CHAPTER 4 SYSTEM ARCHITECTURE

LEARNING OUTCOMES

By the end of this chapter, you should be able to:

1. Explore dealing with the system decomposition, how to divide a large system into a number of manageable subsystems.
2. Discuss how to define the interfaces between them and how to develop them incrementally.

INTRODUCTION

System architecture has two parts: hardware architecture and software architecture. If the systems are addressed as a whole, these architectures are developed together – hardware and software are codesigned. The hardware/software codesign is a complicated matter which involves integration of different modelling techniques.

Software and hardware designs are often separated, and hardware design precedes software design. The use of standard hardware components or the use of hardware from earlier versions of the same product are often the reasons for this. Even if hardware is specially designed for the system, its design is often controlled by non-functional arguments like the cost or power consumption of the components. Software design affects the hardware structure only if it turns out to be unfeasible to implement the desired functionality on the given hardware. Most of the time, the hardware structure and its peculiarities are treated as additional constraints for software.

However, external requirements and hardware requirements have to be separated. Otherwise, the structure of the hardware would drive software design, and the design would become very sensitive to changes in the hardware. To reverse the order, the OCTOPUS method isolates the hardware behind a software layer called the *hardware wrapper*. Parts of the analysis and design of the hardware wrapper can be postponed until the requirements placed on it by application software are known. The hardware wrapper handles the hardware and provides services to the application subsystems (refer *Figure 4.1*). This is related in *Chapter 5* and *6*.

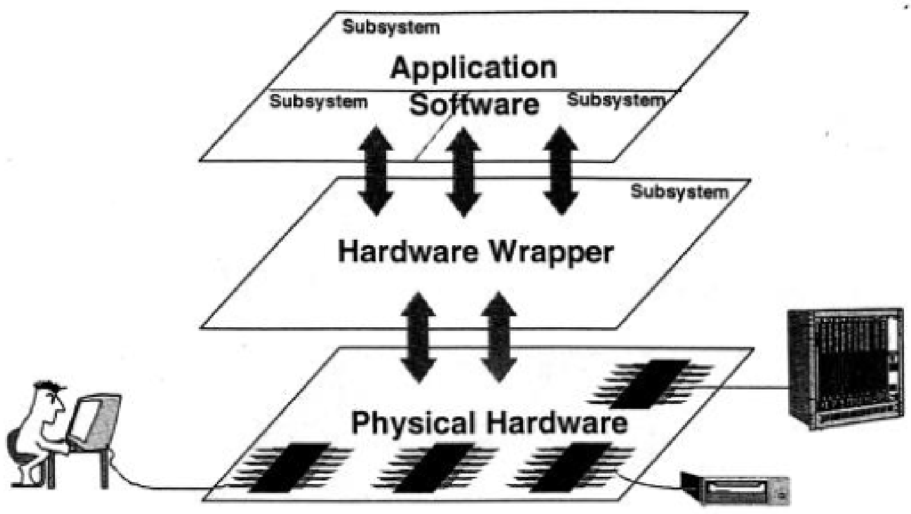

Figure 4.1 : The hardware wrapper interfaces to the hardware and application subsystem

Embedded systems are often developed as a family of products. The products in the family may differ for unavoidable reasons, such as differences in communication standards, supply voltages or safety regulations. Often the differences are used to broaden the customer base by varying the functionality, capacity, outlook and price of the products. Despite these differences, it is essential to be able to develop the family as a whole. The inability to do so will radically increase the cost of development. This causes variance in the software controlling the product. If the product is developed as a family, software architecture on the patterns of system organization which describe grow functionality is partitioned and the parts are interconnected [Shaw '90].

Each product in the family has to satisfy its functional and non-functional requirements. The software must be efficient enough in terms of time and space; it must be as responsive and as reliable as required. Often, the non-functional and some of the functional requirements are similar throughout the whole family of products, and the architecture of the family must support these requirements. The architecture should also possess qualities related to the software development process. It should be easy to extend and modify the architecture to develop and test new products based on it. Qualities like extendibility, portability, or testability are determined by the architecture. *Figure 4.2* shows the focus of this chapter within the software development cycle.

4.1 MODULAR STRUCTURE

Software architecture is traditionally understood as the modular decomposition of the system. Such a definition for software architecture has FOUR (4) parts as shown in *Figure 4.2*:

- ❖ Modular structure of the system
- ❖ Module interfaces
- ❖ Communication mechanisms
- ❖ General control strategy

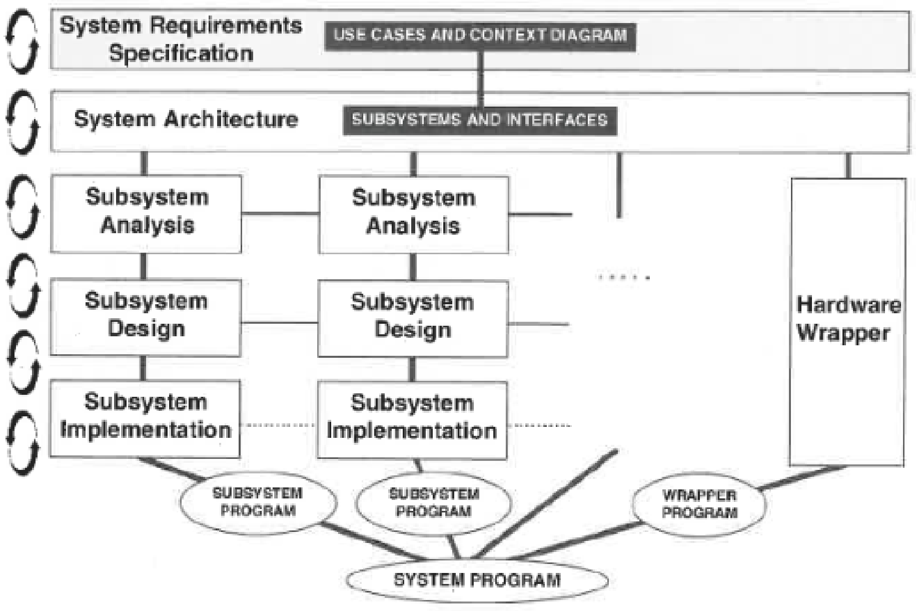

Figure 4.2 : The system architecture phase

This top-down view of software architecture is very useful because it supports early separation of concerns. If modules are weakly coupled, they can be developed and tested almost independently. However, this definition of software architecture is restricted to high-level design. In reality, architectural decisions are made at all levels of detail, and often the fine-grained decisions determine the qualities of the system. Consider, for example, portability. The decisions that undermine portability include use of nonstandard language features, special operating system features and special hardware services. To achieve portability, these dependencies must be encapsulated behind a generic interface. There must also be a known policy to reduce dependencies, and it must be strictly followed. Since a very small detail may undermine portability, the policy is an essential part of a software architecture supporting high portability.

OCTOPUS contains a set of policies, and if these are followed as such, the resulting architecture will be as below.

- Base modularition on concepts of the domain
- Use objects as a mechanism for structuring the software
- Use explicit concurrency in terms of light-weight processes
- Base the process structure on the time requirements of the external events
- Base control mechanism between processes on messages
- Base data sharing on the use of single address space
- Separate the hardware layer from the application

This architecture is applicable to many embedded systems. It provides efficiency and responsiveness, and it supports several decoupling mechanisms. Another architecture can be achieved if other policies are defined and followed. Note, however, that the further the architecture drifts from the one described above, the harder it becomes to use the OCTOPUS method.

Since architecture determines the decomposition, interaction and overall control policies, it can be supported on the design and implementation levels by capturing the interaction and control mechanisms in a *framework*. A framework is a generic implementation which can be specified by adding the product-specific details.

4.2 EARLY DIVISION INTO SUBSYSTEMS

In a large system, it is desirable to share analysis work among several teams. This can be achieved by dividing the system into smaller subsystems, then analyzing and developing these subsystems independently. Early division involves risk. System-level requirements are not well defined before the division, and it may be difficult to satisfy them using the chosen subsystem structure. Early division also increases the work. Subsystems do not partition the conceptual space of requirements. The same requirements will be analyzed several times. Common concepts in different subsystems will result in multiple classes having similar names but different definitions, which is a problem for further evolution of the system. Also, the subsystem interfaces will evolve as understanding of the requirements increases.

If the analysis work has to be divided, a good way to ensure reasonable independence between subsystems is to use different domains as different subsystems. A domain is "separate real, hypothetical or abstract world inhabited by a distinct set of objects that behave according to the rules and policies characteristic of the domain."

Since the concepts in a domain are defined within that domain and make sense only in that domain, domain partition the concept space. Moreover, since the domains are independent, subsystem division based on domains results in subsystems that can run independently and concurrently. They may potentially be distributed.

As an example of partitioning a system into subsystems based on domains, consider a system controlling a chemical process. You can immediately recognize the following domains :

- Chemical domain : model of the process to be controlled
- Device domain : sensors used to get feedback and actuators used to affect the process
- Operating system domain : concurrent processes, event handling and communication between processes
- Database domain : data storage and queries
- User interface domain : windows, menus, icons and dialogs boxes

These domains can be analyzed separately and implemented as separate subsystems. However, since the domains are distinct, it is possible to provide separate support for them. Actually, all the domains listed above are general enough to be interesting in many applications, thus, operating systems, databases, user interface frameworks, device drivers and even chemical process modelling frameworks are available. The task for developing a system for such a process control can be seen as an integration task. Note also that the independence of domains promotes layered architecture (refer *Fig. 4.3*).

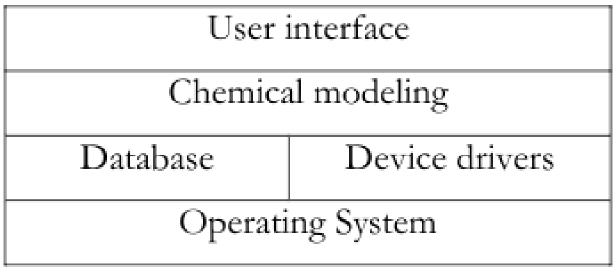

Figure 4.3 : Layered architecture for control of chemical processes

In the process of searching for different domains it is useful to consider services used by the application. If these services involve concepts which are not part of the apllicatopn. They indicate the existence of a separate service domain. Most of the domains in the chemical process example are service domains.

Large domains are rather independent. They are routinely implemented as separate subsystems and have standard interfaces to other subsystems. In a large and complex system, it is also necessary to identify smaller domains like fault-tolerant computing, performance monitoring or alarm handling. These domains depend on the structure of other domains and will not be as independent as large domains. Since these domains are often not implemented as separate subsystems, they have no standard interfaces.

As an example, consider fault-tolerant computing. Embedded systems are often required to continue operation despite problems in their environment and event to sustain hardware failures to some extent. However, the domain of fault-tolerant computing is seldom separated. Instead, the recognition of the fault and the recovery actions are designed and implemented as parts of the application. Separation of this domain would ease the development of the application and wold guarantee uniform behaviour, but it is hard to provide widely acceptable support for fault-tolerant computing.

Other possibility is to provide fault-tolerant processes on a multiprocessor network through replication of computing units and secure communication. Each fault-tolerant process runs on two different processor cards, and the replicas are kept in "hot standby" by also routing all the messags to the replicas. Once a computer unit fails, the replicas take over the role of the origina processes on the failing unit. New replicas are initialized for them and alarms signaled.

The interface of this fault-tolerant subsystem consists of a simple registration procedure, but its structure is very much related to the structure of the overall control mechanism used in the system. No wonder fault-tolerance services are typically implemented as an addition to an operating system and not sold separately.

Once a domain has been identified, it can be tested by questioning its independence of other domains. If the domain is independent, it should be possible to imagine several different implementations for it, and it should be possible to sell a package providing its services.

Before the analysis of domains is continued, the system decomposition into subsystems based on these domains should be considered. The control should be defined and the interfaces between the domains designed. The system can be shown as a set of subsystems connected by client/server relationships (refer Figure 4.4). The arrows point to the server domains.
In practice, division into domains is not sufficient. Domains tend to be large and often commercially available solutions exist for them. In our example, failure to reuse existing solutions for database, operating system and user interface parts would radically increase the cost of the system. However, the chemical modelling and device control parts are still large problems, and further subdivision is needed. Further division can be based on finding functionality clusters. The clusters of related functions are not as separate as domains, but will still give some division of work.

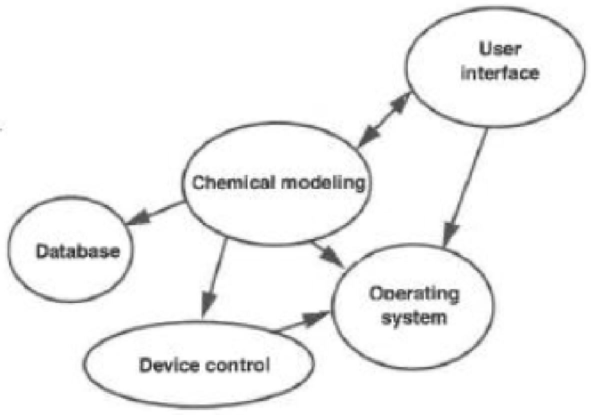

Figure 4.4 : The relationships between the chemical plant process control system

4.3 SUBSYSTEMS DIAGRAM

The object model notation can also be used to describe the decomposition into subsystems. As shown in *Figure 4.5*, OCTOPUS separates the hardware wrapper from the other subsystems. Client/server relations between subsystems can be shown as associations.

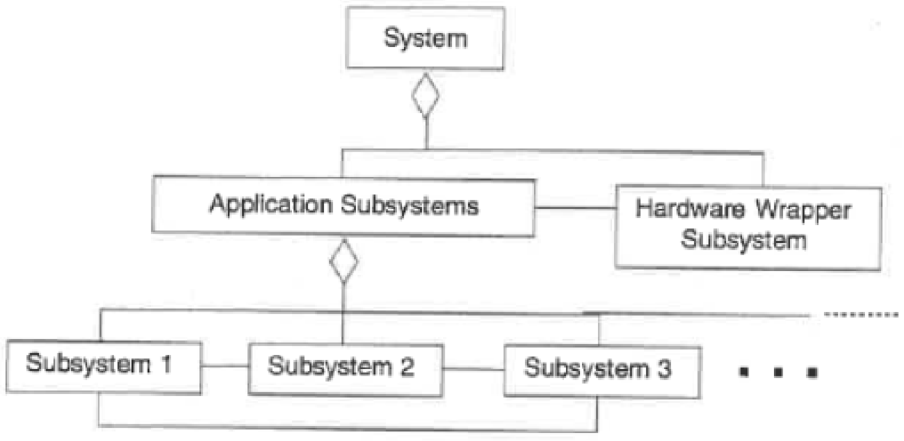

Figure 4.5 : Subsystems diagram

4.4 INCREMENTAL DEVELOPMENT

Decomposition into subsystem may reduce the time needed to complete a software project because subsystems can be developed in parallel. It also has other positive organizational and managament effects: the distribution of tasks to the project members becomes easier and more flexible. It is recommended that each subsystem be allocated to at least two designers [Sixtensson '93].

Decomposition into subsystems promotes reuse at a higher and more useful level than the class level. A whole, well-designed subsystem can be reused in another system. For example, a subsystem for remote downloading of files in a telecommunication network can be designed in such a way that the dependencies on a specific telecommunication protocol and the underlying hardware are implemented in their own subsystems. In this case, this download subsystem can be configured for use in different products and according to different telecommunication protocols.

Decomposition into subsystems is not without cost. It necessitates the specification of the interfaces between the subsystems. Therefore, having subsystems that are highly dependent on each other may not be justified. OCTOPUS recommends analyzing the subsystems in parallel before designing and implementing any of them in order to clearly specify the interfaces between them and to improve the original decomposition into subsystems. The responsibilities of some subsystems might change, two subsystems that are highly dependent in each other might merge, or new subsystems may be defined.

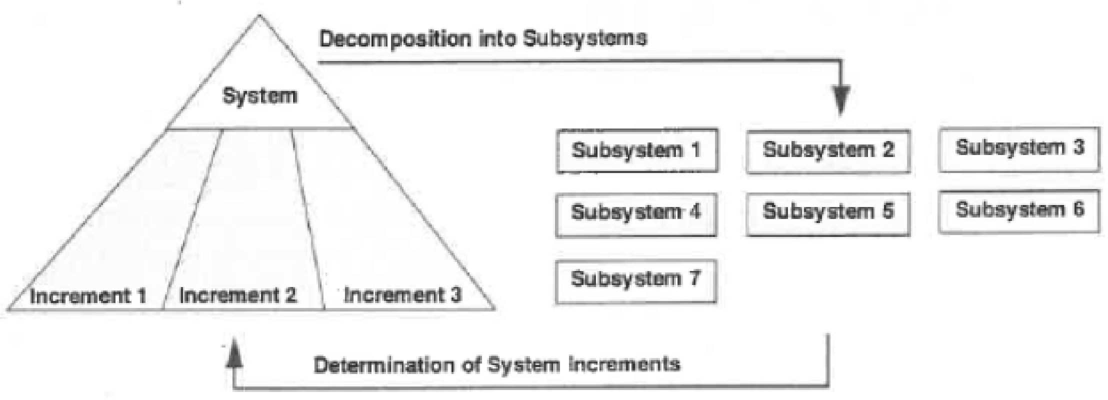

Figure 4.6 : Subsystems and system increments

A large software system must be developed incrementally. A system increment is composed of one or more subsystems which, when fully implemented, produces a partially operational system that can be tested [Sixtensson '93]. In *Figure 4.6*, for example, subsystems 1, 4 and 5 make up one system incrememt, and subsystems 2 and 3 another. A system can become usable even if all the increments are not implemented. Naturally, the increment is implemented in an order that matches their relative importance. The core increment is implemented first. It is the one on which other increments depend, and without it the system cannot run. An increment can be added to the previously implemented increments until a fully operational system is reached.

4.5 INTERFACES

The description of the interfaces should be tuned according to the control mechanism. Typical architecture for embedded systems is based on reactiveness: the control is often event-driven, and operating system messages are used to implement the events. Try to select the mechanism so that the message encoding and decoding does not eat up the performance.

The subsystem interfaces are defined in the analysis. The functional model describes the services, and the dynamic model shows how the subsystems interact. There is also a relationship between the subsystem interface and its object model. The interface can be seen as a collection of services provided by the objects in the environment of the subsystem (refer *Figure 4.7*). Relationships to other subsystems are visible in the object model as relationships to the objects in the environment. This makes it possible to show the nature of the relationshop between subsystems and, later, the exact interface needed by this relationship.

Since subsystem interfaces are described as a result of analysis, but subsystem division is determined before analysis, there is a need to iterate between architectural design and analysis. In particular, the groups analyzing subsystems in parallel which are known to be associated should coordinate their work.

The correctness of subsystem division can be checked by taking each use case of the system and drawing a message sequence chart showing how the subsystems communicate. It should be easy to understand the responsibilities of each subsystem.

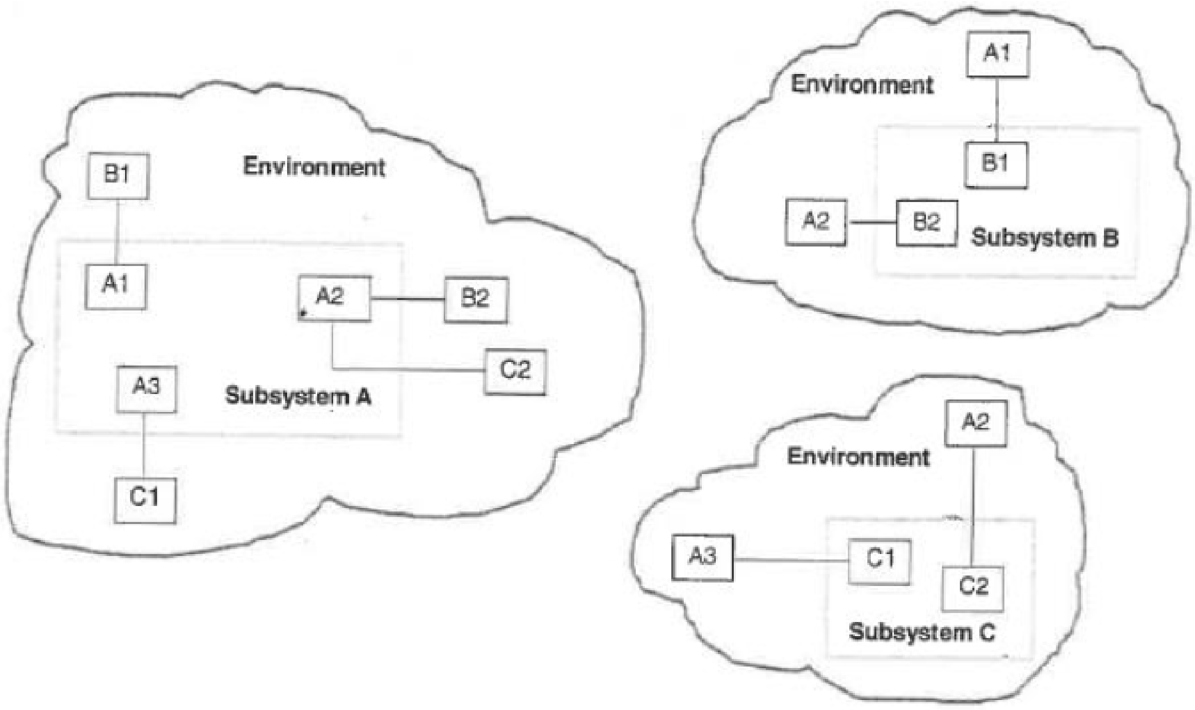

Figure 4.7 : Subsystem interfaces as objects in ther environments