

Pragmatic Design with Ada

Caroline LANGENSIEPEN

Time High Data Ltd, Hawkstone, Sibthorpe, Notts NG23 5PN

Abstract This paper examines some of the problems associated with the design and implementation of large real-time systems in Ada. Methods for reducing these problems are described, including the identification of patterns within the behaviour of the system leading to the use of standardised patterns for the structure of the solutions. Case studies from real-life projects are used to illustrate these ideas.

1. Introduction

Ada is normally considered the language of choice for large, complex real time systems, certainly if they have some safety, reliability or security requirement. Unfortunately such systems are prone to problems associated with their scale. This paper examines a few heuristics which may assist in solving these problems.

2. Problems in large projects

The main problem with large systems is communications. Other risks - timescales, underestimating effort, performance are true in all projects, but are compounded here by failures of communications. Typical aspects of the problem are:

- *Documentation.* On projects of more than 20 designers (particularly if they are working at geographically separated sites), design requires documentation standards for internal technical notes, codes of practice, formal reviews, reissue of notes etc. So any design change has significant overheads.
- *Team Consistency.* Because Ada has only been in use for a relatively short time on large projects, the staff experience will be very variable. There will be some staff with applications knowledge but little Ada. They will use a style they know, no matter that it may be inappropriate to the overall scale of the system. Some will be Ada experts but with little experience of the application. They will use sophisticated constructs which are hard to understand.
- *Solving the Wrong Problem.* In the past, the main problem on small projects was performance. This assumption has carried over to large projects, leading to an early concentration on performance at low level, determining data representations etc., even before the structure of the design has been established.

3. Heuristics

Some strategies to improve communication and aid the design process are now proposed. The

larger the scale of the project, the earlier in the lifecycle the problems have to be attacked and strategies set in place, since it becomes very difficult to adopt them later.

4. Object Oriented Design and Firewalling

Object Oriented Design (OOD) is a good technique to apply on large projects. The main reasons are its ability to allow the isolation of problems, encapsulation of data, and firewalling risk - in other words, to ensure the right level of communication between applications. Its advertised features of maximising reuse, inheritance etc are of much less interest to the manager of a particular project unless the features enable him to reduce his cost to completion - the ability to reuse the code developed for one project on another cut no ice if the first project gains no direct financial benefit from it.

In any case, Ada is not really an OOD language, but the package structure enables Abstract Data Types (ADTs), Abstract State Machines (ASMs) etc to be cleanly built. Its lack of an inheritance capability seems not to be a problem, as in the situations where it is used, there does not seem to be that much need for it. Large systems tend to be wide rather than deep - ie there are very few layers of specialisation on any class, but there may be very many classes.

If one has a very wide staff profile of experience, then OOD allows one to set novices to work in areas of low risk, and be confident that they cannot harm those areas of high risk. Areas where application knowledge is most important can be isolated from those "general software" areas - giving the former to applications experts and the latter to Ada experts.

It is often worth providing an interface package to an area of functionality to be used by all other clients, even if they could legitimately get in at a lower level. This firewalling ensures that if that functionality moves to another processor, then only the interface package body changes, to make VME calls, RPCs across Ethernet etc. Of course, the decision where to place these potential hardware boundaries depends on the data bandwidth of the boundary as well as its level of abstraction. This may seem obvious, but staff with experience only of small projects often do not consider the changes that might have to occur to the architecture of a large system. It is the capability to handle such changes that have to be built into the design methodology as well as the design itself, since the maintenance period on such a system could be up to 15 years.

In addition, when (not if) initial performance is much lower than required, individual areas can be "tuned" without impacting the rest of the system. Of course, if the system was designed without regard to the overall constraints of the hardware and primary customer requirements, then tuning will not be enough, but a "reasonable" design will benefit from the encapsulation first, and then worrying about performance later. The benefits gained by easier testing and integration outweigh the overheads particularly for very large systems.

5. Constructive Structural Distortion

"Lapses from purity" may be necessary in an otherwise object based design for practical and political reasons. Practically, there may be some aspects of the system which are too large and complex to allow the individual classes to deal with. Normally such functionality occurs at the periphery of the system - where the abstractions within it collide with the constraints of the user or devices attached. For example, the human computer interface (HCI) where driving the display devices may be too complex and device dependent for the individual classes to use. This is one area where tuning is almost inevitable, and since it is more likely that the tuning

will be needed for HCI in general rather than the display of one class in particular, then this area should be isolated - even if its relationship with the rest of the system is now somewhat distorted.

Politically, it may be necessary to abstract on the basis of customer requirements rather than what seem the "best" boundaries for implementation. Use of the client view rather than the designer view can be beneficial because customers can see their prime functionality in the design, requirements traceability becomes easier and the inevitable specification changes tend to be more isolated, since they occur within the boundaries as seen by the customer.

Although the stability of the "Problem Domain Component" of design has been noted elsewhere (eg. by Coad and Yourdon [1]), the importance of ensuring that the design of this component closely maps to the customer view has not been stressed. Improving requirements traceability aids system integration, testing and acceptance, since these final critical stages of the system life-cycle are strongly tied to the requirements matrix, which is the formal way in which the customer communicates his needs to the designers.

6. Conventions and Nomenclature

One of the most cost-effective ways of improving communications is by establishing conventions early on. Most projects have standards for the diagrams, documents and file names. However, these will not tend to address the semantic meaning involved in the names of packages, operations and variables. More relevant is to set up a dictionary of terms, preferably taken from the problem space, which are used in a consistent way throughout the project. If the applications experts and the customer invariably refer to some concept via an abbreviated name, then that is the name that should be used throughout the design - for example 'package RNSH1C' rather than 'package Sub_Harpoon' in a Naval simulation.

Conventions for names, particularly of operations, ensure that clients know what they are getting when they try to use an offered service. A typical convention might standardise the meanings of the container class names eg Set, Queue, List, Stack, and the operation names eg Initialise, Shutdown, Add, Remove, Get, Put, Update. Where additional semantics are involved, the names should show additional sophistication. For example, Database_Set.Add (The_Item) is a simple addition of a new item to a container class, whereas Events_Set.Register (The_Event) will not only add the Event to the container class, but will do something else such as raising an alarm.

7. Don't Panic about Tasks

The risks of early concentration on task removal and other implementation details at the expense of a clean structural design are compounded by the effects of the staff profile. Experienced Ada users from non-real time backgrounds will liberally use tasks, even when no actual parallelism exists. Their software will work but it won't "perform" - that is achieve its response target or processing budget. Domain experts (who are often very wary of overheads) will be niggardly with tasks, trying to get away with a cyclic behaviour even when truly asynchronous events are occurring - then using machine code or just faster cycles to improve their runtime response. This code may go quickly, but getting it to work is difficult.

To achieve consistency of style, use of tasking to desynchronise large areas of functionality should be encouraged, but further down within an application, designers must question whether additional parallelism or desynchronisation is really necessary. Performance of a

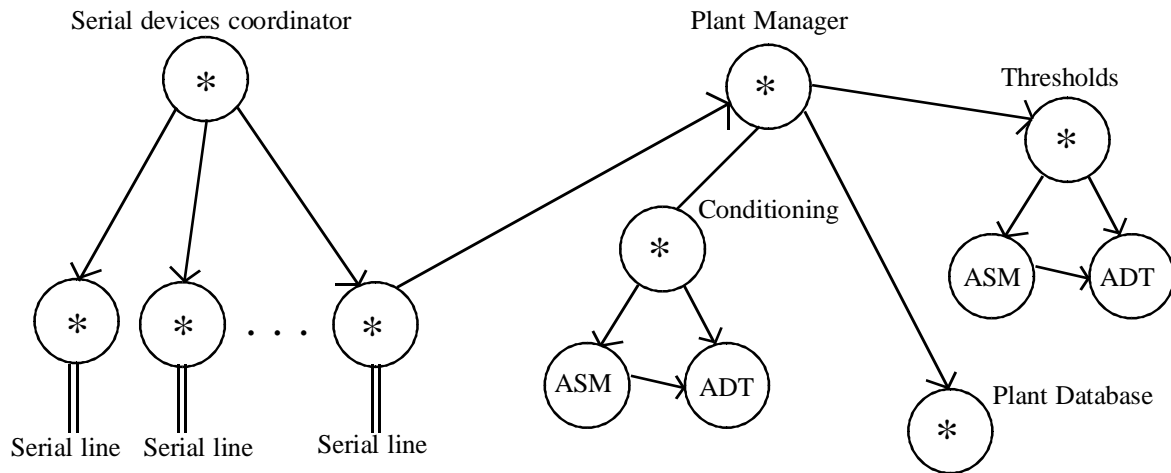


Figure 1 - Plant signal monitoring

system will not benefit from more tasks than there are processors or independent parallel i/o devices, but at the early stages of design there is no point worrying about them too much. If the task entry points are hidden behind a procedural interface, then from experience it seems to be comparatively simple to remove them at a later stage and by using the scenarios relating to the primary functionality, designers can easily count context switches, and test their design at each stage of iteration.

Consider a section of a project required to monitor a large number of plant variables via serial devices. Each of these plant signals requires conditioning (performing mathematical transformations upon it), testing against a set of threshold values, and recording the value of the signal in a database. The resultant design will therefore be based on operations on individual signals. Moreover, because the databases associated with the thresholds and treatments are also quite complex, the design protects them against concurrent access by their own tasking interface. A typical structural design for this is shown in figure 1.

By doing a simple count of context switches during the scenario of receiving a signal, processing and recording it, it soon becomes apparent that the tasking interfaces (as indicated by * in the diagram) would require many thousands of context switches a second for a typical large industrial application.. Yet the only really necessary tasks were those due to the true parallelism provided by the serial lines, a controller to initiate and query the states of the cards on these lines, and a tasking interface to ensure that external queries on the database would not clash with writes to it.

This removes the need for tasks in the areas of the conditioning, thresholds and database storage, leaving only one in the Plant Manager. Moreover, in the real situation from which this example was taken, the signals were received in groups of up to 256 per serial line interrupt, and by exporting a service that worked on the granularity of the hardware (ie the group of signals) rather than just a single signal, the switches were reduced still further. Context switches were reduced by a factor of 100, and this was achieved with no structural changes to the design and the provision of only a single extra exported operation in the Plant Manager. If the design had been started with the premise of using minimal tasks, then it is likely that the structure would never have been exposed - resulting in something very hard to test and maintain.

8. Find Patterns and Reuse them

Because of the 'width' of the large systems Ada tends to be used on, reuse of code tends to be minimal. Apart from using the same trivial set or queue construct, there is not much that be

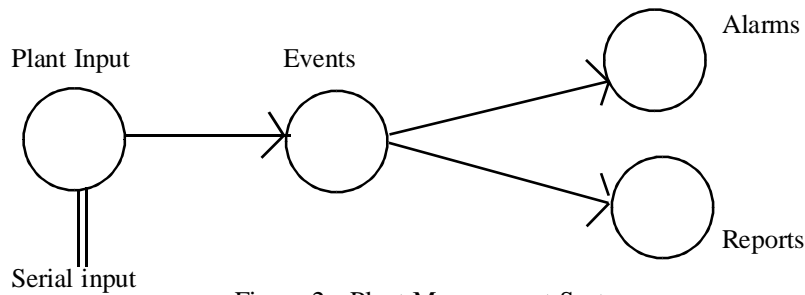


Figure 2 - Plant Management System

reused from one application domain to another. Reuse of design also tends to be minimal unless obvious variants are being built. Reuse of 'meta-design' (that is, the principles and patterns used within the design) is possible and worthwhile.

Finding patterns in the problems and reusing the same structures of solutions in different areas helps in two ways. Once a solution has been found, finding the same pattern in the problem space of another application means the solution can just be ported. And using the same pattern (assisted by consistency of package and operation names) means that a maintainer can recognise it elsewhere and reduce his learning curve when looking at an unfamiliar area of software. The communications throughout the project are improved - everyone is talking 'the same language'.

8.1 Patterns in the Large

The first pattern to look for is the overall shape of the system to be developed. Three real-life problems will be considered as examples

a) Plant Management and Monitoring

This very large project for the monitoring of an industrial plant consists fundamentally of a set of serial devices (as described earlier), writing the status of the plant to a database, causing events to be generated if significant changes of state occur. The events and associated complex logic trees give rise to consequential actions - alarms are raised and reports may be generated (figure 2). The structure is very encapsulated, with each application area performing its functionality and then triggering another area to perform its behaviour as if by a control flow.

b) Command Team Training Simulator

This large project, to train a submarine command team, consists of a simulated scenario of ocean and vehicles, sonar simulations which build their 'pictures of the world' based on this scenario, and a simulation control module which allows the instructors to modify the scenario and examine the states of the sonar simulations (figure 3). This structure is very layered, with

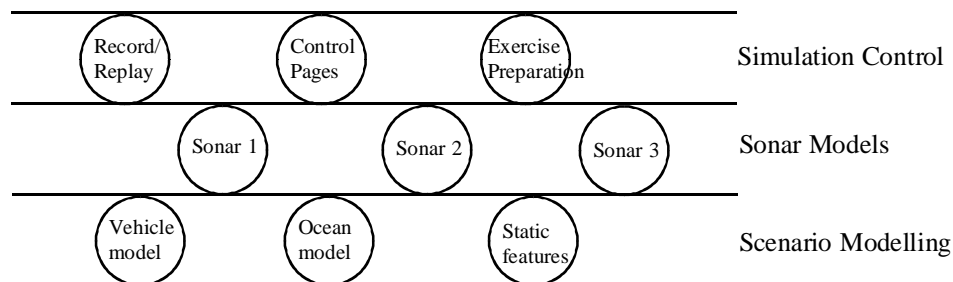


Figure 3 - Command Team Trainer

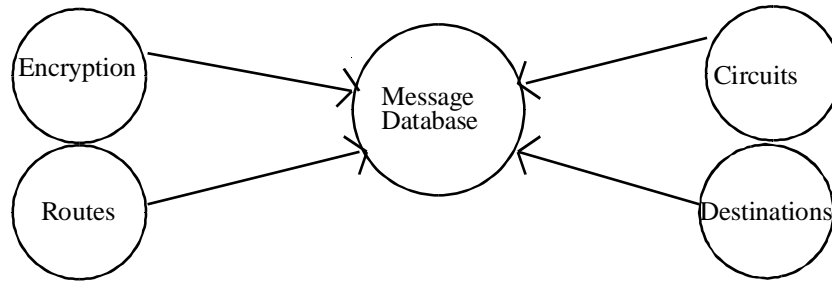


Figure 4 - Message Switch

all the simulations depending on the whole of the scenario, and the simulation control having a view of everything.

c) Message Switch

This smaller project, to receive, analyse, and route messages across a wide area network, consists essentially of the message database, with 'satellite' facilities serving this central concept. There may be facilities to set up routes, to engage and control encryption, to set up names of destinations etc, but all these are subordinate (figure 4). The structure is very centralised, with only one primary abstraction.

Note that nothing is 'purely' one shape. The Plant Management system might have a generalised layer for network communications, the message switch might have an event driven chain of side effects, but the primary shape is based on the primary functionality. Given these overall shapes, Ada will tend to suggest different development strategies:

a) There is not much commonality from one application area to another here, so given a definition of the very restricted interface provided to clients, each large area of functionality can be designed in parallel.

b) This structure means that every layer is very dependent on the layer(s) below it, so development has to be very bottom up, by defining the base scenario types (mass, time, length, acoustics etc) and the top scenario exports before then going on to work on the sonar simulation details.

c) In this situation, the message database exports must be very well developed first. Only then is it worth going on to develop the other lesser abstractions, since any change to the message will have an effect throughout the rest of the design.

8.2 Error Handling in the Large

One of the major problems with a large system is deciding on a consistent error and exception handling strategy that can be applied throughout the project. Of course, certain areas such as HCI have an obvious strategy, since errors in input are not really 'special' or exceptional, and so must be dealt with very locally. However, for the remainder of the system, the error handling will depend on the patterns seen so far.

Figure 5a occurs where the dependency is wide, and cannot be segregated. For example, in the simulator described earlier the sonar depends on the ocean model, the vehicle model and practically every other element of the scenario. The simulator control has an equally wide dependence. Thus as soon as a fault occurs, say in the ocean modelling part of the scenario simulation, it affects all the sonars being simulated, and there is no point continuing. In that case, the fault may just as well be raised as an exception, and that allowed to propagate through the whole system.

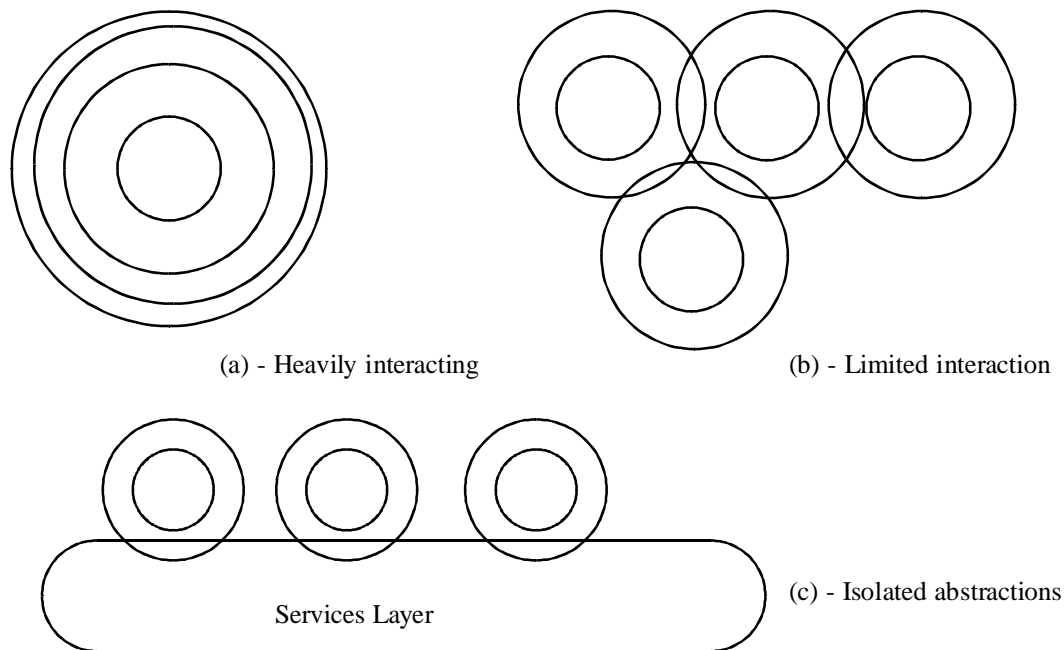


Figure 5 - Patterns in Dependency

Figure 5b has layers of functionality with high dependence, but then the outer layers of these interact with the outer layers of other applications. This is obviously true of the plant monitoring system, where there are layers within each application, but they only interact in a very limited way. The Shlaer-Mellor domain idea [2] can be mapped onto the layers in these applications. In this case, the existence of a fault deep within one domain should cause it to close down immediately, but a controlled closedown of the other domains can be performed since the interaction is so limited that the fault can be considered as localised.

Figure 5c takes this further, as in an ambulance command and control system. In these situations there may be some common layer, but a fault in one area need not impact another. Individual objects can be considered as isolated rather than the classes or class categories. This differs from cases like the plant input monitoring system shown earlier, where the signals are generated by common hardware in groups, and so cannot safely be considered as isolated. However, the incident in an ambulance command/control system is independent of other incidents. Allowing the fault to be isolated to the incident in which it occurred, and tripping out to manual only on that incident, ensures that the rest of the system can continue to function. In these safety related cases, losing the whole system due to a fault in a single object and forcing it to go to wholly manual operation reduces safety rather than improving it.

Later it will be seen where these layers can be considered to interact, and therefore where any exception handlers should occur.

8.3 Patterns in the Medium Scale

The patterns used in the medium scale tend to depend on their position in the application. At the periphery of a system, where the system interacts with humans or disc/tape devices, the behaviour is very different to that of the applications surrounded by other parts of the system.

a) Puppet Masters

Where the system meets the outside world, it has to translate commands from a generalised, non-abstracted form (eg keyboard input) into operations on the abstractions on the system. The knowledge of the abstractions contained in the semantics of the keyboard input has to be

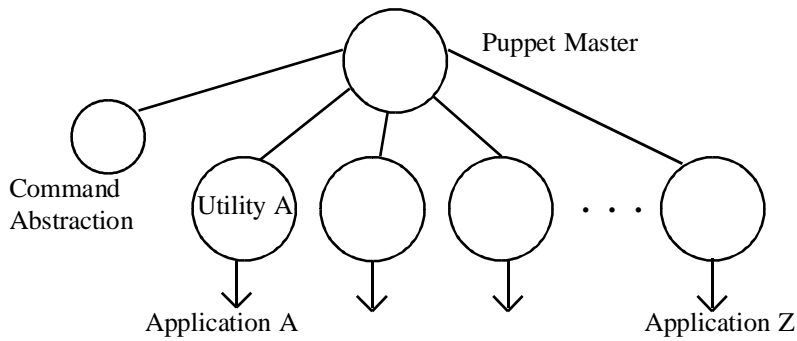


Figure 6 - the Puppet Master

converted into operations specifically acting on those abstractions. Thus the form tends to be of a transaction centre (a class utility) which accesses the generalised input/output (probably through additional layers of refinement and abstraction) and an associated ADT which expresses the final abstraction of the command or query (figure 6).

This transaction centre acts as a 'puppet master' pulling the strings of all the applications in the system as a result of the command. The ADT contains the semantics of the command, distilled away from the syntax or rules associated with the format of the interface. For an HCI layer which performs queries and commands on various applications, the ADT would express a verb (modify, delete), a subject (class name + id/name of object instance), and adjectives (attributes to modify). This ensures that whether the input is via a WIMP, preset hard keys, or standard keyed text, the abstracted command is the same. Below the puppet master, there are agent utilities which can interpret the adjectives to ensure correct parametrisation of the final call to the application. This structure ensures that not too much knowledge is forced up to the transaction centre, and the layers protect it against change, since the most likely change is in the parametrisation of specific operations on an application.

Another example of this is where a system has to have an associated simulator. Here, a text-based script is analysed and converted into script commands, which are then applied to various application areas. The advantage of this method is that the basic system can be written without worrying too much about the simulator. The simulator-specific functionality in the applications can be 'bolted on' as separate sub-packages within the top layer of each application. On top of all this sits the 'Script Puppet Master' which uses the script to run the applications by actioning their normal exported interfaces plus the new specific functionality.

A similar construct to this is known as the "Notifier" in SunView. The important point is not that this construct is new, but to recognise where it has to be created and used.

b) Managers

The most fundamental structure that is of use to a large system is that of an abstraction represented by an ADT and a container class to store instances of this abstraction. There is nothing going on but the addition and removal of various items from the container, and the examination and modification of their attributes.

Most situations in large real-time systems are more complicated than this. There may be side effects which have to be ensured, perhaps as a result of addition or removal of items from the container; multiple containers; state machines. A Manager is required to coordinate this complexity (figure 7).

The Manager, which has no specific abstraction, and no major local state, can ensure that concurrent access is controlled, and allow one to use a standard single-client version of the container class. As a finite state machine, it ensures that transient incomplete states are not

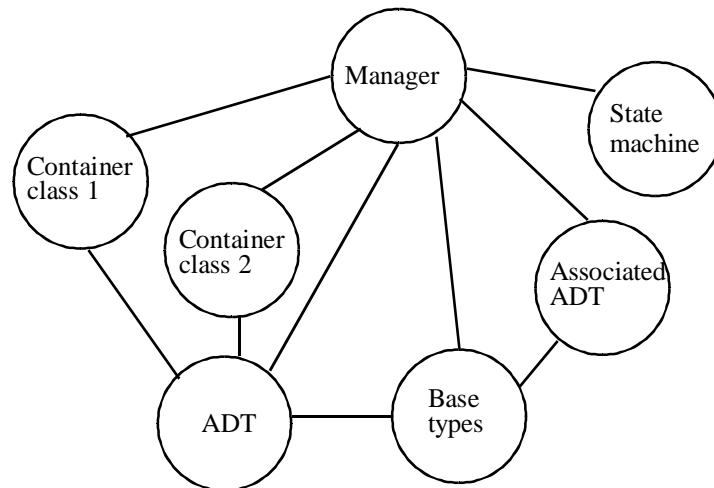


Figure 7 - the Manager

visible to clients. So, for example, if the generation of a new alarm must always be accompanied by the sounding of an audible annunciator, the manager ensures that it occurs. The manager also allows resistance to change. New hardware may allow the actual annunciator device state to be interrogated. When that occurs, the annunciator attribute associated with a particular alarm becomes redundant. By using a manager as the repository for this 'knowledge' of alarms, and annunciators, clients are unaffected when this change occurs.

It is also at this Manager level that any exception handling should occur, since the Manager can "interpret" the exception into a useful fault report. This limits the numbers of exception handlers required.

The concept of a Manager is very amenable to scaling. If a design starts off with a manager coordinating containers relating to separate classes, and then it is found that there is additional functionality (side effects, state machines) associated with these classes, then they can be hidden by another layer of managers. Note that this is not necessary if all one is doing is adding additional attributes to be recorded, modified, retrieved for an abstraction. Managers are needed to provide intelligence, to understand what needs to be done in addition to the simple change of an attribute. The question to be asked here is 'what is the added value?'. If a manager is not adding any additional meaning to what is going on in the container class or ADT, then it is not necessary.

Another advantage of the Manager structure is the ability to abstract away the 'real-time' behaviour of the system. In many large systems there is a substantial amount of off-line setting up of databases, which are then used online. For example, in a Sonar simulator, a complex mathematical model of the Ocean has to be run offline, and the results loaded to the online system for use during training. The offline and online systems must use the same ADT and possibly the same container class.

However, in the online system, there will be additional behaviour associated with these abstractions. If the ADT was maintained separately for both systems, there is a risk that they might get out of step, and the offline generated database would be incompatible with the online. If all offline facilities were available online, there is the danger that a future maintainer might accidentally build an application which used an illegal or inappropriate operation. However, by ensuring that all clients use the Manager interface, then safety is preserved, and the offline system can simply be built without the manager, to allow access to the full functionality to create the objects, but with no need to perform the online behaviour associated with the object.

c) Traffic Lights

Once a basic structural design is in place, the design has to start taking account of optimisation.

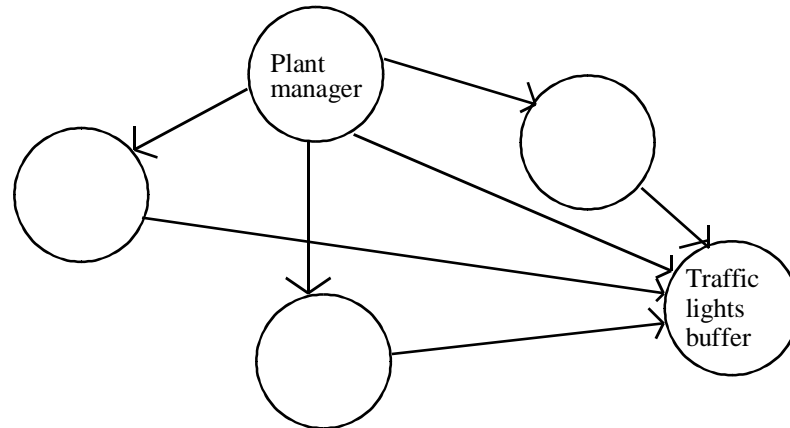


Figure 8 - the Traffic Lights Buffer

This is not searching for every last microsecond, which is not cost-effective on a large project, but performing optimisation in the large. Although the 'pure' design works in single entities, the hardware may work in particular aggregates, or it may be acceptable to run some things cyclically, if the loss of the last cycles information is acceptable on system failure. For example, the plant input system described earlier was optimised to handle a group of signals at a time, since that is the way the hardware provides them. Each signal may give rise an event, or generate a report, or may have to be passed on to another machine. If these are done individually, they would also slug performance. So the concept of a 'traffic lights' buffer is required (figure 8).

In order to ensure causality, all the actions resulting from a particular set of signals should be performed at the same granularity. If the report generation went on for each signal, while the events were buffered on some arbitrary basis, and the whole set of signals were not recorded until later, then the audit trail relating input and output actions could become fragmented. In addition, the report initiated by a signal might assume (legitimately) that the signal was already in the audit trail database when the report was being compiled, but in the above scenario this would not be true.

The traffic lights are 'set to red' while a group of signals are being processed. As a result of the process a number of different actions may be necessary, but instead of calling the appropriate application, the action waits at the traffic lights. When the group has been processed, the lights are 'set to green' and all the actions are performed, in the appropriate causal order, for the group. So in this case, the signals would all be recorded in the audit trail database before any events were registered or reports initiated. The release of this traffic is under the control of the manager processing the tranche of inputs, rather than the whim of the destination.

8.4 Patterns in the Small

Use of standardised Sets, Queues etc. is commonplace, but these are so low level that the cost saving is small. More useful is to generate a house style for the specifications and bodies of project specific abstractions. For example, on the plant monitoring system discussed earlier, safety considerations and the possible fragmentation of the heap by use of 'new' declarations meant that true dynamic creation/deletion of objects was prohibited. Thus a 'Managed ADT' was required, which was a managed storage set masquerading as an ADT. By providing a standardised version of this common requirement, including its documentation, specific Initialise, Create, Destroy operations, and example implementations, one could be confident that the whole project would follow the same design for this component.

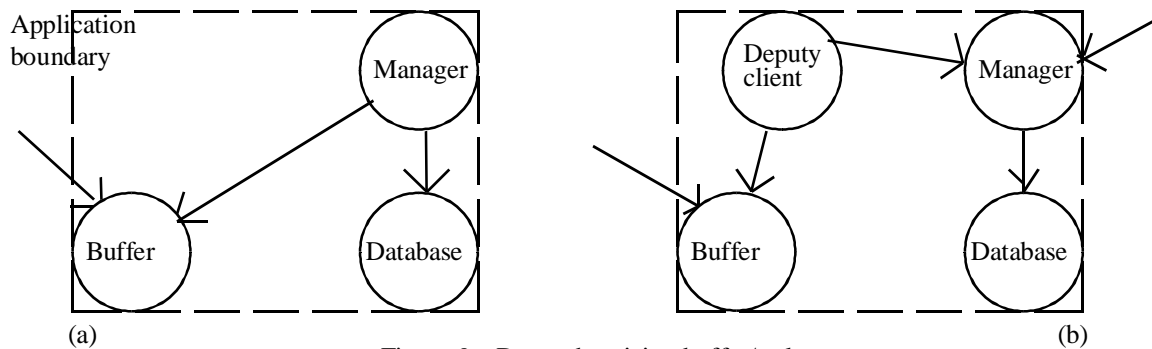


Figure 9 - Desynchronising buffer/task

Because of this prohibition on dynamic creation/deletion, another part of this system required a 'pseudo-dynamic' method of registering interest from various clients in Events. The events on which the resultant client action relied could be specified dynamically, but the client would not want to poll the Events database continually for anything of particular interest to it. The Events partition therefore exported a generic procedure which was instantiated by all potential clients.

Any partition could register itself as a client with Events on system initialisation and then register an interest in particular event ids. When an event changed state, the list of clients would be checked, and the generic procedure would action the procedure provided to it at instantiation as a formal parameter by each client. The advantage of this mechanism was that extra clients could be incorporated as the design evolved, and these clients could dynamically register and deregister interest in specific events without causing any dynamic space allocation/deallocation. This may be contrasted with the 'Phidippides' concept [3] which uses truly dynamic creation and deletion of tasks as messengers.

Other general requirements are the desynchronising buffer and desynchronising task. They are needed for clients whose main task is associated with interrupt generating hardware, for example the serial line devices mentioned earlier, or a system time facility which uses the clock interrupt, but 'kicks' applications to perform specific functionality at set times or intervals. Holding up these clients while a server performs some lengthy operation could cause them to miss their next interrupt.

The buffer concept is an obvious solution here, but the important point is to place it within the domain boundary of the server application (figure 9a). In that way, any modification which might alter the way in which the server manager interacts with it has no impact outside the scope of the application. If the buffer were added as a separate design construct living somewhere between the clients and the server, then there would be overheads in documentation and retesting.

If an application has some operations which can be buffered, and some which have to be performed in the thread of the client, then the latter have to go directly through the Manager, which, if it is providing concurrent access control, or atomicity of combinations of operations, will have a central task accepting their rendezvous. A single task cannot selectively accept client calls and take any buffered calls from the desynchronising buffer without polling or busy waiting. This leads to the need for a desynchronising task, acting as a 'deputy client' (figure 9b). By preparing and disseminating standard versions of these constructs, the same structure could be reused easily.

9. Search for Misuse

In general, if one has an application area deep within a system, one would expect to see the following:

- 1 package for base types relating to attributes (the equivalent of 'Standard' for this application)
- A few true classes represented by ADT packages
- 1->a few container classes, the ASMs
- 0->a few utility packages, combining operations for the container classes
- 0->very few straight functions or side effects or state machines
- 1 Manager if >1 container class or side effects present.

In the above 'a few' means 2-3, but this is not a rule, just a measure for deciding whether the design needs looking at more closely to check its behaviour. For a transaction centre (as in the MMI or script case), expect to see a lot more utilities, but all with the same relationship to the central puppet master. If a partition has some critical response requirement, and a very limited number of clients, then one might expect to see utility or manager packages with operations "tuned" specifically to those clients. This is where pragmatism has to apply. A pure client-server architecture has servers who know nothing of their clients. But reality is never pure, and such compromises can improve performance out of proportion to their undesirable aspects.

If one sees more than one manager, then one worries about how atomicity of operations is being ensured, and concurrent access is being controlled. If there are a large number of classes exported by a package (rather than simple types exported by the 'Standard' package) then is there some confusion or amalgamation going on? A package may legitimately export more than one class if they are essentially different views of some underlying connecting abstraction, but would not expect to see that situation very often, and then only with a very few classes in a single package.

One must also look at how the behaviour will impact the clients. If an operation is exported which takes some time to perform, then can a client afford to be held up while it occurs? It is often forgotten that if a client cannot afford to be held up, then it is impacted by other clients who can, and which are waiting while some lengthy operations is performed within their thread of control. This is where desynchronising buffers or amalgamation of utility operations into the container classes for efficiency may be required.

10. Conclusion

The suggestions given above are not rules - they are simply ideas as to where to look for patterns in problems and solutions. The main purpose is to give some guidelines for improving communications and consistency, producing reasonable designs and recognising in time when the design is going wrong.

Acknowledgements

The author would like to thank Dr.R.J.Cant of Nottingham Trent University, Dr. P.R.Rees & D.Farquharson of Ferranti Industrial Systems, and T.Stuttard of Ferranti Simulation & Training for useful comments and suggestions.

References

- [1] P.Coad, E.Yourdon, Object Oriented Design, Yourdon Press, 1991.
- [2]S.Shlaer, S.J.Mellor, An Object-Oriented Approach to Domain Analysis, Software Engineering Notes, ACM Press, Vol. 14, No. 1, July 1989.
- [3] A.Burns, Concurrent programming in Ada, Cambridge University Press, 1985