

**Colorado Technical University**



**Technical Report  
Computer Science**

**Entity-Life Modeling:**

**Modeling a thread architecture on the problem environment**

**Bo I. Sandén**

Colorado Technical University

Colorado Springs, CO

[bsanden@acm.org](mailto:bsanden@acm.org)

<http://iis-web.coloradotech.edu/bsanden>

**Technical Report Number**

**CTU-CS-2002-003**

# Entity-Life Modeling

## Modeling a thread architecture on the problem environment

Bo I. Sandén  
Colorado Technical University  
Colorado Springs, CO  
[bsanden@acm.org](mailto:bsanden@acm.org)  
<http://iis-web.coloradotech.edu/bsanden>

### Abstract

In object oriented development, classes and relationships found in the problem domain are carried seamlessly into the design and implementation. *Entity-life modeling* extends this modeling idea into the time domain. Sequences of event occurrences called *event threads* are identified in the problem space and form the basis for control threads in software. This modeling can occur at two levels. The *basic level* mechanically finds sequences of events in the problem, while the *pattern level* deals with concurrency structures in the problem such as resource users contending for resources. Such structures are modeled by means of threads and synchronized objects in software. This modeling approach to concurrency leads to the concept of a *concurrency level*, which indicates the number of threads that is, in a certain sense, optimal for a given problem.

### 1. Introduction

With Java threads and the wider availability of multi-processors, more and more programmers are confronted with concurrency. Concurrent threads let you take advantage of multiple processors to speed up execution. They are also useful on a single processor where one thread can compute while others are waiting for external input. This paper outlines an approach called **entity-life modeling (ELM)** for designing multi-thread programs [5, 6, 9].

Multi-threading is more difficult than sequential programming. Complicated multi-thread architectures often have subtle synchronization errors that cause irreproducible failures. A program with too many threads interacting in obscure ways can easily become a debugging and maintenance disaster. Unnecessary threads take up stack space even when inactive and incur runtime overhead for context switching.

For these reasons, a well thought-out architecture is particularly important with concurrency. A good thread architecture should have a simple rationale that can be immediately grasped, such as “one thread per call” in a telephone switch application, “one thread per travel agent” in a booking system, “a periodic thread that gathers meteorological data” in a Mars rover, etc. The architecture must give a clear mental picture of the software in problem-domain terms. This is similar to the idea in object orientation that software can be *modeled* after objects with intuitive roles in the problem environment and their relationships. Extending the concept to the time dimension, ELM describes how multi-threaded software can be patterned on concurrent structures in the problem. In addition, ELM takes a restrictive view of concurrency and aims to eliminate unnecessary threads and context switches.

## 1.1 Concurrency structures in the problem domain

Some examples of concurrency structures in the problem domain that lead to intuitive concurrent software solutions are the following:

Independent processes. In many problems, different users carry on their work independently. Examples include phone operators for a catalog business running transactions against a database, and customers using the different ATMs of a bank. Each phone operator may get a thread. Allocating one thread to each ATM has the same effect. Inanimate entities such as the various elevator cabins in a multi-elevator control system can also have their independent threads.

Asynchrony. Consider a jukebox in a diner that is connected to panels at the various booths where guests can insert money and select songs. Customers may select a number of songs at their pace. The jukebox CD player then plays the songs at a rate determined by the length of each and what songs different customers have ordered. This asynchrony between song ordering and playback can be captured by *Panel* threads and a *Player* thread connected by a buffer of song requests.

Another example is a data entry system where one thread captures data from a keyboard into a buffer, while another thread writes data from another buffer to disk.

Resource contention. If a problem deals with resource contention, the implementation can have threads corresponding to the resource users in the domain, and synchronized objects that represent the resources. Track segments used by switch engines in a switchyard are examples of shared resources [5, 9].

Another example concerns a flexible manufacturing system (FMS) where jobs contend for workstations and automated guided vehicles [5, 6, 9].

State machine with concurrent activities. Many electromechanical devices can be modeled as state machines with concurrent activities. Examples include a cruise controller for a car, a bicycle odometer and a garage door controller as well as much more complex devices [8].

These concurrency structures may coexist and overlap in a given problem. For example, the *Panel* threads in the jukebox problem are intuitively independent because customers in different booths can order songs simultaneously. But the jukebox example can also be viewed as a resource contention problem where the customer threads contend for access to the one CD player. Each example is further discussed throughout the paper.

ELM attempts to lead the analyst/designer to intuitive architectures that apply in a given problem. The term "entity-life modeling" is meant to suggest that each thread models an **entity** in the problem environment, such as an operator, a jukebox customer or an elevator. An entity is an object whose "life" can be modeled as a thread. Sometimes, the thread itself has an intuitive role in the problem environment, and identifying an entity becomes unnecessary. The thread that captures meteorological data may be an example. In any case, the goal is for each control thread to have meaning in problem-domain terms. ELM does not produce internal threads that, for example, read messages from a queue, perform some computation, and place the result in another queue. That kind of dataflow threading is further discussed in the section on other de-

sign approaches. Instead, each external impulse is normally handled completely by a single thread.

## 1.2 Two modeling levels

ELM lets you model the thread architecture of software on the problem domain at one of two levels. At the basic level, you model control threads on **event threads**. (When there is a risk of confusion, I shall refer to threads in the software as *control threads*.) An event thread is defined as a sequence of event occurrences in the problem domain. An example is the interaction between a customer and an ATM machine, where the customer creates events by inserting a card, pressing buttons, etc.

At a higher level of abstraction, a software architecture of threads and shared objects is modeled directly after concurrent *patterns* in the problem, similar to the concurrency structures listed above. In some cases, the problem can be represented as a state machine with associated activities. In software, the state machine is implemented as a synchronized object and the activities as threads. Another common pattern involves resource contention as when different trains, one at a time, use the same track segments. Such situations are also modeled by means of threads and shared objects in software. The pattern approach is evidently quite intuitive; my experience is that design teams easily identify, for example, a resource allocation problem in the domain and see how it can be represented in software.

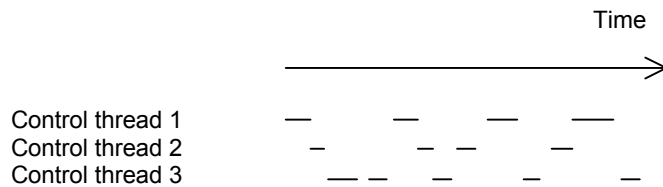
The basic principle of modeling control threads on event threads is a fallback when no pattern can be identified. It is general and tends to find all technically possible solutions. This can sometimes stretch the designer's mind and produce workable solutions that no one had thought of. But it can also produce threads without intuitive interpretations in the problem domain. In the following, I first discuss the basic approach, then the patterns approach.

## 2. The basic level: Identifying event threads

Object orientation can be called a modeling approach to software development in that the classes and objects that you find in the analysis of the domain seamlessly carry into design and implementation, which are also based on classes and objects. The same holds for the dynamic model where a domain object's behavior is captured in a state diagram, which is then implemented.

The ideas of object orientation first emerged in programming languages such as Simula and Smalltalk and were later abstracted into design and analysis. Given that a program can be expressed in terms of classes and associations, object-oriented analysis becomes a matter of identifying those entities in a problem that map seamlessly onto those programming constructs.

Approaching concurrency in the same manner, we want to identify those entities in the problem domain that map seamlessly onto control threads. For this we must recognize that threading primarily deals with time. It is common to visualize control threads by laying out the computation of a multi-thread program along a time line as follows:

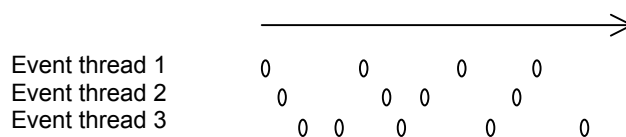


This diagram shows a series of activations of each thread. On a single processor, the activations are interleaved; with multiple processors, they can overlap. A single activation is broken when a thread is preempted by a higher priority thread and then resumed.

With this view of control threads, how can we use them to capture some aspect of the problem domain? We shall assume that each thread activation is caused by the occurrence of an *event* that the software must react to. In the ATM system, a customer creates events by inserting a card, pressing keys, removing money, etc. In the elevator controller, the arrival at a floor is an event signaled by a sensor in the elevator shaft. We also assume that a thread spends a finite (non-zero) amount of time processing each event occurrence.

The model must include **timing events** defined as "x seconds have passed since event y". A timing event is not based on an external impulse. Instead, it is created by the software itself, typically by means of a sleep statement, and used when the software must take an action of its own such as refreshing a display or sampling some quantity in the environment.

In the following diagram, each thread activation in the previous diagram has been replaced by the event occurrence that causes it. Note that an event such as "elevator 1 arrives at floor 3" can have multiple occurrences, each of which is shown separately. The diagram shows the sequence of all the event occurrences in the problem, partitioned into *event threads*.



Each **event thread** is a sequence of problem-domain event occurrences that are separated in time, which allows for the processing of each event. A **thread model** is a set of event threads that accounts for all relevant events in the problem. We map a thread model onto the software by implementing the event threads as control threads.

## 2.1 Event partitioning

Conceptually, event threads are identified by means of the following *partitioning* process:

1. Create an imaginary *trace* by putting all the event occurrences that a software system has to serve along a time line. Include events from the problem domain as well as timing events.
2. Partition the trace into event threads such that the following holds:

- Each event occurrence belongs to exactly one thread
- The event occurrences in each thread are separated by sufficient time for the processing of each event.

The result of this process is a **thread model** of a problem, defined as a set of event threads accounting for all relevant events in the problem.

Data-entry example: A simple data entry application prompts an operator to input 100 numeric sales figures into a buffer and then uses those values to update a revenue spreadsheet on disk [1]. The events in this problem are *keystrokes* on one hand, and *writes* to the database on the other. The keystrokes are separated from each other in time, as are the writes. The keystrokes that go into a given buffer are also separated in time from the writes from that buffer. This gives us two possible thread models:

Model 1: One *Operator* thread and one *Database* thread. *Operator* consists of all the keystrokes. *Database* consists of all the writes. This model captures the asynchrony between typing data into a buffer and writing that buffer to the database.

Model 2: Two or more *Buffer* threads. Each *Buffer* thread consists of the keystrokes that go into one buffer and the database writes from that buffer. Because data can be entered into only one buffer at a time, and only one buffer at a time can be written to the database, each *Buffer* is a *resource user thread* that exclusively accesses the keyboard and the database in an alternating fashion.

## 2.2 Concurrency levels and optimal thread models.

The partitioning rule does not limit the number of threads. For example, we could have four threads in the data-entry problem:

Model 3: *Operator1* that represents the keystrokes into buffer 1  
*Database1* that represents the database writes from buffer 1  
*Operator2* that represents the keystrokes into buffer 2  
*Database2* that represents the database writes from buffer 2

This model is intuitively redundant because the threads are synchronized pair-wise. For example, *Operator1* is synchronized with *Database1* because the buffer must be filled before it is written. On the other hand, *Operator1* is synchronized with *Operator2* because only one buffer can be filled at a time. This means that we may as well combine threads in one of the following ways:

1. Combine *Operator1* and *Operator2* into one thread called *Operator*, and *Database1* and *Database2* into another one called *Database*. This brings back Model 1.
2. Combine *Operator1* and *Database1* on the one hand, and *Operator2* and *Database2* on the other, and get two *Buffer* threads. This brings back Model 2.

This suggests that having two threads is somehow optimal in the data entry problem. Whether we choose Model 1 or Model 2, the two threads in each model can be busy at the same time, one handling keystrokes and the other handling database writes. With more threads, all but two

must be waiting at any given time. Two threads can operate independently; additional threads cannot.

A notion of *coincidental simultaneity* has proven useful to identify what threads are intuitively independent in this sense. The idea is that a set of event threads are independent if once in a while they all happen to have an event at the same time. A keystroke and a database write can occur at the same time (if they concern different buffers). This makes the *Operator* thread, which consists of keystrokes, independent of the *Database* thread, which consists of writes. The same holds if we have two *Buffer* threads, one of which can be capturing keystrokes while the other one is being written to disk. But with three *Buffer* threads, there is never a moment in time when each has an event occurring. One of the threads has exclusive access to the keyboard and another to the database, while the third must be waiting.

Events are instantaneous, so strictly, the likelihood that they occur at the same time is zero. For a more careful definition, we shall say that threads **co-occur** if an arbitrarily short time interval can be found where each of them has an event occurrence. In Model 3, *Operator1* and *Database1* do not co-occur because they need exclusive access to the same buffer. But *Operator1* and *Database2* co-occur since one buffer can receive a keystroke at the same time as the other buffer is being written to the database.

This allows us to define the **concurrency level** of a problem as *the maximum number of events that can ever occur within an arbitrarily short interval*. As we have seen, the concurrency level of the data entry problem is *two*: At any given time, at most one keystroke and one write can occur. For some realistic problems, the concurrency level cannot be determined, but you can often find at least an upper limit, which is sufficient. An **optimal** thread model is one where *all threads co-occur*. The number of threads in an optimal thread model of a problem is equal to the concurrency level of the problem.

### 2.3 Latitude for the analyst/designer

Although the concepts of concurrency level and optimality may sound rigorous, ELM is not an attempt to mathematize software design. In practice, it is up to the analyst/designer to determine how close in time two events can occur. This is similar to analysis based on state machines where it depends on the problem whether an operation can be abstracted as an instantaneous action or must be regarded as an activity.

The purpose of co-occurrence and optimality is to indicate to the designer what is a reasonable number of threads. It is a way to evaluate and characterize a thread architecture. Optimality is a safeguard against excessive and counter-productive threads, but is not considered an absolute requirement in entity-life modeling. In fact, non-optimal thread models are sometimes useful. A designer may choose a model with additional threads for various reasons, often to separate concerns or to design for change.

As a contrived example of how a non-optimal solution may come about, assume that we have settled for Model 2 in the data-entry problem. It then turns out that a diligent data entry operator gets far ahead of the database updating and sometimes must wait for a buffer to become available. This can be solved by adding a *Buffer* thread, leading to a non-optimal model where one thread is always waiting. (Although this is workable, Model 1 offers a preferable and optimal solution where the buffers are passive objects without threads, and an extra one can easily be added.)

## 2.4 Modeling event threads as control threads.

Each thread model for a given problem is potentially the basis for a software design. As a rule, each event thread maps onto a control thread. By basing the control threads on a thread model of the problem, we ensure that each event is handled by one control thread, and that no control thread is inundated with simultaneous events. Such simultaneous events are taken care of by different threads, which, furthermore, may execute on parallel processors. The closer a model is to optimum, the fewer redundant control threads it has.

Sometimes, more control threads than event threads are needed. An example is when an event cannot be attributed to a thread without further analysis of its associated data. On the other hand, some events are hardware interrupts and can be dealt with entirely by interrupt handlers, which may incur less overhead than threads. This is illustrated by the state machine pattern. Further, as long as you have a single processor, it is always possible to implement the software as one sequential process. The concurrency level is not a universal constant for a given problem but dependent on the entity-life modeling philosophy.

## 2.5 Finding intuitive entities

One premise of ELM is that the threads must have intuitive roles in the problem domain. Often they are associated with intuitive entities. Event partitioning and the idea of coincidental simultaneity may produce many thread models for a given problem and even several optimal ones. Even if a model is clearly workable, it is sometimes difficult to come up with an intuitive entity for each thread.

In practice, the search for thread models instead often starts with entities or intuitive threads of events. For example, an elevator cabin or an operator may be an intuitive entity candidate, whose "life" is an event thread. Then the question is, does a given set of entities lead to an appropriate thread model? The partitioning process turns into the following checklist:

1. Do the entities and threads account for all event occurrences in the problem? If not, additional entities and threads are needed.
2. Do all simultaneous event occurrences belong to different threads? If not, the candidate entities need to be partitioned.
3. Is the solution close to optimum? If not, is each thread justified, or could it be eliminated? As mentioned, optimality is no absolute requirement but is left to the designer's discretion.

Human users are often entities, and the *user thread*, which maintains a dialog with a human user, is a common thread type. Each ATM has such a user thread -- an instance of a user thread class. Threads representing different users typically co-occur. For example, two ATM customers may both hit the enter key at the same time.

## 3. Modeling concurrency patterns in software

In many problems, you can identify not only standard entity or thread types such as user threads, but *patterns* of cooperating entities. You can often base an entire thread architecture on such patterns without explicitly partitioning events. Two widely used patterns for resource

sharing are discussed below. The *state machine pattern* represents a case where an event thread does not need its own control thread.

### 3.1 State machine pattern

In UML, the dynamic behavior of an object over time can be captured by means of a state machine [3]. A state machine is useful for describing the possible orderings of the event occurrences in a thread. It also describes instantaneous *actions* that the object takes in response to events and *activities* that go on in particular states.

The *state machine pattern* applies to a fairly common problem type that can be described by means of a single state-machine and its activities [8]. Even though a state machine defines an event thread, it is convenient in this particular case to implement it as a synchronized object, and let the activities be threads. The technique is to minimize the computations made in the synchronized object itself and instead move them to the threads.

A classical cruise control system is a simple example of a feedback system. A synchronized object, *CruisingState*, say, contains the current state of cruising, which can be modified by the car driver. One activity is the periodic adjustment of the throttle. This leads to a *Throttle* thread with an obvious role in the problem. *Throttle* obtains the current state of cruising and the current target speed from *CruisingState* and adjusts the throttle as necessary. The cruise controller receives the driver's inputs either through hardware interrupts or by means of sampling. The latter solution requires one or more sampling activities with their own threads.

In some cases, the activities in different states are radically different and require different threads for separation of concerns. The *mode change*, where one set of activity threads are phased out and another is phased in, can be non trivial. With entity-based threads, this can sometimes be simplified in that a thread can handle the mode change internally. This is the case with *Throttle*, which can quite easily adapt to the current state of cruising. It is active in two states, one where constant speed is required and another, where constant acceleration is required.

In another example, a bicycle odometer has the states Speed, Distance, and Time, and in each state, the activity consists of periodically displaying the current speed, the distance traveled and the time spent during the current trip, respectively. This is handled by a single thread that repeatedly queries the state machine for the current state, then displays the appropriate value.

### 3.2 Resource-sharing patterns

Most concurrent problems deal with resource sharing. Dual patterns exist for dealing with accesses to a shared resource. I call them the *resource-user thread* and the *resource thread* patterns. They ensure that the operations on each shared resource are completed in a serial fashion, without interfering with each other [7].

In the **resource-user thread** pattern, each *resource user* has a thread that contends with others for some shared resource, which is represented by a synchronized object. The *Buffer* threads in the data-entry problem are resource users that need exclusive access to the keyboard and the database. In a database system, each thread may represent an operator, while the shared resources are database records.

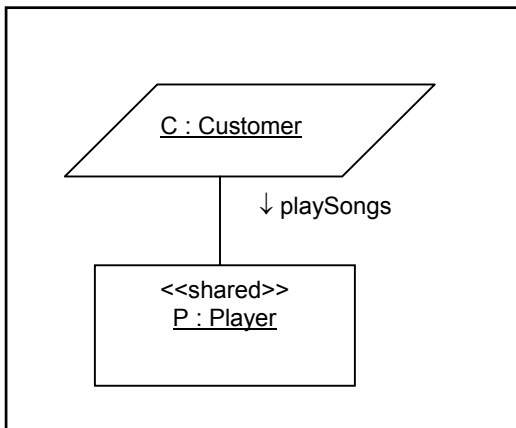
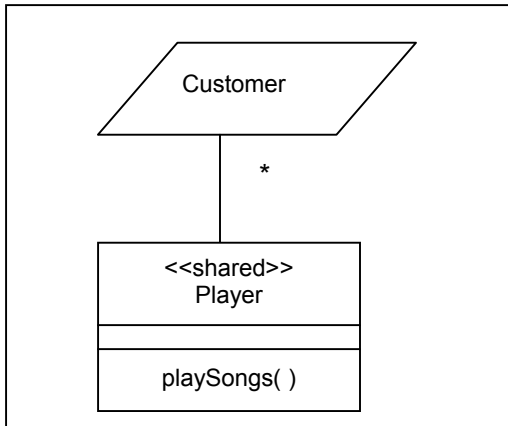
In the **resource-thread** pattern, each thread represents a resource. Resource threads are often arranged in a virtual *assembly line* and connected by queues implemented as synchronized objects. The *Operator* and *Database* threads in the data entry problem are resource threads forming a small assembly line with two stations. With this metaphor, buffer objects travel along the assembly line from one thread to the other.

As long as each resource user has exclusive access to no more than one resource at a time, the designer often has a choice between a resource thread and a resource-user thread solution. This is illustrated by Models 1 and 2 of the data-entry problem and also by the following example, where the resources are devices in the problem environment.

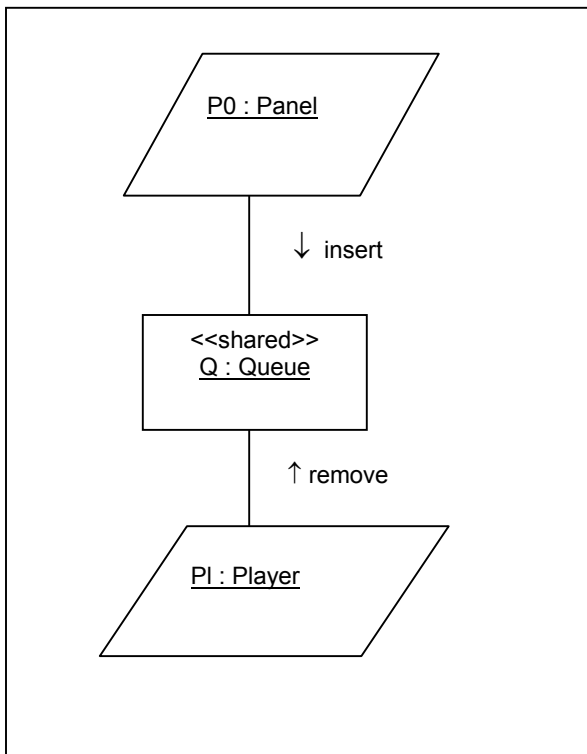
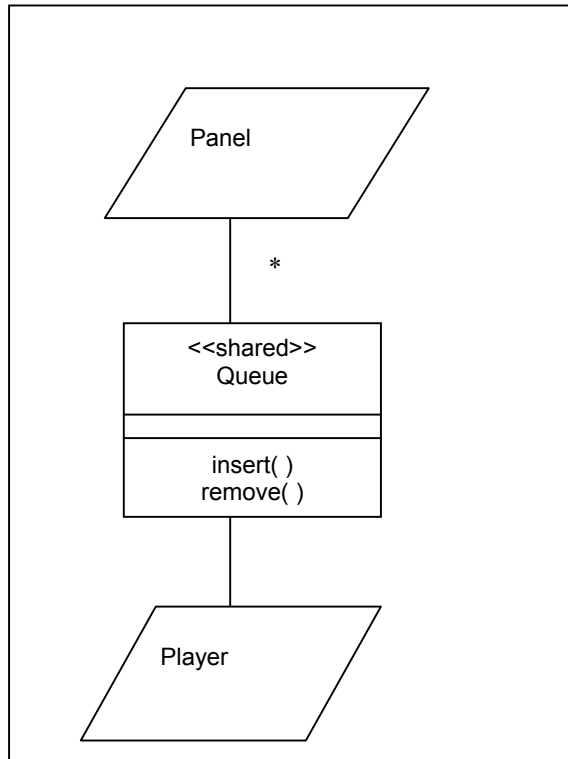
Jukebox example. The jukebox software must allow customers to use the panels in their booths to request songs simultaneously and must arrange for the songs to be played one at a time on the CD player. Events in the jukebox problem environment include: select song, start playing song, stop playing song, etc.

Fig. 1 shows a resource-user thread design in the UML class and object diagram notations [3]. I use a parallelogram as an icon for a class or an object with a thread [4]. Each panel initially has an associated *Customer* thread, which dialogs 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 creates another thread instance for the next customer, and requests exclusive access to the player for the first customer's songs. This exclusive access occurs inside an operation *playSongs()* on *Player*. For the jukebox to work properly, threads waiting for exclusive access to *Player* must be kept in a FIFO queue as specified in RTSJ [2]. That there are waiting threads indicates that this is not an optimal thread architecture, although perhaps quite an intuitive one.

In an alternative, resource thread solution (Fig. 2) *Player* is a thread that owns the physical player and successively plays songs from a *Queue* object that contains the waiting requests. A *Panel* thread is associated with each panel. *Panel* owns the panel resource and allows customers to enter money and select songs, which it inserts into the queue. In this solution, the *Player* thread communicates asynchronously with the *Panel* threads via the queue.



**Fig. 1. Customer thread design of the jukebox.**  
a) Partial class diagram. b) Partial object diagram.



**Fig. 2. Player thread design of the jukebox.**  
 a) Partial class diagram. b) Partial object diagram.

The resource thread and resource user thread patterns are very common. Once you have constructed, say, a resource-thread solution, it's useful to look at the dual, resource-user thread solution. I have often found that the dual solution is simpler than the one I first thought of.

### 3.3 Simultaneous, exclusive access to multiple resources

The resource-thread and resource-user thread patterns also apply when resource users need *simultaneous*, exclusive access to *multiple* resources. In an automated switchyard problem [5, 9], switch engines travel from track segment to track segment. Switch engines are resource users and are represented as threads in software.

In the flexible manufacturing system (FMS [5, 6, 9]), jobs being processed in a factory have simultaneous exclusive access to multiple resources such as a workstation, a forklift and an automated guided vehicle (AGV). In a resource-user thread solution, each Job has a thread while synchronized objects guard the access to the various resources. Job threads waiting for access to shared resources are held in wait sets. (As in the jukebox, the wait sets have to be FIFO queues.) When a Job thread releases a resource, any other Job thread that needs it is notified.

The FMS problem has an alternative, resource-thread solution, where the jobs are passive objects, and threads are associated with workstations. It turns out that more than one thread is needed for each workstation. One thread attempts to keep busy by finding jobs to work on. Because it must arrange for moving the jobs between workstations, it is also a resource-user thread that contends for access to an AGV or a forklift [6].

It is interesting to compare these solutions to an approach that is used by some designers of concurrent, object oriented systems. In that approach, you identify intuitively active objects and give each of them a thread. The switch engines are such intuitively active objects so in the automated switchyard problem, the active object approach leads to a design with resource user threads.

This is not the case in the FMS problem however, where the intuitively active objects are the workstations, AGVs and forklifts. Giving each of them a thread makes for rather a complex solution with the Jobs as passive objects and the queues of waiting jobs also implemented as objects. When an AGV takes a job off of a stand, for example, the AGV thread has to find any other job that needs the stand, update its status and queue it for an AGV or a forklift [6]. This takes the AGV thread quite far afield from its intuitive business of running a vehicle. In each of the solutions presented here, the interface is simpler: When a thread releases a resource, it only calls *notify()*.

## 4. Aspects of ELM designs

### 4.1 State representation in threads

In the state machine pattern, a state machine is implemented as a synchronized object. But a resource user entity, for example, which is implemented as a thread, also moves through different states. You can often construct the control structure of the *run()* method in such a way as to capture much of the state information. The job thread in the resource-user thread solution of the FMS is a good example. Its simplified structure is schematically as follows [9]:

```

Get information about first job step
while (not done)
{
    Acquire workstation instand
    Acquire storage stand
    Acquire forklift
    Travel to storage stand
    Release forklift
    while (True)
    {
        Acquire AGV
        Travel to instand
        Release storage stand and AGV
        Inform workstation about job
        // Process continues at workstation microprocessor
        Wait for job to appear on outstand
        Get information about next job step
        Break from inner loop if job done
        Acquire instand of workstation for next step
    }
    Acquire storage stand
    Acquire AGV
    Travel to storage stand
    Release AGV
    Acquire forklift
    Travel to bin
    Release storage stand and forklift
}

```

Here, the current state of the job and the resources it currently holds or is waiting for are implicit in the control structure. For example, at the top of the “Floor” loop, the job has acquired a workstation and is waiting for an AGV that will take it there. Incidentally, this is the programming style resulting from Jackson structured programming [5]. The same style may also be used in a simulation program based on process interaction. Not all threads lend themselves to this kind of representation, however. Instead, you can implement the *run()* method with a state variable and switch statements.

#### 4.2 Event threads and use cases

A *use case* is a sequence of operations on different objects initiated by an external *actor* toward a particular goal [3]. It is often (but not necessarily) a dialog between the system and a human operator. The jukebox has the use case “Request and Play Songs” where a Customer actor inserts money and selects one or more songs, which are subsequently played.

Use cases and threads are related because an actor's life is often an event thread, making the actor a candidate entity in the ELM sense. A human actor often gives rise to a user thread. The *Customer* thread in the jukebox example executes the use case once. In an ATM machine, a single control thread may execute a series of transactions from different customers one after the

other. Human actors and user threads are fine for interactive systems that keep track of each individual user request from beginning to end.

There is no guarantee that the actors found in use case analysis are useful as ELM entities. For example, an analyst of an elevator system may identify an actor Traveler and a use case "Travel" consisting of "call elevator, enter elevator, select destination floor, exit elevator". But an elevator implementation need not keep track of individual users from entry to exit. Instead, the elevator cabin runs its course, typically as far up as there are requests, then as far down as there are requests, etc. It stops at each floor that has been requested by a cabin passenger or where someone has pressed the Up or Down button. This makes the Travel use case impractical for design purposes because there is little point in giving each traveler a thread. It's more useful to introduce an *inanimate* actor, *Cabin*, that executes its own use case repeatedly, and may have its own thread. The Job in the FMS system is another such inanimate actor.

## 5. Other design approaches for multi-threading

Books and papers on real-time design and on multi-threading in general often include examples of good design but no systematic way to arrive at them. The design advice is often quite eclectic [4]. ELM intends to convey the state-of-the-practice among experienced thread designers to those starting out in a more systematic way. Those systematic design methods for multi-threading that do exist tend to teach what I will call *dataflow threading*. Dataflow threading is where an input is first handled by one thread, then handed to another thread, then to a third one, etc., usually via message queues. Dataflow threading used to be taught based on structured analysis and continues to be taught based on objects.

There are legitimate reasons for dataflow threading. One is to provide run-time configurability. Objects that have their own threads are loosely coupled and can easily be dynamically reconfigured to connect to other objects in new ways. Another reason for dataflow threading is to obtain a provably schedulable set of threads. A set of periodic threads that are scheduled according to the rate monotonic algorithm can be proven to meet given deadlines. This is important in real-time systems with competing hard deadlines. A computation has a hard deadline if it must be completed by a certain time to avoid dire consequences. But most multi-threaded software has no hard, competing deadlines, and even for most real-time systems, only some computations have hard deadlines.

Sometimes, dataflow threading is used for no apparent reason. A classic example is where each layer in a protocol stack has its own thread. Dataflow threading has no concept of optimality, and provides no way to tell whether, for example, a cruise control system architecture with eight or ten threads is reasonable. You can often trim the number of threads, possibly to a single one. With a single thread, the system must no longer be thread safe and can usually be radically simplified.

Dataflow threading has potential problems. First, it relies heavily on context switches since each input tends to activate a cascade of threads, one after the other. In a real-time system, the overhead for all the context switches may be unacceptable even if the system isn't explicitly deadline dependent. If the work of each thread is limited, the context switching may take up most of the time.

A second problem is that the message queues are dynamic data structures that grow with the number of messages queued. In a system with reliability requirements, the designer must en-

sure that no message queue can overflow under normal or abnormal circumstances. This means that either the thread producing the messages must stop when the message buffer is full, or that the receiving thread must remove messages from the buffer even if it cannot process them. One of those solutions must be implemented for every message queue in the system.

## 6. Conclusion

Concurrent threads in software can be patterned after concurrent structures found in the problem domain, just as classes and objects are patterned after domain objects. In entity-life modeling this is combined with a design that minimizes the number of threads and thread interactions.

I have illustrated ELM with small, textbook examples. The approach is more justified in larger and more realistic examples such as the flexible manufacturing system [5, 6, 9], but a complete discussion of the design alternatives in software of that complexity can be rather overwhelming for the reader. The multi-elevator problem is a popular example and not as simple as it is often portrayed [5]. Yet another non-trivial example is the automated switchyard where engines vie for access to track segments [5, 9].

## References

- [1] R. R. Asche, Multithreading for rookies, Microsoft Development Library, September 1993.
- [2] G. Bollella, J. Gosling. *The real-time specification for Java*. IEEE Computer 33:6 (June 2000), 47-54
- [3] G. Booch, J. Rumbaugh, I. Jacobson, The Unified Modeling Language User Guide, Addison-Wesley 1999
- [4] B. P. Douglass, Real-time UML. Developing Efficient Objects for Embedded Systems, Addison-Wesley 1998
- [5] B. I. Sandén, Software Systems Construction with Examples in Ada. Prentice-Hall, 1994.
- [6] B. I. Sandén, *Modeling concurrent software*. IEEE Software, Sept. 1997, 93-100.
- [7] B. I. Sandén, *Concurrent design patterns for resource sharing*. Proc. TRI-Ada, St. Louis, MO, Nov. 1997, 173-183.
- [8] B. I. Sandén. *A design pattern for state machines and concurrent activities*. Proc. 6th International Conference on Reliable Software Technologies - Ada-Europe 2001, Leuven, Belgium, May 14-18, 2001, Dirk Craeynest, Alfred Strohmeier (Eds.), Lecture Notes in Computer Science, vol. 2043, Springer-Verlag, 2001, 203-214.
- [9] B. I. Sandén. *Multithreading in resource control applications*. Submitted to IEEE Computer 2001.