Communicating Process Architectures 2007
Alistair McEwan, Steve Schneider, Wilson Ifill and Peter Welch (Eds.)
IOS Press, 2007
© 2007 The authors and IOS Press. All rights reserved.

High Cohesion and Low Coupling: the Office Mapping Factor

Øyvind TEIG
Autronica Fire and Security (A UTC Fire and Security company)
Trondheim, Norway
http://home.no.net/oyvteig

Abstract. This case *observation* describes how an embedded industrial software architecture was "mapped" onto an office layout. It describes a particular type of program architecture that does this mapping rather well. The more a programmer knows what to do, and so may withdraw to his office and do it, the higher the *cohesion* or completeness. The less s/he has to know about what is going on in other offices, the lower the *coupling* or disturbance. The project, which made us aware of this, was an embedded system built on the well-known process data-flow architecture. All interprocess communication that carried data was on synchronous, blocking channels. In this programming paradigm, it is possible for a process to refuse to "listen" on a channel while it is busy doing other things. We think that this in a way corresponds to closing the door to an office. When another process needs to communicate with such a process, it will simply be blocked (and descheduled). No queuing is done. The process, or the programmer, need not worry about holding up others. The net result seems to be good isolation of work and easier implementation. The isolation also enables faster pinpointing of where an error may be and, hence, in fixing the error in one place only. Even before the product was shipped, it was possible to keep the system with close to zero known errors. The paradigm described here has become a valuable tool in our toolbox. However, when this paradigm is used, one must also pay attention should complexity start to grow beyond expectations, as it may be a sign of too *high* cohesion or too *little* coupling.

Keywords. Case study, embedded, channel, Office Mapping Factor, cohesion, coupling, black-box encapsulation

Introduction

The system we are describing here has been discussed in two published papers [1-2]. It was a rather small project with up to four programmers, running for some time. The result was several hundred KB of fully optimized code in an 8-bit microcontroller. The product we discuss has high commercial value for our company. It is *part of* a new fire detection panel, with one such unit per cabled loop. A loop contains addressable units, for fire detection and other inputs or outputs. (Autronica pioneered "addressable fire detectors" in the late seventies.) Together with fire detectors and fire panels it completes Autronica Fire and Security's range of products. The product described here is called *AutroLooper* and is not available as a separate product.

Several AutroLoopers communicate (over a text based protocol) with a "main" processor on the board. The team, which developed that system partly, used object orientation, UML and automatic code generation. The degree of "office mapping factor" in that system is not discussed here.

Working in industry on rather small projects, we seem to get on with the next project as fast as possible when one project has been properly tested, documented and then “closed”. This paper is meant as a reflection over an observation, squeezed into a glitch between the project proper and its follow-up, a linear expansion of the product we describe here.

This case study tries to list and explain our experience. We have not done any comparative study or discussed what we could have done differently. There are no metrics involved in this industrial “case observation”.

Observe that in this document we often refer to what “we” did and how “our” implementation is. Please see [1-2] for all these cases.

In this document, a *process* is what some embedded real-time systems mostly consist of, closely related to the *thread* and *task* terms.

1. The Office Mapping Factor

By *office mapping factor* we mean the degree to which a program architecture may be mapped to separate offices for individual programmers to implement, and the degree to which that implementation work is complete (high cohesion) and undisturbed (low coupling). We assume that module properties like high cohesion and low coupling cause high office mapping factor, and that high office factor is wanted.

Prior to the mapping to offices, the program architecture would have gone through a partitioning phase mostly based on functional decomposition. We believe that high office mapping factor gives high satisfaction in the office, as the programmer knows what he should do, and does not have to think about how the others do their parts.

Figure 1. Software architecture and office floor plan

The architecture we built (glimpsed in Figure 1) was, to a large extent, a consequence of informal meetings around a whiteboard, and an understanding of the semantics of our processes and channels (pictures of those whiteboards are our minutes). This way the architecture itself reflected each team member's speciality. Management had picked people with some, but not much overlap in knowledge. We believe that this contributed to a higher office mapping factor. Not only beneficial for development, we also think that as time passes and maintenance needs to be done, getting different people on the project will be easier with this high office mapping factor.

Office mapping could also allow that one programmer does more than one process. It would mean that he would mostly need to relate to the communication pattern between his own processes. Role wise he would first do his one job, exit the office and enter it again for another. And mostly forget about the internals of his finished work only to concentrate on the present.

2. High Cohesion and Low Coupling and the Office

In the context of a confined office, having high cohesion means that the programmer knows what to do and is able to fulfil the task having little communication or coupling with the others in the team. He would not need to know how the others solve their processes.

Cohesion and coupling in this case seem to be inversely related. The less complex the protocols between the processes are, the more complete is a process' work.

However, the programmer must understand that the protocol or contract must be adhered to 100%, and he must know that he cannot "cheat" by sharing state with the other processes – other than by concrete communication. Going by the agreed-upon cross-office rules (the protocol message contents and sequence semantics) also gives a concerted feeling: one is part of a real team.

But, is this not is how programming has been done since the fifties? For procedural programming languages a function has always taken parameters and returned values. A function has had high cohesion, and coupling has been through the parameters. However, concurrent constructs (or even Object Oriented constructs) may in some cases be at stake with the cohesion and coupling matters. Processes may be implemented more or less as black boxes and may have subtle communication patterns. The lesson learned with occam [3] in the eighties and nineties was that the clean process and communication model was worth building on. This is what we did in this product. occam (without pragmas) ensured complete "black-box" encapsulation.

3. Process Encapsulation Check List and the Office Mapping Factor

A check list for process encapsulation might be like this (below). One could say that "wear" of the office mapping factor may be caused by:

1. For a process, not being able to control when it is going to be used by the other processes. Serving "in between" or putting calls in a local queue makes it much more complicated to have "quality cohesion". *[Java classes, for example, cannot prevent their methods being called (except, through 'synchronized', where other synchronized methods can't run if one is already running). But that does not prevent the method from happening. It just delays it. It cannot be stopped, even if the object is not in a position to service it (like a "get" on an empty buffer). Not being able to control when it may be used by other processes means that things of*

importance to a process may change without it being aware of it. This trait is further discussed here, since the other points (below) only to a small degree are valid in our project.]

2. Incorrect or forgotten use of protection for inter-process communication. *[So, use a safe pattern for it – as we have used in this project.]*
3. Communication buffer overflows would most often cause system crashes. *[We use synchronous channels, which cannot cause buffer overflow during inter-process communication. However, buffer overflows on external I/O are handled particularly by link level protocols.]*
4. Mixing objects and processes in most languages and operating systems, since most languages have been designed to allow several types of patterns to be implemented. *[We use the process definition from a proprietary run-time system which gives us occam-like processes and intercommunication in ANSI C.]*
5. Too much inheritance in OO. There is a well documented tension between inheritance and encapsulation, since a subclass is exposed to the details of its parent's implementation. This has been called "white-box" encapsulation [4]. This is especially interesting here, if a process is an instance of an object.
6. Aliasing of variables or objects. Aliasing is to have more than one name on the same memory cell simultaneously. This type of behaviour is required in doubly linked lists, but would cause subtle errors found well into the product's life cycle. *[We don't think we have these.]*
7. Tuning with many priorities. Priority inversion may soon happen. How to get out of a potential priority inversion state may be handled by the operating system. However, many smaller systems do not have this facility. Therefore design with many priorities is difficult to prove not to have errors. *[We have medium priority for all Processes (scheduling of them is paused when the run queue is empty), low for Drivers (called when ready queue is empty), and high priority for all interrupts (which never cause any rescheduling directly). This is a scheme, which holds for the rather low load that our non pre-emptive system needs to handle.]*
8. Not daring to do assert programming and instead leave unforeseen states unhandled or incorrectly handled. System crashes by assert programming puts the pain up front, hopefully before any real damage is done. However, it also removes subtle errors at the end. *[We have used a globally removable macro for most asserts, and we have hundreds of them. Overall, they seem to cause so little overhead and such high comfort that we have not felt it correct to remove them. This author thinks of them as self-repairing genes of our "software cell": on each iteration with the programmer, the cell enters longer and longer life cycles.]*

4. Mapping

The mapping of the processes was easily done, since the team members in our case had their specialities. After all, that is why management had co-located us in the new facilities with subunits of 6-and-6 offices around small squares (Fig.1).

The application proper was handled by two persons, the intricacies of the loop protocol by a third and the internal data store and high level text protocol by a fourth. And importantly, the fifth – a proper working project leader. One of the five also had responsibility for the architecture (this author). His experience (of which much was with occam and the SPoC implementation [5]) projected, of course, that way of thinking on the architecture. This was much discussed in the team prior to the decision, but with the some

20 years of average embedded real-time experience for the rest of the team, the concepts were soon understood and accepted. Even, with some enthusiasm.

With the mapping of processes to offices (in most respects here, "office" really means "person"), we had a parallel architecture that also enabled parallel development. We think this shortened development time – the higher the office factor, the greater the shortening.

5. The Closed Door Metaphor

The first point in the numbered list above mentions "not being able to control when it is going to be used by the other processes" as a potential problem for the office mapping factor.

With the mapping scheme, terms from computer science become metaphors for office layout or human behaviour. Below, when a *door* is mentioned, it both means the physical door to the office, a person's possibility to be able to work undisturbed for some time, and that an embedded process is able to complete its work before it goes on to handle other requests. It is, as one can see in this paper, difficult not to anthropomorphise the behaviour of the computer.

When we talk about being undisturbed, we mean both undisturbed programmers and processes. Low cohesion means a good protocol between processes and a corresponding small need to discuss further with the other team members, because it is all in the protocol description.

Below, we outline three important facets, which in our opinion, may influence the office mapping factor: *sloping*, *non-sloping* and *sender-side sloping* message sequence diagrams. These terms are probably invented here, as eidetic tools.

5.1 Sloping Message Sequence Diagram

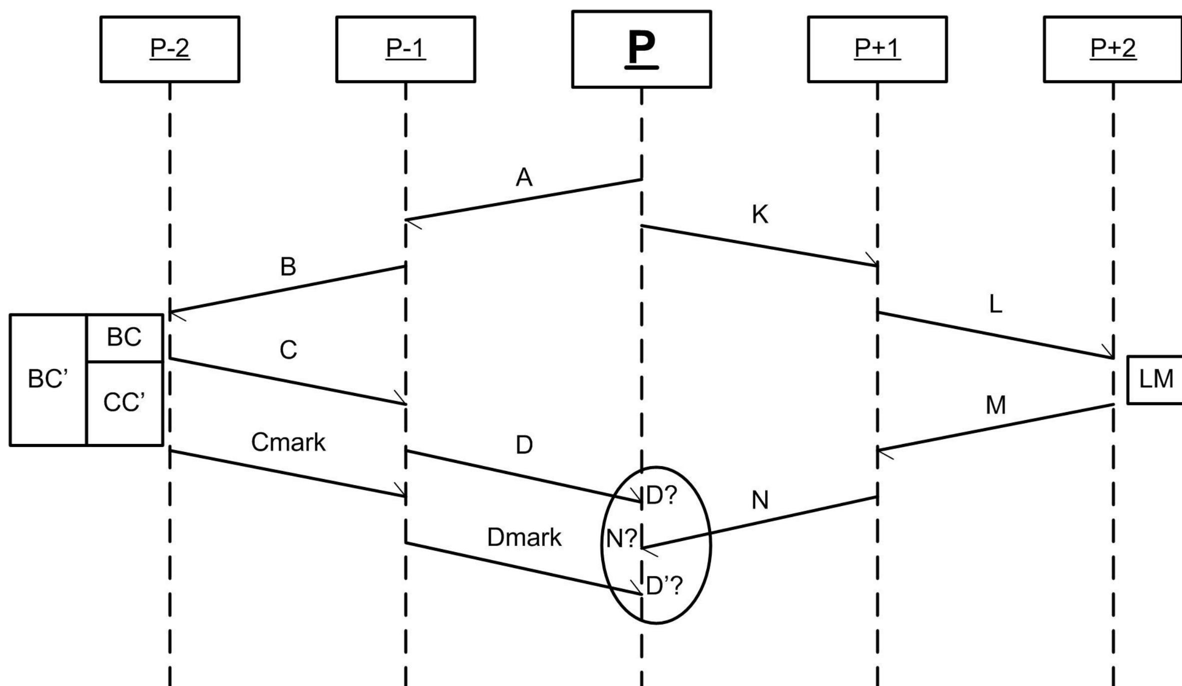

Figure 2. Asynchronous communication is "wide open"

A sloping message diagram describes a communication scheme where a process (the top boxes in Figure 2 show 5 of them) sends with no blocking or synchronization. This is called

asynchronous communication. Here sending takes time, meaning that the time line (the vertical lines, where time flows downwards) is present both for sender and receiver. Sender sends at a time, the run-time system buffers the message, and the receiver receives it some time later. The important thing here is that neither party blocks or is descheduled for the sake of the communication. *Time flows and exists* for them – to do other things as required.

This communication scheme is much used. However, for a concurrent design it is only one important tool in the toolbox. If the asynchronous behaviour is wanted, it is the right tool. Otherwise, there may be certain advantages for *not* using this scheme.

In Figure 2, P sends two orders: A (first) and K for which it needs to have confirmation messages D and N. The middle, left box shows that the time for P-2 to respond is either time B to C ("BC") causing reply C, or BC' causing the same data to be sent later as reply Cmark. Depending on whether P-2 has to have the extra CC' time, the confirmation ordering back to P (of its original A then K messages) will be switched.

Not knowing which response comes first is illustrated by the question marks ("D?", "N?" or "D′?") in the centre bottom ellipse – to indicate that the acknowledges are indeterminate with respect to when they arrive.

Sometimes, the order and the confirmation must be in phase. Either it *must*, or it is *simpler* this way – with less internal complexity. Relying on *any* order of the replies could be equally problematic. With synchronised channel communication, we can be in charge on this point: we could decide to listen (and hold) any way we want.

With the scheme above, it would be better to make the design able to handle order swapping. Easy: just get on with the next job in P when the number of pending replies have reached zero.

But what if, instead of merely a swapped order, completely unrelated messages arrive from other senders? Then, it is not so easy: the process soon becomes a scheduler for itself. This adds complexity, because in the deepest sense every program has to know something about how the other parties with which it communicates behave internally. "*Can you wait a little in this case?" / "I will send you an extra confirmation when you may go on."* This kind of out-of-office conversation could be a warning sign that the next time the programmers enter their offices, it will take longer. And then, longer again. We do not have *WYSIWYG* semantics.

5.2 Non-sloping Message Sequence Diagram

Here we describe another tool in the toolbox: the *synchronous blocking communication* scheme. (Note that *blocking* means descheduled or set aside until the communication has happened. It does not mean making the processor slower or unable to do any meaningful work. The throughput of a synchronous system might even be higher than an asynchronous system, provided adequate buffering and asynchronicity is applied at the terminals.)

In Figure 3, we see messages drawn by an offline log client that we had made for us. Here, each time-stamp has a complete message; it is the time of the communication. The "rendezvous" happens there. It is *not* the time when the first process on the synchronous one-way channel gets descheduled. (In our case, we have a non-preemptive run-to-completion scheduler beneath.) At this point, *time stops* for this process. Time, of course, flows; but not for the descheduled process. It may only be scheduled again when the second process has been at the other end of the channel, and the run-time system has **`memcpy`**'d data across, directly from inside the state space of the sender to inside the state space of the receiver.

Although deadlocks may happen with synchronous systems unless safe patterns to avoid them are used [1-2], the synchronous communication scheme has some advantages.

Firstly, there is no possibility of buffer overflow.

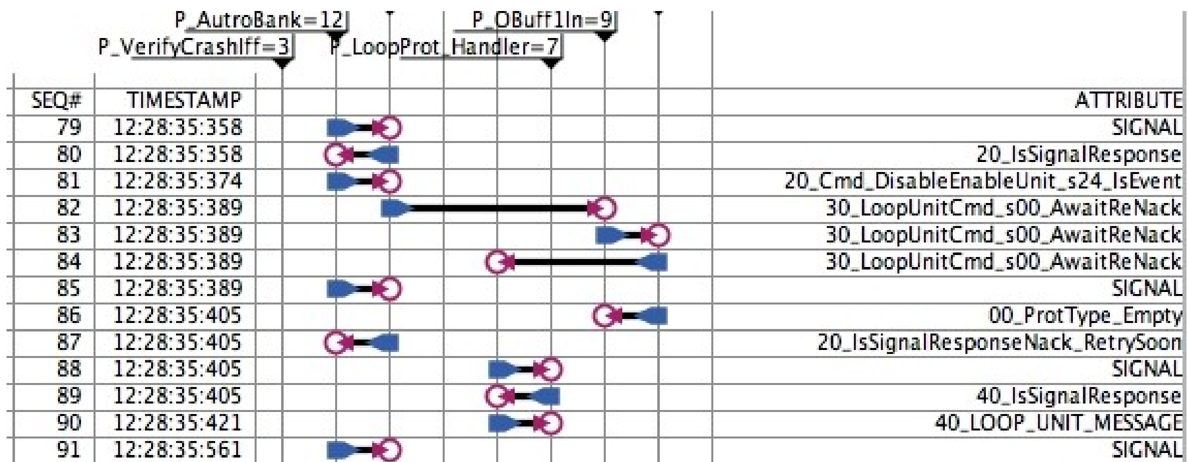

Figure 3. Synchronous communication has "door"

Secondly, and this is stressed here, the receiving process need not "listen on the channel" when it is active doing other work. It may communicate with other processes and not worry about whether there is any other process waiting on the ignored channel. Observe that it does not care, the process and data-flow architecture has been designed so that the waiting of one process does not have any consequences which the busy process need worry about. If, far above these processes there is a conversation going on with another processor, which needs a reply even if several processes are blocked further below, the design must have considered this[1].

Not listening on the channel is equal to having the office door shut. Building our system with this paradigm, we believe, has given lower coupling and higher quality cohesion. All communication in the system we describe here is based on this. We believe that this is one of the reasons why we seem to have a high office mapping factor[2].

Observe that the ALT construct makes it possible to listen to a set of channels or an array of channels, with or without timeout. This listening is blocking and – according to the door metaphor – individually closable. So, there is no busy-polling of channels (if this is not what we really want – at some asynchronous I/O terminal).

5.3 Sender-side-sloping Message Sequence Diagram: Pipes

It is possible to have asynchronous *sending* and blocking *reception* if we use *pipes*. With pipes there is one queue per pipe. A listener may then choose not to listen on a pipe. Most often a pipe may have at least one "buffer". Some times they block when they have received N elements, some times they just return a "full" state. Often a pipe cannot have zero buffers – which would have allowed for true synchronous messaging.

It is possible to build this kind of system also with a composite buffer process and synchronous blocking channels. We have one in our system, and it contains two small processes (it may be spotted in Figure 1 as *P_OBuff1In* and the process below it).

[1] In our case, it is handled with active process "role thinking". An in-between process is a *slave* both "up" and "down" and a mix of synchronous blocking data-rich and asynchronous data-less signals is used.
[2] The run-time layer we used to facilitate this we built on top of an asynchronous system. This was considered (correct or not, at the time) to be the only viable way to introduce this paradigm into the then present psycho-technological environment.

A pipe construction is a versatile tool. However, using it *may* give a little lower office mapping factor. We may have to know more about the sender: *"Does it block now? Should I treat it now? When does he send?"*. And the receiver: *"Is it listening now? May I be too eager a producer? How do I handle it if I have too much to send? Should I change state and go over to polled sending then?"*.

The fact that time has not stopped for the sender, after a sending, may therefore be a complicating factor.

6. Scope

The system we have described contains medium to large grained processes, which contain long program sequences. Whether the office mapping factor has any significance for a system with small state machines realised as communicating processes, we have not investigated.

Also, as mentioned, we have not done any comparative studies of other paradigms, like OO/UML. For the scope of this article, whether more traditional programming or OO/UML is used *inside* each office or process, is not discussed. It is the mapping of the full process data-flow architectural diagram onto offices that is discussed.

Taking a 100% OO/UML architecture, with only the necessary minimum of processes, and investigate the office mapping factor would be interesting.

7. Warnings

7.1 High Cohesion Could Cause Too High Internal Complexity

With high cohesion, there is of course a possibility that a person may sit so protected in the office that the system would organically grow more than wanted. Also, inside a process one has to watch out for the *cuckoo* entering the nest. It is hard to see every situation beforehand, but still it is also a good idea to analyse and design to some depth. Within a real-time process, any methodology that the programmer is comfortable with should be encouraged. This, of course, could include OO and UML.

7.2 Low Coupling Could Also Cause Too High Internal Complexity

We saw during our development phase that, if we modified the architecture, we were able to serve the internal process "applications" to a better extent. The first architecture is described in [1] and the second in [2]. However, not even [2] needs be the final architecture. With low coupling, we then have tools to insert new processes or new channels, or to remove some. This could be necessary if we discover that we do too much in a process. To split (and kick out the cuckoo) may be to rule – but it does not have to be. These considerations should be done any time an unforeseen complexity arises, if one has a feeling that it is an architectural issue. On the second architecture we introduced an *asynchronism* with the introduction of a two element (composite and *synchronous*) data buffer process. This led to *more* coupling (communication) and *less* cohesion (state) in the connected processes – but ultimately to lower complexity.

8. Testing

Inside each office individual testing was done, in the more traditional way, on smaller functions, with debugger and printouts.

However, interesting to see was that testing of the processes was almost always done *in vivo*, with all the other processes present – on each office's *build*. The reason that this was possible was that with the parallel implementations, the protocols were incrementally made more and more advanced, on a need to have basis. It seemed like the tasks were well balanced, because there was not much waiting for each other. Programming and testing was – almost, synchronous.

We kept track of each error and functional point. Before release of version 1.0 (yet to come) we have zero to a few known bugs to fix. It seems like it is easy to determine which office should do an error fix. There have been little errors in interprocess communication. It has been easy to determine where an error might be located.

9. Other Teams

We released incremental new beta versions for the other team to use, mostly on set dates. The date was the steering parameter, not a certain amount of functionality. We felt it was easier to keep the enthusiasm this way, and that it helped the office mapping factor. This has briefly been described in Norwegian in [6].

10. Conclusion

It seems that a successful mapping from a process data-flow architecture to offices is possible. Simultaneous programming with high cohesion (in process and office) and low coupling (between processes and offices) is defined as high "Office Mapping Factor", a term coined here. It seems like the product we have developed, described here and in two other publications ([1-2]), has benefited from the architecture chosen. We have not studied whether other methodologies would be better or worse off, since this paper is an industrial *case observation*.

References

[1] Ø. Teig, "From message queue to ready queue (Case study of a small, dependable synchronous blocking channels API – Ship & forget rather than send & forget)". In *ERCIM Workshop on Dependable Software Intensive Embedded Systems*, in cooperation with *31st. EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA)*, Porto, Portugal, August/September 2005. Proceedings: ISBN 2-912335-15-9, IEEE Computer Press. [Read at **`http://home.no.net/oyvteig/pub/pub_details.html#Ercim05`**]

[2] Ø. Teig, "No Blocking on Yesterday's Embedded CSP Implementation (The Rubber Band of Getting it Right and Simple)". In *Communicating Process Architectures 2006*, P. Welch, J. Kerridge, and F.R.M. Barnes (Eds.), pp. 331-338, IOS Press, 2006. [Read at **`http://home.no.net/oyvteig/pub/pub_details.html#NoBlocking`**]

[3] Inmos Ltd.: "Occam-2 Reference Manual", Prentice Hall, 1998.

[4] E. Gamma, R. Helm, R. Johnson and J. Vlissides, "Design Patterns: Elements of Reusable Object-Oriented Software", Addison-Wesley, ISBN 0-201-63361-2, 1995.

[5] M. Debbage, M. Hill, S.M. Wykes and D.A. Nicole, "Southampton's Portable Occam Compiler (SPOC). In *Proceedings of WoTUG-17: Progress in Transputer and occam Research*, R. Miles and A.G. Chalmers (Eds.) , pp. 40-55, IOS Press, ISBN 90-5199-163-0, March 1994.

[6] Ø. Teig, "Så mye hadde vi. Så mye rakk vi. Men får de?" (In Norwegian) ["We had so much. We made it to this much. But how about them?" In *Teknisk Ukeblad* internett February 2006 and #15, May 2006, page 71. Read at **`http://home.no.net/oyvteig/pub/pub_details.html#TU_feb_06`**

Øyvind Teig is Senior Development Engineer at *Autronica Fire and Security, a UTC Fire and Security company*. He has worked with embedded systems for some 30 years, and is especially interested in real-time language issues. See **`http://home.no.net/oyvteig/`** for publications.