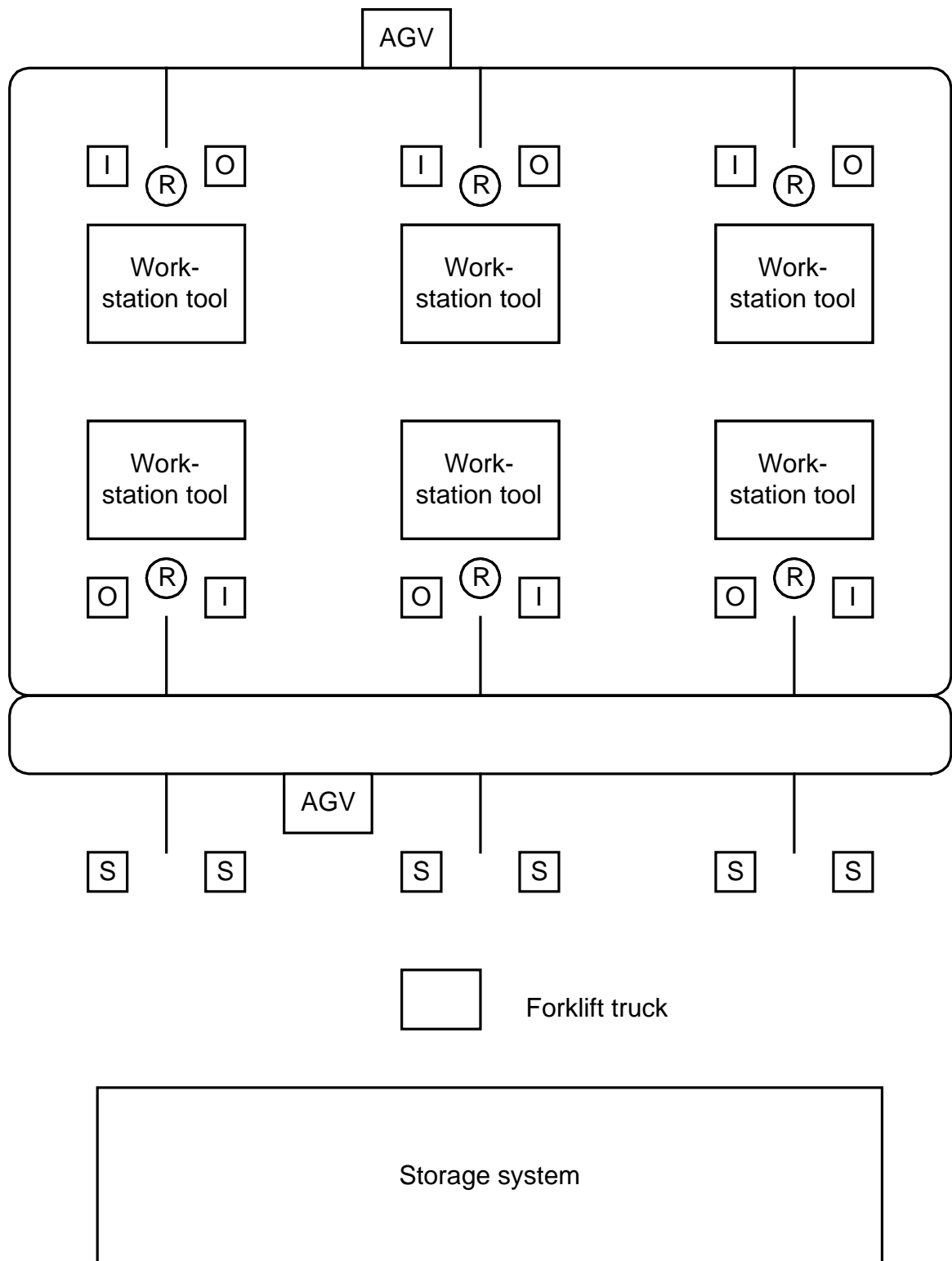


Fig. 5. Layout of a flexible manufacturing system. I: in-stand; O: out-stand; S: storage stand; R: robot; AGV: automated guided vehicle

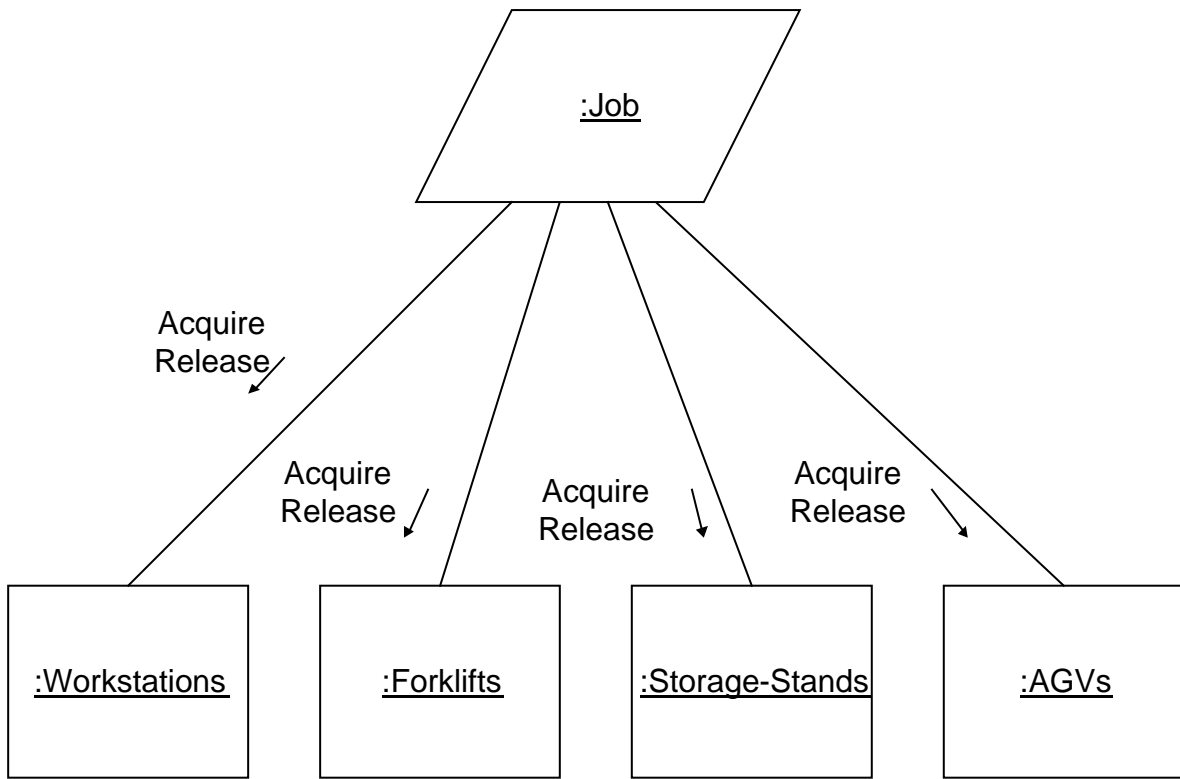


Fig 6. Object diagram of the resource user thread solution of the FMS problem

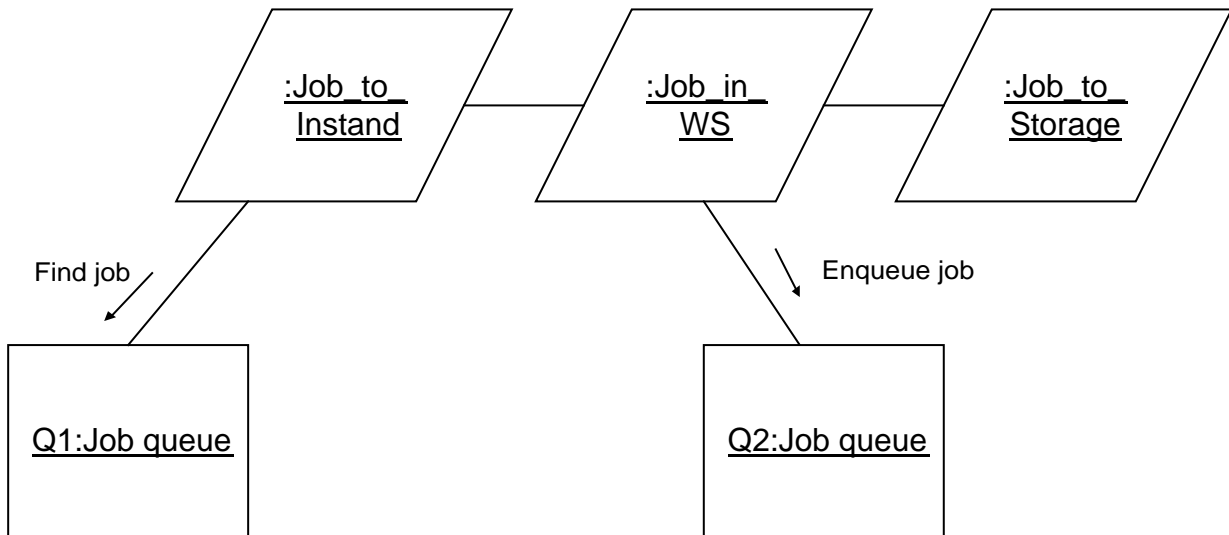


Fig. 7 Interaction between the resource threads in a single FMS workstation. Different workstations are connected via the job queues.

References

- [1] J. R. Carter and B. I. Sandén. Practical uses of Ada-95 concurrency features. IEEE Concurrency 6:4, (October/December 1998), 47-56.
- [2] B. P. Douglass, Real-time UML. Developing Efficient Objects for Embedded Systems, Addison-Wesley 1998
- [3] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley 1995
- [4] D. Hatley and I. A. Pirbhai. Strategies for Real-Time System Specification. Dorset House, 1987.
- [5] D. Lea. Concurrent Programming in Java™. 2nd Ed., Addison-Wesley 2000.
- [6] B. I. Sandén. Software Systems Construction with Examples in Ada. Prentice-Hall, 1994.
- [7] B. I. Sandén. Using tasks to capture problem concurrency. Ada User Journal, 17(1):25-36, March 1996
- [8] B. I. Sandén. A course in real-time software design based on Ada 95. Asset A 825, ASSET repository 1996. <http://www.coloradotech.edu/~bsanden/DISA/rtcourse.html>
- [9] B. I. Sandén. Modeling concurrent software. IEEE Software, Sept/Oct 1997, 93-100.
- [10] S. J. Young. Real-time Languages: Design and Development, Ellis Horwood, Chichester, West Sussex, England 1982