A GENERAL SOFTWARE ARCHITECTURE FOR INFORMATION SYSTEMS

Kristine Stougaard Thomsen
Computer Science Department
Aarhus University, Denmark,
on leave from
Mentor Informatik A/S
Fredens Torv 6, DK-8000 Aarhus C

July 1992

Abstract

A general software architecture for large and medium size information systems is presented. The architecture is a multi-level client-server architecture, where all dependencies on hardware and software platform are encapsulated into modules. The architecture provides a skeleton around which to grow an information system and is independent of the actual data model, functionality and user interface of the system. It supports division of labour during development, and it supports maintenance, enhancement and portability of the resulting system. In all these respects, the architecture is superior to the architecture obtained when using fourth generation languages.

The architecture has been successfully used in several large projects during the last 6 years and practical experience shows that it provides a reusable starting point for future development projects.

1 Introduction

In this paper, an *information system* is considered to have the following characteristics. It is structured around a *data model* a database, that models a part of reality that is relevant for some administrative purpose of the use organization. The users of the system update and query the data model as an integrated part of their daily work routines. The different ways of manipulating and querying the data model constitute the *functionality* of the system, for instance in the form of screen and report applications. The users interact with the system through the *user interface* which provides the "look and feel" of the system. Some of the manipulations and queries to the database may be complex, but the main source of complexity is the size and complexity of the data model, and the large number of relatively homogenous applications and their integration with the work routines of the users.

During the last 6 years I have been working in a Danish consulting company, Mentor Informatics Ltd., where we have developed a general system architecture for information systems. The architecture has been successfully used in several large projects. The two most important projects were the development of an educational administration system (ESAS) and a financial system (ØSE) for the Danish trade schools. There are 115 such schools with a total of approximately 2000 users. The initial project effort of the ESAS project was 220 person-months, and at present, after 5 years of use, maintenance and enhancement, the ESAS system consists of 239 screens and 143 reports. The initial project effort of the ØSE project was 115 person-months, and the ØSE system now consists of 105 screens, 45 reports and (approximately) 30 batchjobs. The platform and tools used in both projects are mini-computers running UNIX, C and ORACLE. The ESAS and ØSE projects will be used as examples throughout the paper.

The paper is organized as follows: Section 2 discusses requirements for a sound software architecture for large and medium size information systems; Section 3 describes the general architecture that we use, and Section 4 gives a brief introduction to the production environment that we have established to support development of systems with this architecture; In Section 5, the advantages of the architecture are discussed in relation to the requirements in Section 2. Moreover, the architecture is compared to the architecture obtained when using 4'th generation languages.

2 Requirements for an Architecture

Most of the requirements for a sound software architecture for large and medium size information systems discussed in this section apply to computer systems in general, but the actual architecture suggested in Section 3 on the basis of the requirements is particularly suitable for information systems.

Support Maintenance and Enhancement

An information system is used as an integrated part of the daily work in an organization. An organization is dynamic and changes in an interplay with its environment. Changes in the organization result in changes in requirements for the information system. The lifetime of the system therefore depends on the ease of maintaining and enhancing the system to meet changed requirements, which is closely related to the modular structure of the system's architecture.

Parnas (1972) has described the principles of modular software architecture and later in more detail in (Parnas et. al., 1985). The basic idea is that each design decision that may change independently of others should be encapsulated into a module. The module hides the design decision and reveals as little as possible about the secret in its interface to other modules. Quoting (Pamas et. al. 1985, p. 260) the following goals of such a modularization are important:

a) Each module's structure should be simple enough to be understoodfully.

b) It should be possible to change the implementation of one module without knowledge of the implementation of other modules and without affecting the behaviour of other modules.

c) The case of making a change in the design should bear a reasonable relationship to the likelihood of the change being needed (...)

d) It should be possible to make a major software change as a set of independent changes to individual modules, i.e., except for interface changes, programmers changing the individual modules should not need to communicate. (...)

e) A software engineer should be able to understand the responsibility of a module without understanding the module's internal design.

(...)

The recommendations of Pamas et. al. are based on the "Divide and Conquer" principle, which has been successfully applied in algorithm design. Parnas et. al. applies this principle to the larger scale of software engineering and combines it with the idea (c) of taking into consideration the likelihood of change. This has been discussed in more detail by Jackson (1983) and Coad & Yordon (1991) as one of the principles behind object oriented design. A system's overall structure should be based on relatively stable properties and concepts of the application domain together with relatively stable design decisions. Properties and design decisions that are more likely to change should not be the basic structure of the system but rather be encapsulated into modules that can be easily changed.

Support Division of Labour

The Divide and Conquer principle supports division of labour, and several of Parnas's goals are directly concerned with enabling a team of system developers to share the development task among them without too much overhead.

If a task would take K months to complete for 1 person, the ideal would be, that the task could be partitioned in such a way, that x people could complete the task in K/x months. As argued in (Brooks, 1975) this is rarely the case in practice. When a task is divided, several workers need training, and coordination between subtasks is needed. The amount of time spent on training grows linearly with the number of workers, whereas the time spent on coordination tends to grow polynomially (or even worse) with the number of people involved. Brooks describes this phenomenon in order to argue that "adding manpower to a late software project makes it later".

Brooks also establishes that some tasks have a sequential nature that makes them unpartitionable. Most tasks involved in systems development have some degree of sequential nature, e.g. design must take place before programming, programming before test, and test before installation. That is, even with extensive modularization, we can only obtain a limited partition of the original task. There will be an inherent limit to the number of people that can meaningfully be assigned to participate in performing the task. This limit is determined partly by the degree of sequential nature of the task and partly by the quality and fine-grainedness of the modularization.

Support Portability

Portability can be considered a special case of maintenance where parts of the hardware or software platform on which the system is build, is replaced. A software architecture that supports portability can be obtained, using Parnas's principles, by encapsulating the design decisions conserning choice of hardware and software platform into modules thet prevent these decisions from being used by other modules.

Portability in a broader sence also includes the possiblity of using different hardware configurations than the one anticipated when the system is designed. At present, we see a trend towards distributed configurations connected in networks. This indicates that changes from centralized to distributed configurations are likely to occur. According to Parnas's principle (c), this change should therefore be easy to implement if a central solution is chosen initially. It should be easy to change the system to run on various different distributed configurations, including heterogenous configurations of different cumputers.

Support Growing instead of Building Software Systems

Brooks (1987) distinguishes between building and growing a large software system. He expresses the opinion that the most promising contribution to improvements in software development during recent years comes from the field of prototyping. For a discussion of prototyping, see (Floyd, 1984) and (Floyd, 1987). Growing a software system means incrementally adding more and more functionality to the system while constantly being able to test and evaluate the system according to user needs. As Brooks points out, growing a system in this way requires top-down design. Some fundamental principles for structuring the whole system must be decided initially allowing flesh to be added to the bones of the overall architectural skeleton as development proceeds.

We would like an architecture that can act as such a skeleton for a system that is to be grown in cooperation with users. That is, the basic structuring of the system should be independent of the actual data model, functionality and user interface of the system.

3 A General Software Architecture

In this section, the general software architecture is presented as if the whole system is running on a central machine using a single database system and a single set of user interface tools. This is the way the architecture has primarily been used till now in our projects, but among the major advantages of the architecture is its openness to distribution on heterogenous configurations, as discussed in Section 5. Now, however, we look at the centralized version, and terms like "server process" and "distribution of operations" should not be confused with the similar terminology within the field of distributed computing.

The architecture is a client-server architecture which makes a division of the system into subsystems and an orthogonal division into layers. The layers span the gap between the presentation of the system to the user, and the underlying database system. The architecture is shown in figure 1, and the different components of the figure are described in detail in the following subsections. Figure 1 is a runtime picture of the system, showing the running processes and how they communicate. There is not a one to one correspondance between processes in the running system and programs or modules of the system. How processes relate to program modules will be made clear in the description of the different layers.

3.1 System Layers

The horizontal layers shown in figure 1 are the database system layer, the logical database layer, the server layer, the application layer, the presentation layer, and the layer consisting of the interface tool available on the terminal, PC or work station from which the user interacts with the system.

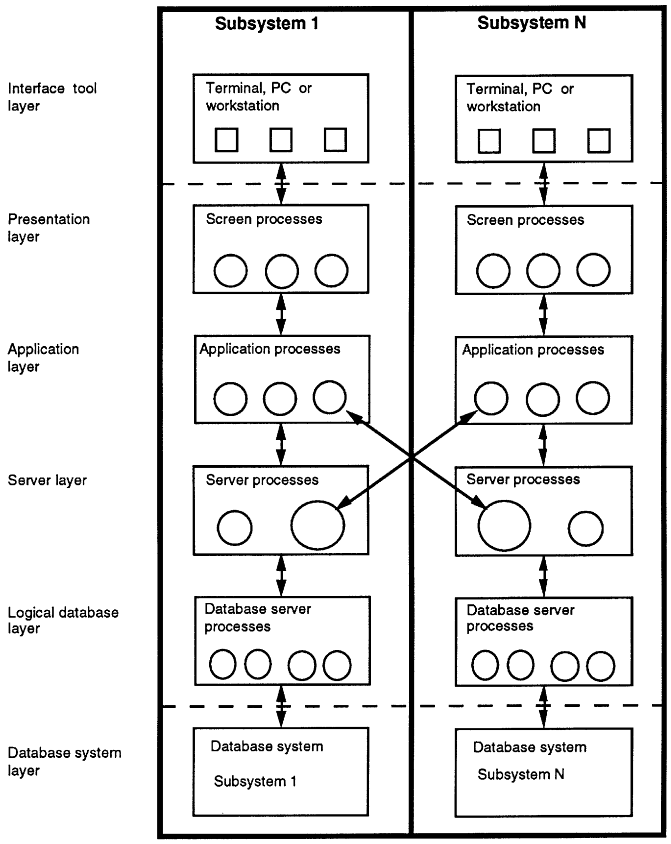

Figure 1: **A General Software Architecture**

The top and bottom layers consist of basic software that constitutes the environment for our architecture, while the intermediate four layers constitute the architecture of the system. Each of the four intermediate layers consists of a number of separate processes. A process acts as a server that waits to receive requests from processes at higher levels. In the processing of such a request, the process can act as a client by sending requests to processes at lower levels, asking for services provided by this layer. A general communication module linked on all processes encapsulates the design decision of how communication is implemented between all intermediite layers in the architecture.

The Interface Tool Layer Provides Means For User Interaction

The interface tool layer is part of the environment in which the system is built. It provides tools for programming the interaction between user, display, keyboard etc.

The Presentation Layer Encapsulates the Interface Tools

The presentation layer consists of a number of screen processes, one for each screen application that is run by a user at the given moment. Thus, the number of processes in this laver varies with the number of current users and their activities.

The presentation layer implements the *general functionality* of the user interface and the *specific layout* of screens, regardless of their underlying functionality. In a system with a character oriented user interface, the screen process handles the sequence of fields to be filled by the user or shown to the user. Moreover, activation of function keys are caught and leads to activation of different functions, some of which may be activated by sending messages to the application layer.

Only one general screen program is involved in implementing the presentation layer. Besides, there is a set of datafiles with screen layout descriptions, one for each screen. The general screen program is a module that encapsulates two important design decisions that may vary independently of the rest of the system: the choice of basic interface tool on which the system is built and the general principles of the user interface. The different screen processes in the presentation layer shown in figure 1 are all instantiations of the same general screen program using different screen descriptions for the actual screen.

The Application Layer Encapsulates Specific Functionality

The *specific functionality* of each screen application is implemented in the application layer. Each screen process has an associated application process, that determines what is the effect of manipulating the user interface given in the screen process. Besides, report applications and batch jobs run as application processes with no associated screen process.

There is an individual program for each application. It knows about which data in the database are relevant for the application, on what basis they are fetched or updated, and what computations are to be performed on them before or after the presentation of them in case of a screen.

The Server Layer Provides Interfaces between Subsystems

The server layer consists of a fixed number of server processes. It is at this level interfaces between subsystems are defined. One (or more) of the servers provide the high level operations by means of which applications in other subsystems can interact with the database owned by the subsystem at hand. In figure 1, the interface server processes are shown as larger circles in the server layer, and it is shown by means of arrows, that they allow for communication between subsystems.

Moreover, the server layer provides higher level operations on top of the logical database layer, including bundling of database operations and consistency checks.

There is an individual program for each server.

The Logical Database Layer Encapsulates the Underlying Database System

The logical database layer consists of a number of database server processes. The services provided by these processes are logical operations on the underlying database, such as retrieval and update operations of varying complexity.

Each operation is characterized by an interface, consisting an operation code and a format of a communication record. The implementation of the operation in terms of operations on the underlying database system is hidden within the logical database layer and is of no concern to the client processes.

The logical database layer is implemented by a single program, the database server program, from which all database server processes are instantiated, and a data file, called the dbt-file, that contains the implementation of all logical operations. The logical database layer encapsulates two important design decisions: the choice of underlying database system and the representation of the data model implemented in the database, that is the layout of tables and fields etc.

In principle, all database server processes could support the same logical database operations, but a table describes how the services are distributed among them. More details on this are given in Section 4.1.

The Database System Layer Provides Traditional Database Facilities

The database system layer is part of the environment in which the system is built. We assume that it provides traditional database facilities such as transactions, concurrency control, error recovery etc. Similarly, the question of distribution of the database, replication etc. is considered to be part of this layer and not the architecture of the system.

The Flow of Requests Through the Layers

As an example of the flow of requests through the layers in the runtime architecture, we consider a user interacting with a screen in the financial system. The screen allows the user to check the balance of an account.

The typing of the account number takes place as an interaction between the user and the screen process of the screen in question. When the user activates a specific function key, the screen process sends a message to the application process, which knows what to do about it. In this case, the application process knows that an account has to be found, and its balance computed. Therefore, the application process sends a request to a server process, in this case an account server process in the server layer. As a consequence of this request, the account server process sends a series of requests to some database server process to get the account with the typed account number and all the postings that have been registered on this account. As the database server process at the logical database level responds to these requests, the account server process sums up the amounts of the postings, sends the result back to the application process, which in turn sends values to the screen process, which presents them to the user.

The account server process mentioned is in fact an interface server process in one of the subsystems of the ØSE system, and the operation of finding the balance of an account is an operation that is accessible to other subsystems in the financial system.

The application process may send requests directly to database server processes instead of going through the server layer. This is relevant when only low-level logical operations on the database are needed by the application.

3.2 Subsystems

As shown in figure 1, the system is divided into vertical subsystems. Each subsystem can be considered a high level module that encapsulates all design decisions of the subsystem making only the interface routines in the server layer visible to other subsystems.

The Division into Subsystems is Based on User Work Practice

The division into subsystems is performed at an early stage of the development process. There is no well defined method for performing this division. However, we have found it useful to base the division on criteria given by the work practice of the users. That means defining a subsystem as a meaningful unit seen from a user's perspective, containing functionality related to a logical set of tasks in the work situation.

As an example of subsystem division, the system for educational administration was divided into 10 subsystems: Student administration, schema and absence registration, examiniation planning, community home administration, personnel administration, building and rooms administration, material, equipment and furniture administration, activity planning, data exchange (with other schools) and system administration.

Such a division eases incremental implementation (growing) of the system in the use organization, e.g. by implementing one subsystem at a time.

Of course it is also important to minimize the dependencies between different subsystems, since explicit interfaces must be designed between subsystems. These interfaces should be kept as simple as possible.

A Subsystem is Responsible for its own Database

Each subsystem has its own database, at least logically. Physically, they may be just different subsets of tables in the same database system, or they may be based on different database systems, maybe even on different machines as discussed in Section 5. Responsibility for maintaining the database and ensuring its consistency is localized to the individual subsystems. Moreover, responsibility is localized to the two levels called the logical database level and the server level.

Besides, each subsystem consists of a number of different applications, that help the user in performing some tasks that involve querying or manipulating the database. Each application in the system, e.g. each screen and each report etc., belongs to a specific subsystem.

Interfaces between Subsystems are Localized to the Server Layer

The subsystems interact with each other through the interfaces defined by the interface server processes in the server layer, as mentioned in the previous section. The operations provided by the set of interface server processes of a subsystem, are the only means by which other subsystems can access the database of the subsystem.

4 A Production Environment

Before a system can be developed using this architecture, a production environment must be established. To give an impression of the extent of this, our present production environment is briefly described.

Our production environment provides standard modules that form the skeleton around which to grow the information system together with a set of tools for building components of the system. Moreover, a set of guidelines and standards are maintained that can lead programmers through the tasks of developing screens, reports, server processes, interface operations etc. Both standard modules, tools and guidelines are expected to be tailored to the needs of each specific software development project.

The standard modules in our production environment are primarily the communication module, the general screen program and the database server program. They can be reused in different software development projects, since they are totally independent of the actual functionality of the system to be developed. However, they must of course exist in a version that matches the platform on which the system is to be developed.

The tools and guidelines in our production environment take advantage of the characteristics that an information system consists of a large number of relatively similar system components, i.e. screens, reports and background jobs. The tools and guidelines support mass production of such components but are intended to be flexible enough to enable easy incorporation of new ideas that come up when growing the system, both regarding user interface and functionality.

4.1 Tools and Standard System Modules

The Communication Module

The entire system architecture, shown in figure 1, is based on communicating processes organized in a multi-level client-server configuration. A communication module that provides a high level way of sending requests and responses between processes at all layers is therefore a fundamental system module provided by our production environment. Each process in the system has this communication module linked with it.

The communication module provides a procedure interface that allows a client process to send a communication record containing a code that uniquely identifies a specific service and the actual parameters of the operation that implements the service. The client need not know the identy of the server that supports the service. In a global table, information is maintained about which processes in the system supports which services, and the communication module automatically forwards a request to me server process that supports it. The table is initialized on the basis of a text file that can be edited on location followed by re-initilization. This is particularly relevant when detaining how logical database operations are to be distributed among the database server processes.

The communication module also includes mechanisms for receiving a response or a sequence of responses as a result of a request to a server, and mechanisms for exclusive communication between two processes for a period of time without interruption from other processes.

At present we have a UNIX version of the communication module, and a VMS version is forthcoming. Furthermore, we have a prototype version of a communication module that supports distribution as discussed in Section 5.

The Database Tools

Our production environment provides a standard program that implements the database server processes. The database server program will typically need adjustment to each project, for instance to implement specific security requirements. The database server program is compiled with no knowledge of the operations that it is to support. The actual operations, together with the size and format of the communication records associated with them, are described on a separate file, called the *dbt-file* that is read as data by the running database server processes. A database server process indexes into the dbt-file when receiving a request with some operation code from a client. That is, format and size of communication records are not known at compile time. This makes the process of generating new logical database operations quite fast, since it does not involve recompilation of the database server program.

We have tools that support the process of creating new tables in the database and creating logical operations. One tool automatically creates a set of logical standard operations for a table and translates them into the format of the dbt-file, readable by the database server program. Another tool reads an SQL-expression from a text file and generates a new logical operation on the dbt-file.

At present we have two versions of the database server program that support the relational databases ORACLE and INGRES, respectively.

The Screen Tools

Our production environment provides a general screen program. The program is tailored to each individual project and implements the general functionality of the user interface. Moreover, we have screen tools that provide a high level interface to defining and manipulating the specific layout properties of the individual screens.

At present, our screen tools and our general screen program only exist in a version based on a textual user interface tool, but we are working on developing a version that uses the X Window System.

The Application Tools

The application tools consist of tools that support development of screen applications and tools that support development of reports and batchjobs. The screen application tools form a supplement to the screen tools in supporting the description of the individual functional properties of a screen application. The tools primarily consist of skeleton programs to be filled out with bodies for general procedures, which are activated as the result of requests from the screen process when certain events are caused by the user.

The tools that support report applications and batchjobs mainly consist of skeleton programs.

4.2 Standards and Guidelines

Besides the different tools and standard modules that support the system architecture, each project develops and maintains a set of standards and guidelines that are important for flexible mass production. Actually, this has only been done in one project till now, the project of developing a financial system to the Danish trade schools. However, the experiences of using the standards and guidelines were so positive that we now regard them as inseparable from the tools and standard modules.

We have standards for specifications, screen layout, database tables, programs, and documentation. Most of the details of the standards are created and maintained during the project and are used to ensure consistency throughout the system and between subsystems.

We have guidelines for many of the typical system developer's tasks in the project. Guidelines for how to make a new screen, a new report, a new table with standard operations in the database, a new complex database operation, a new interface server between subsystems, a new operation in an interface server etc. The guidelines offer a different entry to the tools than the set of manuals available for the tools. The guides take the task at hand as the starting point. They describe step by step what to do and what tools to use to do the job.

A few of the guidelines are drafted in the beginning of the project or could be inherited from earlier projects, but most of them are created and maintained during the development process by the involved system developers themselves.

Therefore, the guidelines at all times reflect the accumulated experience of optimal use of the available tools.

5 Advantages of the Architecture

In this section, the architecture and our practical experiences with it are related to the requirements for a software architecture for information systems, presented in Section 2. The discussion is structured as a set of assertions with related reflections. Aspects of our production environment are also considered when they contribute to the quality of the development process or the developed product. Moreover, development of systems with this architecture is compared to development of systems by means of fourth generation languages (4GL).

5.1 Maintenance and Enhancement

The architecture supports maintenance of functionality
By maintenance of functionality, I mean changes and additions to the database, corresponding changes to screens and reports, changes in layout and

functionality of screens and reports, new screens and reports or whole new subsystems.

Changes to the database may be needed in order to enhance the data model or in order to change the representation of existing data for instance to improve performance of critical database operations. When changes to the database are made, the logical database layer shows some of its strength. The application programs contain only calls to logical database operations. The correspondance between these logical operation and their representation in terms of the underlying physical database system is represented only once, namely in the dbt-file used by the database server program. If the database change is only a structural reorganization of tables and fields, only the implementation of the logical database operations in the dbt-file has to be changed. All applications and even the database server program can remain unchanged.

If the database change involves definition of new fields or removal of existing fields, the format of the communication record used between clients and database server processes is changed accordingly. For applications that do not make use of new or removed fields, recompilation is sufficient. Applications that make use of removed fields or should make use of new fields must of course be more substantially changed.

When general changes to the user interface are to be made, changes can be localized to the general screen program, and individual treatment of all applications can be avoided.

When individual changes to screens and reports are to be made, the common structure of the programs is a big advantage. Moreover, most programs are relatively small, since most general properties are separated from the specific. This makes minor changes easy to perform, also for other programmers than the one who made the program.

When new screens or reports are to be developed, the guides for development described in Section 4.2 are invaluable. Without such guides, a lot of knowledge about optimal use of the tools would be lost, because of the gradual turnover of labour among system developers.

4GL are inferior with respect to maintenance of functionality
Typically, applications developed by means of 4GL have access to the data-

base directly incorporated in the source code in the form of embedded SQL expressions. SQL expressions involving a specific database table are spread all over the system's source code. When changing the database table, all these expressions must be found and changed throughout the entire system's source code. In practice this makes the process of changing the database extremely vulnerable and time consuming.

General changes to the user interface is either impossible, in case the required changes involve aspects of the interface defined by the 4GL itself, or a major task involving all applications developed so far.

With respect to individual changes to screens and reports, 4GL is comparable to our architecture.

The architecture supports customization
If the same system is used by several different use organizations, these may have different requirements. Some differences can be easily handled within our architecture.

Assume that two use organizations want two different versions of a specific subsystem, because their work practice differ in this particular area. Provided the two versions of the subsystem can offer the same interface to other subsystems, that is, the interface server processes of the two versions of the subsystem can support the same logical operations, the two alternative versions can be implemented with no effect on other subsystems.

The architecture supports performance tuning on location
Besides differences in functionality requirements, there may also be different performance requirements to the system in different use organizations. For instance because of different size of database, different number of users or different hardware configurations. In this case some individual tuning of the system can be done in the database server layer of the system.

Assume that information has been gathered in a use organization about which logical database operations are used most often and which are most time consuming. We are able to do that by telling the database server processes (on the fly) to output identification and time consumption of all logical database operation that execute for more than a certain threshold of time. Now, the division of labour between the database server processes, i.e. who should support which logical database operations, can be tuned. For instance, some database server processes could be dedicated to time consuming operations that are noncritical with respect to response time, and others could be dedicated to operations that are often used and require fast response. The database server processes are all instantiations of the same program, and the distribution of logical database operations among them is described on a text file, which can be edited on location. We could even decide to add an extra database server process to the system.

Of course some effort in this direction, determining the default situation, must be done before installing the system, but the point is that there may be individual needs to meet.

Unfortunately, our current practice on installing new versions of systems imply that such individual tunings are lost. However, that is a problem in our current practice and not inherent in the architecture. On installation, changes and additions to the file describing distribution of operations should be installed in some intelligent way rather than just installing a whole new version of the file.

5.2 Division of Labour

The architecture is superior to 4GL in supporting division of labour
The subsystem division makes it possible to have several teams of developers work in parallel in autonomous subprojects, coordinating only through the design of interfaces in the server layer of the subsystems. More complex coordination can then be limited to take place within a subsystem rather than between them. This limits the number of other subtasks that each subtask may need to coordinate with.

Within a single subsystem, division of labour can obviously be based on the individual screens and reports. However, even more finegrained division of labour is supported, since the task of implementing specific services in the logical database layer or in the server layer can be assigned to individual developers.

The division of labour supported by 4GL is less finegrained. The only obvious division of labour is based on individual screens and reports, since database tables, fields and operations are designed as an integrated part of the application development.

Moreover, the lack of a subsystem division when using 4GL makes it likely that coordination cost will grow more rapidly as a function of the number of developers sharing the task. The reason for this is that there is no obvious way of reducing the number of subtasks that each subtask needs to coordinate with.

It is thus likely that the inherent limit to the number of developers that can meaningfully share the development task will be lower when using 4GL than when using our architecture.

Production of system components is as efficient as with 4GL
It is our experience that given our production environment, production of system components like screens and reports, including the production of associated database operations and interface operations, is as efficient as production of similar components using a 4GL. Our experience is primarily based on comparison with Oracle's SQL*Forms.

The guidelines help improve productivity and quality
We experience a high motivation among the system developers for creating and maintaining the guidelines, because they are immediately useful. The guidelines often reveal differences in the way different developers use the tools and cause explicit discussions about practice. These discussions sometimes result in suggestions for new tools or improvements in the existing tools. In this way practice can be improved to increase both productivity and quality.

The guidelines are the key to resource flexibility
Since the guidelines at all times reflect the accumulated experiences of optimal use of the available tools, they are invaluable in the training of new system developers on the project. The guidelines are a key to resource flexibility in the project, i.e. the ability to utilize extra resources in peak periods.

5.3 Portability

The architecture supports portability to another operating system
If the system has to be ported to a different operating system, only the communication module needs to be rewritten. All other source codes can remain unchanged (provided they are written in some portable programming language) and just need recompilation in order to generate code to the new platform and link to the new communication module. Of course, the architecture requires an operating system with support for multiprocessing and communication between processes.

The architecture supports change of database platform
If the system has to be changed to use a different database system, only the database server program needs reprogramming together with the dbt-file containing the representation of logical database operations in terms of operations on the underlying physical database system. The other processes in the system remain unchanged, since the application processes communicate with the database server in terms of logical operations, and the format of the communication records remains unchanged.

The architecture supports change of user interface tools
If another user interface tool is to be used, the general screen program must be rewritten. Again, the other processes are unaffected. At least in principle. In practice, a change of user interface tool will often be followed by a wish to make use of new facilities in the new interface tool. For instance, if the change is from a character oriented interface to a graphical interface. If the graphics facilities are to be fully utilized, there will be additional changes to be made in the application layer of the system and maybe in the representation of the individual layout properties of screens. However, to obtain the same functionality of the system on a new user interface tool, reprogramming of the general screen program is sufficient.

This is obtained by the separation of the data model, functionality and user interface into different layers of the architecture. In this respect, our architecture resembles the model-view-controller (MVC) concept in Smalltalk-80. For an introduction to MVC, see (Pinson & Wiener, 1988).

The architecture supports distributed configurations

If the system is to run in a distributed configuration across a network, only the communication module needs to be reprogrammed so that processes communicate across the network instead of just through the operating system. All programs for processes in all layers in the system architecture can remain unchanged.

The changes needed in the communication module are of course nontrivial, but could be implemented by means of a commercially available package for Remote Procedure Calls (RPC). For a closer description of RPC, see (Birrell & Nelson, 1984). However, we have chosen to make our own RPC, primarily because all the commercially available products that we know of require communication records to be known at compile time, which we find unsatisfactory. See for instance the NCS product from Hewlett-Packard described in (Kong et. al., 1990). We want to maintain the flexibility described in Section 4.1, that new logical database operations can be added without recompilation of the database server program. By now, we have a prototype running that can communicate between equal UNIX machines (i.e. without data conversion).

The architecture will also need to be augmented with additional permanent processes to control communication, and administrate replicated tables describing distribution of operations and physical location of processes.

As it can be seen, our architecture does not reduce distribution to a question of a few lines of code, but the important thing is that the change from a centralized to a distributed version can be done without touching the great bulk of code in the four layers of the system that implement the system's data model, functionality and user interface. Changing a centralized system to a distributed system costs approximately the same as it would cost to establish a production environment initially for developing the system in a distributed version from the beginning.

There is one condition for this to be true. Error handling at the application level must be transparent to distribution. Both the centralized and the distributed version include communication between processes, and such a communication may fail for various reasons. Exceptions should therefore be designed to be general enough to cover both the centralized and the distributed case. If different error messages are to be given at the application level, error codes can be used, and their relation to various messages maintained in a single module.

When the above changes have been made to the communication module, the system can be distributed according to several different patterns.

The database server processes could be placed on a server machine together with the physical database, whereas the application processes and screen processes could run on the individual user machines. Alternatively, since the presentation layer and the application layer are implemented as separate processes, they can also run on different machines. If for instance a specific application process requires special computational power available on a particular computer in the network, the screen process can run on the user's machine while the application process runs on the more powerful computer.

These patterns of distribution all make the system meet the definition of client-server processing given by Buzzard (1990) where a continuum of distributed processing schemes is described.

The architecture supports heterogenous configurations
If the system is to run on a heterogenous configuration of machines with different user interface tools, there could be a separate version of the general screen program for each type of machine, whereas each application program needs only exist in one version, because the application process is separated from the screen process.

If the system is inhomogenous in the sense that it involves maintenance of several different databases, the problem of handling this is isolated to the database server program. The database server program should then exist in several versions, one for each underlying database system, for instance Oracle and Ingres. Logical database operations on the Oracle database should be supported by database server processes instantiated from the Oracle database server program. Similarly for operations on the Ingres database. All other layers in the system's architecture need not be aware of which parts of the data model are stored in the Oracle database and which in the Ingres.

When combined with distribution, heterogenous database systems could of course run on different machines together with their associated database server processes. Note however, that this does not mean that the architecture directly supports distributed databases in the sense, that the same database is distributed and replicated on several machines. This kind of distribution could be integrated in the architecture, but it requires an underlying database system that supports distribution.

The architecture supports integration with other systems
When an information system, A, with our architecture, needs to be integrated with another information system, B, this can be done at the server level. The interface server processes of A should implement operations needed by B, and B should be augmented with an interface server process if there is not already one in its architecture. To A, B will then look exactly like any subsystem of A itself.

4GL are inferior with respect to portability
A 4GL system is dependent on a specific database system and a specific interface tool. The system can only be moved to another platform if the 4'th generation tool is available on this platform or the whole system is reprogrammed.

Changing the configuration of a 4GL system from a centralized to a distributed configuration is also limited by the present capabilities of the 4GL, unless you are willing to reprogram the total system.

5.4 Growing Instead of Building Systems

The architecture supports growing systems
When growing a large information system, a top-down design is needed in order to establish a skeleton around which to grow the system. The architecture provides such a skeleton. The architecture is independent of the actual data model, functionality and user interface of the system. Our experiences with using the architecture in connection with an experimental project model for system development are described in (Thomsen, 1992).

4GL are inferior with respect to growing large system
When growing large systems, the lack of such an architectural skeleton that modularizes the system appropriately, will be serious, while less important when growing small ones.

When growing a system, changes of the system's functionality and user interface will be needed during the development process. Users evaluating the system or prototypes of it will propose changes that will make the system better match the use situation. Growing a system therefore requires the same properties of the system as described in Section 5.1 on maintenance and enhancement. Here it was argued that 4GL are inferior to our architecture.

When the users want a general change in e.g. the user interface, our tools are tailored to meet the new requirements. The direct access to tailoring our tools to needs makes production more flexible than when using 4GL, where all design must take place at the premises of the available static tool.

The overhead of using the architecture is insignificant
A potential disadvantage of our architecture is the large number of processes in the running system, compared for instance to the number of processes resulting from the architechture of a 4GL system. Processes take up space and communication between processes may reduce performance. However, in a running system with many users, there may be considerably fewer database processes running than in a 4GL system, where each user typically results in a separate database process. Since the number of database processes is often critical to the price of the database system software, and the database processes often take up much more space than other processes, our architecture may often be superior also in this respect. Moreover, the database server processes are all instances of the same program and the code is therefore only allocated once, regardless of the number of actual database server processes running. Only the data segment of the processes takes up space. The same applies to the screen processes in the presentation layer.

With respect to overhead in time, it is our experience that the time spent on communication is insignificant compared to the time it takes to access the database and compared to user response time in the interaction on screens. It is our conviction that communication time is not significant to the response time of the system as experienced by the users. Of course, the occurrence of bottlenecks in the system is critical. If long queues of requests to server processes occur, the distribution of services among them must be reconfigured.

However, it should be noted that our architecture is intended for large and medium size information systems. For small systems with only few users, 4GL may be a better choice.

6 Conclusion

A general software architecture has been presented. The architecture is a multi-level client-server architecture, where all dependencies on hardware and software platform are encapsulated into modules. The architecture provides a skeleton around which to grow an information system. It supports division of labour during the development process, and maintenance, enhancement and portability of the resulting system. The architecture is independent of the actual functionality of the system and is based on relatively stable properties of information systems in general, while design decisions that are likely to change are encapsulated into modules.

Several projects have successfully used the architecture and practical experience shows that the architecture provides a reusable starting point for new projects. A brief description of an existing production environment has been given. The production environment contains standard modules and guidelines to be used in the development of information systems with the suggested architecture.

The architecture has been compared to the architecture obtained when using 4'th generation languages. It has been argued that for large systems, the proposed architecture is superior with respect to its support for division of labour during development of systems, its support for growing instead of building systems, its support for maintenance and enhancement of the resulting systems and its support for portability of systems.

Acknowledgements

I am grateful to all my collegues at Mentor who share with me the experiences that inspired this paper. Special thanks to Ole Jacobsen for sharing his insight with me. Thanks are also due to Susanne Bødker, Joan Greenbaum, Kaj Grønbæk, Morten Kyng, Ole Lehrmann Madsen and Peter A. Nielsen who commented on earlier drafts of this paper.

This work was supported by The Danish Natural Science Research Council, grant no. 11-8385.

References

Birrell, A. D. & B. J. Nelson, (1984): "Implementing Remote Procedure Calls", ACM Transactions on Computer Systems, 2(1):39 - 59.

Brooks, F. P., (1975): "The Mythical Man Month - Essay on Software Engineering", Reading, MA: Addison-Wesley.

Brooks, F. P., (1987): "No Silver Bullet", Computer, 20(4):10-19.

Buzzard, J., (1990): "The Client-Server Paradigm: Making Sense Out of the Claims", Data Based Advisor, August 1990, pp 72-79

Coad, P. & E. Yordon, (1991): "Object Oriented Analysis", Second edition, Prentice-Hall, Englewood Cliffs, NJ, 1991.

Floyd, C., (1984): "A Systematic look at Prototypes", in: R. Budde et. al. (eds.): "Approaches to Prototyping", Proceedings of the Working Conference on Prototyping, Springer-Verlag, Berlin-Heidelberg-New York -Tokyo, pp. 1-18.

Floyd, C., (1987): "Outline of a Paradigm Change in Software Engineering", in: G.Bjerknes et. al. (eds): "Computers and Democracy", Avebury, pp. 191-210.

Jackson, M., (1983): "System Development", Prentice-Hall, Englewood Cliffs, NJ.

Kong, M. et. al., (1991): "Network Computing System Reference Manual", Prentice-Hall, Englewood Cliffs, NJ, 1990.

Pamas, D. L., (1972): "On the Criteria To Be Used in Decomposing Systems into Modules", Communications of the ACM, 15(12): 1053-1058.

Parnas, D. L., P. C. Clements & D. M. Weiss, (1985): "The Modular Structure of Complex Systems", IEEE Transactions on Software Engineering, SE-11(3):259-266.

Pinson, L. J. & R. S. Wiener, (1988): “An Introduction to Object-Oriented Programming and Smalltalk”, Addison-Wesley Publishing Company.

Thomsen, K. S., (1992): “he Mentor Project Model: A Model For Experimental Development Of Contract Software”, DAIMI PB 401, Computer Science Department, Aarhus University.