Object Oriented Development in an Industrial Environment

Ivar Jacobson

Dept. of Computer Systems
Royal Institute of Technology
S-100 44 Stockholm, Sweden

or

Objective Systems SF AB
Sjöängsvägen 5
S-191 72 Sollentuna, Sweden

Object-oriented programming is a promising approach to the industrialization of the software development process. However, it has not yet been incorporated in a development method for large systems. The approaches taken are merely extensions of well-known techniques when 'programming in the small' and do not stand on the firm experience of existing developments methods for large systems. One such technique called block design has been used within the telecommunication industry and relies on a similar paradigm as object-oriented programming. The two techniques together with a third technique, conceptual modeling used for requirement modeling of information systems, have been unified into a method for the development of large systems.

1. INTRODUCTION

A new programming paradigm - object-oriented programming - is getting a wide acceptance as a standard basic technology within the software industry. The new technique changes the nature of software design from being an art or a craftsmanship to being an industrial process. Software products can be developed in a way similar to hardware products. Programming changes from writing instructions to interconnecting reusable software components. The system designers will be equiped with a component catalogue, the components of which he interconnects to get higher-level products - application products or other more specialized components.

Although the technique is very promising and has been applied to the design of some real systems, it has not yet been developed into a design method for large systems (let us say systems larger than 10 man-years). The approaches taken here ([3], [7]) are natural extensions of techniques for 'programming in the small' but they do not stand on a firm experience of existing development methods for large systems. Such systems are today developed using techniques such as SREM [2], PSL/PSA [22], SADT [21], JSD [14] and similar techniques.

This paper presents an object-oriented technique for developing large, even very large, systems in an object-oriented fashion. It relies on three independently developed techniques - all three carrying the major characteristics of object-orientation: data abstraction, information hiding, dynamic binding and inheritance. These three techniques allow themselves to be naturally unified.

The framework of this technique is a design technique, here called block design, originating from Ericsson Telecom [16, 17] which is now widespread within the whole telecommunication industry. The technique is basically the paradigm behind the CCITT [8] recommendation SDL - the Specification and Description Language. Here a system is seen as a set of properly interconnected building blocks - each building block representing a packaged service of the system. With a complete specification of all the requested services of a system, the building blocks are selected in a top-down fashion based on criteria simplifying system configuration and adaptation to changes in service requirements and in new technology.

This technique has now been used for twenty years in the development of large commercial systems. It has been used for projects comprising hundreds of designers and today there are more than a thousand designers practising it at a large number of sites all around the world. Showing very positive results the technique has been applied to large systems such as real time systems, development support systems, office automation systems, process control systems, centralized or distributed systems. Furthermore, our technique supports in a natural way functional changes and adaptations to new technology. This property in particular gives our systems a long life.

The block design technique has been improved with two other techniques to cover the whole system development work:

1) Conceptual Modeling which has developed within the realm of informatics [19]. The approach emphasizes finding the concepts suitable to model the problem domain. Methods and languages for conceptual modeling have evolved over the last years and have even been required by ISO [26]. An overview of some approaches for conceptual modeling is provided in [4].

2) Object-oriented programming which is a basic technology for designing software components that can be put together to form higher-level application modules. These modules have been identified through the block-oriented design step in which a large system is decomposed into configurable building blocks.

Here we will outline this technique which is called ObjectOry™ We will first present the system development activities and then discuss the concepts used when modeling such systems. We will also briefly indicate the tools we need to develop systems in a rational manner.

2. SYSTEM DEVELOPMENT

The scenario we will assume when building a system is similar to the manner in which construction is carried out in many other disciplines such as house construction, design of electronic systems, or in simplified form: A **system** is built of a set of application modules which in the following will be called **blocks**. A block may itself be made up of other, lower-level blocks or by **components**. Components are standard modules which can be used for many different applications. The lowest-level blocks are made up of components only. Blocks as well as components are naturally implemented as classes using object-oriented programming. The designers are consequently provided with a set of components when building applications by means of blocks.

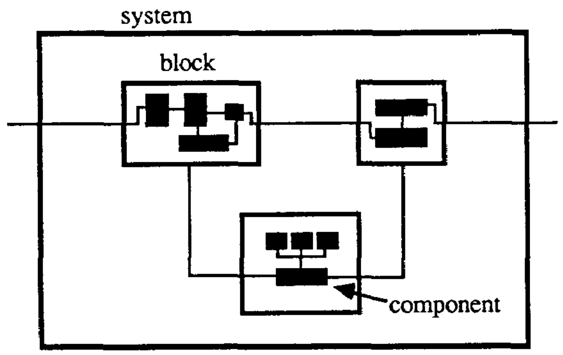

Fig.1 A system is composed of blocks. Blocks are designed using components.

We will now present a strategy to develop large, commercial software systems which are built on this scenario.

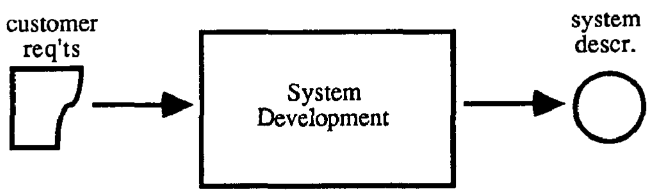

Fig. 2 System development, its input and output.

The input to System Development is the customer requirements and the output is a system description including the complete program code.

The customer requirements and the system description are two different, yet related models of the behaviour of a system. The requirements should be met by the system description. The program code included in the system description has a formal meaning and can be interpreted unambiguously.

Expressed simply, system development contains a set of transformations starting from the requirements model and ending with the interpretable program model. In between these two 'end-points' a number of other models are being designed. The objective is to partition the complex work on a large system into steps and to allow more developers to participate in the work. Each new model gives the developers an abstraction of the system to make the proper decisions on how to get closer to the final model - a well-formed system. Every new model gives more formal structure to the system than the previous one. For the time being our technique suggests only one formal (formal in the mathematical sense) model of the system namely the program. We do not think that the use of formal specifications has been practically demonstrated. Such approaches either seem to be too poor, e.g. algebraic specification [13], only applicable to smaller classes of systems, or they are swollen and based on several paradigms but not offering more expressability than modern programming languages such as Smalltalk-80™ [25] does.

In order to make transfers between the different models as simple and free from errors as possible, it must be easy to relate one model of the system to the next. We say that two models are *seamlessly* related to one another if concepts introduced in one of the models can be found in the other model through a simple mapping. Seamlessness is an important property of modeling, since it is a prerequisite for the development of powerful support tools.

The simplifications, made for this presentation only, are essentially three:

We have assumed here that the system is developed for one customer only. A system is often developed for many customers and must be configured to meet the individual requirements of each customer.

The work on different models has been described as sequential. Normally it goes on in parallel.

Only the first development cycle has been outlined. Most commercial systems must live a long life and during this time they undergo constant changes. The described technique is especially oriented to support changes. We view system development as an activity that changes a system from being one 'thing' to being another different 'thing'. The first development cycle is only a special case and a change from 'nothing' to 'something' [18].

System Development is naturally divided into two separate but interacting organization units, here called factories: System Analysis and System Design. A *factory* encapsulates the activities as well as the objects manipulated by the activities. Factories contain different concurrent processes that communicate with one another using some kind of messages.

System Analysis uses the requirements and produces in cooperation with the customer a specification of the requested system. The other sub-factory, System Design, transforms the system specification into a verified system description.

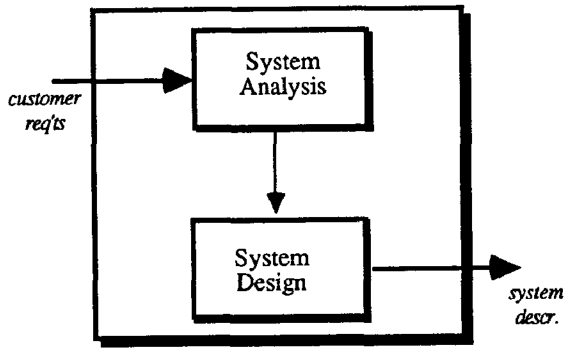

Fig. 3 System Development contains two sub-factories, System Analysis and System Design.

The System Analysis specifies the functional behaviour of the system under ideal conditions. It is consequently assumed that an infinite processor capacity is available, that the execution speed is immensely high, that there is endless storage volume, and that you can disregard parallelism and instead serialize the course of events.

In the System Design these ideal conditions will successively be replaced by the selected implementation environment including operating system, data base management system, user interface library, etc. However, the structure of the specified functional behaviour must not unneccessarily be violated by the chosen design environment.

3. SYSTEM ANALYSIS

3.1 Sub-Factories

System Analysis contains three sub-factories corresponding to three models of the system.

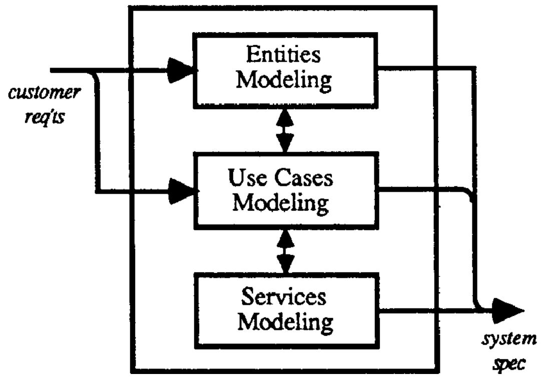

Fig. 4 System Analysis with its sub-factories.

We have already mentioned two of the concepts in Objectory, namely blocks and components, and that the blocks of the lowest level and the components are implemented as classes in an object-oriented language, e.g. in Smalltalk-80™ or Objective-C™. They will be described here in more detail and three additional concepts will be introduced: entities, use cases and services.

Conceptual modeling as described for example in [5] or in [4] is very well developed to model the static objects of the real world and the relations between these objects. Thus this technique has been adopted in Entity Modeling where entities correspond to these real world objects. However, the technique is not developed well enough to model dynamic behaviour. See e.g. [26] in which the requirements for static modeling are very elaborate but the requirements for dynamic modeling are almost absent. Therefore we have made some natural (seamless) extensions which allow us to also model the events of the system. Our extensions are two new concepts: use cases and services.

3.2 Systems

For completeness we will just mention that systems are modeled in System Analysis by three related conceptual models - the entity model, the use case model and the service model. These three models will be presented subsequently and they will together be referred to as the system specification.

3.3 Entities

Entities are things or events in reality which are of interest to the system that is being developed. They are things close to reality about which you want to make assertions and about which you want information. For example, subscribers, lines, routes, etc. are entities in a telephone system, and customers, accounts, and bank offices are entities in a banking system. Since the entities mirror objects in reality, it is of utmost importance to reflect the entities in the developed system. Then the system can follow reality and changes in reality can be transferred in a simple manner to changes in the system.

Entities are of two types:

Objects that can have an existence that is independent of other objects. For example, bank and customer are two such entities that can exist independently of each other.

Objects whose existence depends on the existence of other objects, for example a bank account, which requires the existence of both a bank and a customer. This latter type of entity is sometimes called relationship.

Entities are related to one another in many different ways. We distinguish between *ISA*-relations and *attributes*. ISA-relations are class membership, such as J. Smith ISA customer, or class relations, such as customer ISA person. Attributes are all other types of relations between entities, such as the relation between a customer and his bank account.

Entities are described by lexical objects, here called *data*. For example name, acoount number, street name. All this data is simple in contrast to structured data, for example an address.

The entities of a system are modeled by a so-called conceptual diagram i.e. a graph consisting of nodes and arcs between the nodes. In this case the nodes correspond to entities and the arcs represent relations between the entities.

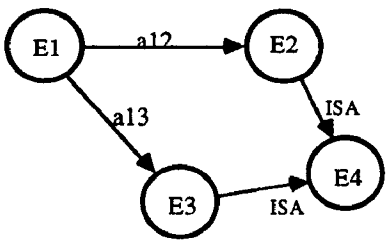

Fig.5 A schematic conceptual diagram modeling some entities and their relations .

The output of Entity Modeling is used to model the customer requirements and it will be used by Use Case Modeling. It will later be used by System Design to give an appropriate structure to the developed system.

3.4 Use Cases

A large system participates in the execution of a great number of functions. The static structure of a system is usually described in the form of a tree, where the root is the system and the other nodes constitute the system's functions, data, processes, etc., in other words what we usually call its modules. The dynamic behaviour of a system is not possible to sum up in one single description. Usually the entire behaviour is described by describing its parts, i.e. by describing the behaviour module by module in the system. These assertions apply to several of the techniques mentioned earlier. The system is described in ObjectOry as a black box by describing a number of aspects of the system. These different aspects, each corresponding to a behaviourally related sequence, are called *use cases*.

The basis of ObjectOry is that it shall be designed for its users. We want to build the system for them. In order to safeguard that the users really get the system they want and need, we want to structure the system's total behaviour in aspects, where each aspect corresponds to what we can call a use case. The collected description of the amount of use cases then constitutes the total behaviour of the system and is the input to the System Design.

It is then a matter for the designers to guarantee that the requested methods of using the system are implemented by means of design elements, in our case by means of blocks. The designers therefore structure their descriptions of the system according to these use cases. Finally tests are made to assure that the completed system fulfills these use cases and consequently that the user will receive the requested system and nothing else. It must therefore be possible to trace a given use case throughout the entire design. We will revert to this in chapter 4.3.

What is a use case? In order to define this concept, we must first define two other concepts: role and user. The word user as used in general is a far too vague term and we need something more precise.

A system is handled during its lifetime by a great number of different persons who are acting in different types of functional roles. A *role* is defined through a specified task or a group of closely related tasks which are performed by persons during the development and/or operation of a system [19]. We say that persons in different roles handle the system. Examples of such roles are specifiers, designers, testers, operators, etc.

A *user* is a specific type of role, namely a role that participates in the operation of the system. Marketers, designers, installers, etc. are other roles - but they are not users. Examples of users of a telephone system are subscribers, operators, and subscribers connected through lines. In a banking system, the bank clerk and loan officer are different users. The fact is that an ordinary subscriber represents two different users: The A-subscriber (calling subscriber) and the B-subscriber (called subscriber). One and the same physical person can consequently have several different roles. The behaviour as A-subscriber is totally different from the bevaviour as B-subscriber.

A *use case* is a special sequence of transactions, performed by a user and a system in a dialogue. A transaction is performed by either the user or the system and is a response initiated by a stimulus. When no more stimuli can be generated, all the transactions will finally ebb away. The use case is ended and a new use case can be initiated.

Example: In a telephone exchange the local calls and long distance calls constitute two different use cases. In both cases the user is an A-subscriber. In a banking system transfer from one account to another is a feasible use case, initiated by a bank clerk.

Only a specific user can initiate a given use case. This does of course not mean that other users cannot be called and connected to the use case. These other users can be of the same kind or of different kinds. Actually, the initiating user can leave the use case and the users connected later on can take over and terminate it. After having left the use case, the initiating user is free to start a new use case and this new use case can be of an entirely different type. A given user can initiate several different use cases.

It is proper to ask whether the number of use cases for a large system is not infinitely great. This is a philosophical question, but let us say that the amount can become next to infinite. However, with a correct structure for the description of the use cases, the description of the collected use cases will become large but nevertheless possible to handle. If you can present a manual for the system (which is necessary) then you can also present a surveyable number of use case descriptions.

The collected number of use cases is described in a conceptual diagram in which the nodes represent use cases or entities.

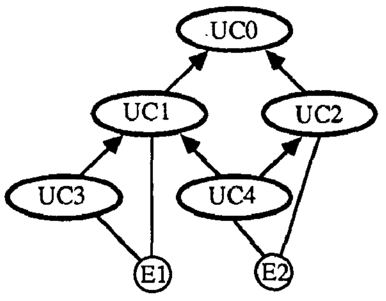

Fig. 6 A schematic conceptual model of the system's use cases.

The diagram contains two types of relations:

built-on relations which mean that one use case can be based on another, more basic use case. The term built-on is used to let a use case be a specialization of a more general use case. In this manner only the difference will have to be specified in the special use case. The built-on relation is an extended form of inheritance relation. Multiple inheritances are common.

access relations, which exist between a use case and the entities accessed (inspected or manipulated) by the use case.

For example, 'Ordinary local call' is a basic use case on which the use case 'Local call with abbreviated dialling' can be built. The description of the latter use case is made by referring to the description of the use case 'Ordinary local call'.

Each use case is described in a semi-formal manner by using structured English or by using graphics for example a data flow chart as suggested by Gane and Sarson [10]. You begin by describing the normal manner of using the system for the revelant case. Other methods of using the system that shall be included will be described later. It is important that the description will not reveal the internal structure of the system. The system shall be regarded throughout as a black box.

The use cases are handled during a great number of activities. Therefore they have unique identities that follow them through these different activities. They are used in the discussions between the customer and the supplier (seller) before ordering the system. They are also used in the final phase when the system is delivered. They are used internally by the supplier to guarantee that the system is designed according to these use cases.

3.5 Services

A conceptual model consisting of entities and use cases is a complete, functional specification of a system in a semi-formal form. The collective behaviour of the system has been described in such a manner that both the customer and the designers can be satisfied.

The system is presented to the customer in terms of entities corresponding to the objects in his world, and in terms of use cases corresponding to the different manners in which his system will be used.

Through the entities the designers have obtained a source in order to structure their model of the system into suitable modules. By letting each entity correspond to a block, we obtain a system which seamlessly reflects its surroundings. Changes in the surroundings, which frequently are local in existing real objects, will then also be local in the system.

Unfortunately the use cases are not that easy to modularize. A use case cuts right through the system and normally means that many mutually dependant system functions are used. Let us show an example. 'Local call' and 'Call to another exchange' will both contain one part each to handle the A-subscriber. These two parts are not identical but have much in common. 'Local call' and 'Call from another exchange' also have one part each - not identical but similar - to handle the B-subscriber. There is a great number of use cases that include one part to handle the A-subscriber and another part to handle the B-subscriber. There is similar experience from other types of systems such as banking systems. In order to modularize the use cases we must therefore introduce a supporting term that corresponds to a cluster of these similar parts of use cases. We obtain clusters that handle the A-subscriber, clusters that handle the B-subscriber, etc. Such clusters are called *services.*

Services is an important supporting term used to identify a suitable structure for the system. Services have two important tasks during the system development:

> One is to identyfy the packets that contain behaviourally related functions so that changes in a functional requirement on a system normally will affect one packet and more seldom two or more packets. We want to give locality to the functions in order to achieve robustness in case of changes.
>
> The other task is to identify the different packets of functions that shall be offered to the buyers of the system. Each customer shall be able to order his own set of such packets.

Services are consequently a primary requirement when structuring the system and they are ordering units when offering the system. These two functions of services motivate that Service Modeling is a System Analysis activity and not a System Design activity

A *service* is an indivisible ordering packet of functions for a specific system. Even if the packet could be implemented in several parts (with several modules) which often is the case, each customer wants to have either all such conceivable parts or none at all.

The services are related to the use cases in such a manner that:

> A use case employs a specific set of services.
> A service can participate in many use cases.
> The number of use cases for a specific customer determines the mixture of services to be ordered.

The service concept is explained simply with the following reasoning:

1) Use a specific stage in the development of the system as a basis. When this stage has been reached, the system is complete to some extent, i.e. it contains a number of functions that can be regarded as meaningful even if it is not sufficient. Let us assume that at this stage we have a system that can be installed and taken into operation. The users will receive something that is working even if it does not have sufficient functionality. On the basis of this system we can start thinking in terms of services. Each new, indivisible packet of meaningful functions constitutes a service. It is indivisible if it cannot be divided into smaller packets that each has a meaning to the users.

In this manner we build up a system in indivisible packets, each corresponding to a service.

2) Now we retreat and think in the same manner when we build the first issue of the system. We try to find complete, indivisible packets of the system that are related logically. First we identify the packets of functional requirements of which each should be optional and handled individually. Optional means requirements that one but not all customers desire. The functional requirements that all customers desire finally constitute the mandatory services. They are identified with some heuristics: a) A user role is connected to one service only. b) A service should normally only be attached to one role. For example, one service can handle A-subscribers, another can handle B-subscribers. c) Identify services that could have been treated as optional even if no customer wishes to exclude them at present. d) Collect functional requirements to one service which are 'logically' related with the rest of the service.

The conceptual model of the services in the system shows all the services and their mutual relations.

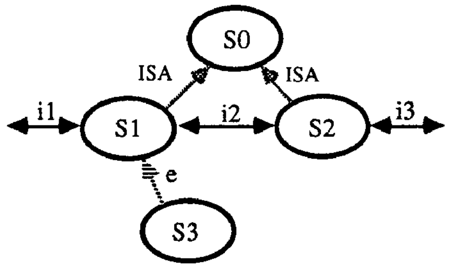

Fig. 7 A schematic conceptual model of a system's services.

There are three types of relations between services:

> *ISA*-relations which mean that a service offers the same behaviour as another service but in addition has a special and related behaviour. The service is a specialization of another, more general service. The ISA-relation is an extended form of an inheritance relation. Multiple inheritance is common.
>
> *extensions*, which mean that a service is an extension of another service [18] and invoked by this other service. This kind of relation is very useful when making functional changes in an existing system without changing the existing, well-working services of the system (e in fig.7).
>
> *interactions* which mean that a service communicates with another service and transfers information (i1, i2 and i3 in fig.7).

The conceptual model of the services is the input to the system design activities and provides support when selecting the architecture for the system.

4. SYSTEM DESIGN

4.1 Sub-Factories

Input to System Design is the system specification with the three conceptual diagrams of entities, use cases and services. System Design itself is partitioned into three sub-factories:

> System Level Design, in which the system structure is identified.
>
> Block Design of which one factory is created for each block in the system.
>
> Component Level Design, in which the system components are identified and designed. For very extensive system development activities, this factory should be common to the design of several types of systems. It should then be a factory on the same level as System Design.

The conceptual models will now be seamlessly transformed into design models. Each one leads us closer to the final implementation in the form of the program code.

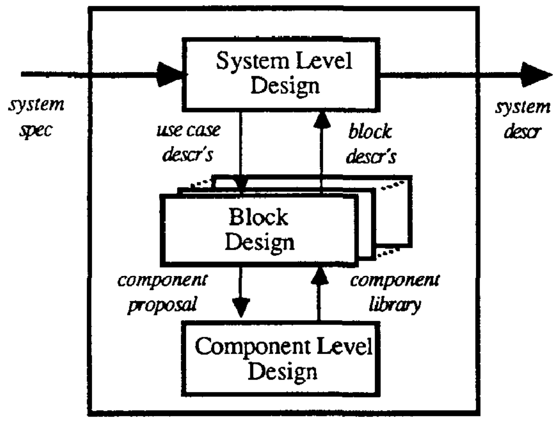

Fig. 8 System Design and its sub-factories.

System Level Design is in turn divided into four sub-factories:

System Structuring, in which entities and services are used to identify the blocks of the system in a seamless fashion. A new model of the system is formed: The system is a set of communicating blocks.

Use Case Design, from which you create as many processes as there are use cases. The conceptual model of the use cases is translated seamlessly into a new model showing how each use case is implemented by means of the identified blocks.

Use Case Testing. The blocks that implement a use case are combined and the use case is tested. Each use case is tested separately to safeguard that the system meets the requirements of the user. Please note that the use cases constitute the key aspect through the entire development activities.

System Testing, during which integration tests are made of the total amount of use cases.

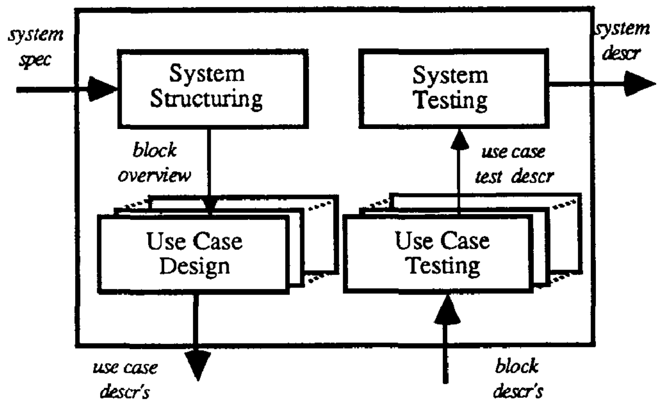

Fig. 9 System Level Design and its sub-factories.

Component Level Design contains two sub-factories:

The Component Group which consists of designers with experience of several applications. The Component Group is responsible for the component library and consequently approves specifications and descriptions of new components.

Component Design, of which one factory is created for each component in the library.

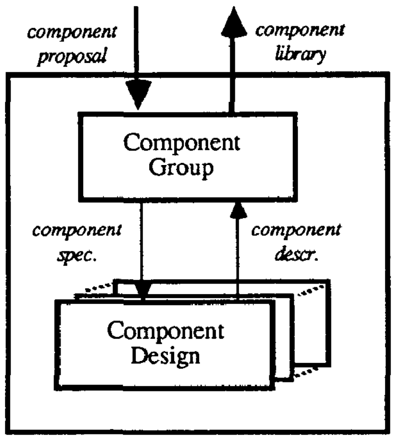

Fig. 10 Component Level Design and its sub-factories.

We will now describe the concepts used in System Design in more detail. These concepts are blocks and components, mentioned several times in the preceding pages. However, a new model of systems and a new model of use cases - both being design models - will first be introduced.

4.2 Systems cont'd

The design model of a system is statically a set of interconnected blocks. These blocks communicate dynamically over well-defined interfaces by sending some type of stimuli to one another. These stimuli can have different semantics depending on how these blocks are implemented. They can be signals and have a 'send/no-wait' semantics such as in PLEX (a programming language developed by Ericsson Telecom) or CHILL [9] (developed by CCITT). They can be messages with a 'rendez-vous' semantics as in Ada [1], or they can be messages with a 'send/wait' semantics as in Smalltalk-80 [25]. Blocks are finite state machines, which given a state and an input stimulus produce a response and a new state.

The selection of a block structure is a comprimise between satisfying the requirement on a simple, ideal structure and obtaining a system with good performance attributes. The ideal structure is achieved if each entity and each service defined during System Analysis will be implemented as a block. In many cases it is possible to translate an entity and a service directly into a block. When structuring the block, however, it is necessary to consider the available tools such as language, tools, implementation enviroment. Other behaviourial requirements such as response times, reliability and efficiency will also influence the structure. If the blocks communicate through messages due to the selected implementation technology, it is for example necessary to implement an extension (see section 3.5) by changing an existing block. An extension cannot be introduced simply by introducing a new block. The existing block must send messages explicitly to the new block.

In this manner we identify one set of entity blocks and one set of service blocks. For large systems it is practical to group these blocks in larger blocks called subsystems. This technology will not be discussed here. The design model is shown in the form of a block diagram presenting the blocks and the communication paths between the blocks. The diagram only shows the blocks corresponding to entities and services at the bottom of the ISA structure. Blocks corresponding to entities and services higher up in this structure will instead be shown in an inheritance diagram. This diagram shows the inheritance relations between the blocks. The semantics in these relations is determined by the selected programming language.

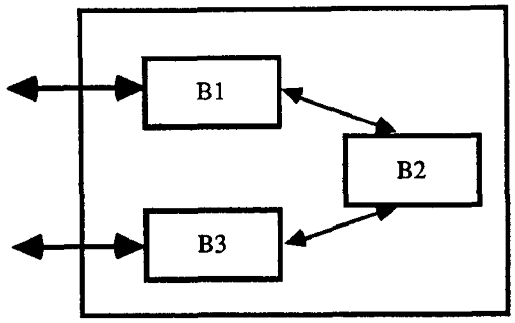

Fig. 11 A schematic block diagram.

4.3 Use Cases cont'd

Now that the blocks have been identified, the design work continues as each specified use case is implemented and described in terms of a subset of the blocks. This subset comprises the blocks that participate in the use case. This work is shown for example as an interaction diagram that shows on a vertical time axis the interactions between the blocks when carrying out the use case. The exact semantics of the diagram is specified by the selected programming language.

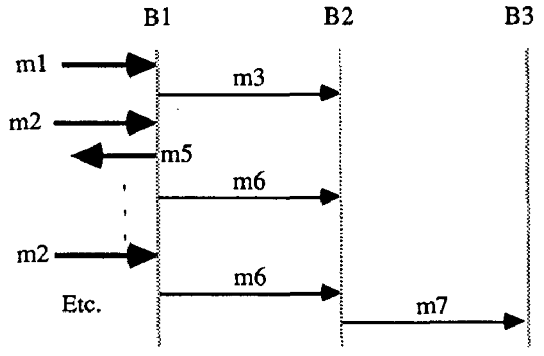

Fig. 12 A schematic interaction diagram.

A skeleton of an interaction diagram with the permitted communication paths can be generated mecanically from the block diagram.

4.4 Blocks

Blocks are application modules created in order to be combined with other blocks into a system. The blocks communicate with one another to fulfill the requested use cases. For example, a telephone system contains blocks to handle the communication with subscribers, blocks to handle the communication on junction lines to surrounding telephone exchanges, blocks to interconnect subscribers with other subscribers or lines, blocks to keep track of all the telephone calls, etc.

Blocks are reusable building blocks for customer adaption of a specific application. The services are related to the blocks as follows:

> There is an unambiguous relation between a service and a set of blocks. A given service mix indicates a set of blocks directly in the requested system.
>
> The blocks are equipment units in order to assemble a system for a customer. Services are units that can be ordered.

A block is classified as belonging to one of four classes depending on whether the block is mandatory or optional, and if it is a standard block or specific for the customer. Mandatory means that all customers require the block. Standard means that the block has the same function for all customers.

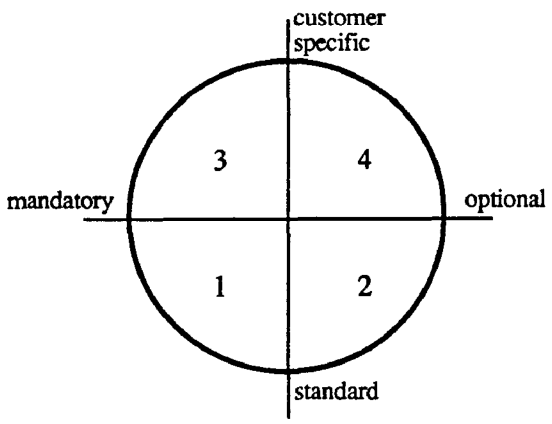

Fig. 13 Classification of blocks.

Each block shall be classified for one of these classes. The class affiliation will later control the further development of the block. A block placed in class 3 or 4 will to a great extent be dependent on the wishes of the customer, whereas a block in class 1 or 2 requires that you coordinate the wishes of different customers and try to find a general, often well worked out structure for the block. For customer-specific blocks, it is necessary to search for or create another block that is standard and higher up in the inheritance structure

The input to Block Design is a set of 'method stubs' produced mechanically from the set of use case interaction diagrams. A 'method stub' is the name of an input stimulus, possibly followed by a task-specifying text in structured English. The set of 'method stubs' of the block are found by collecting the use cases in which the block participates, and from their interaction diagrams extract the stimuli sent to the block. For instance, from the interaction diagram shown in fig. 12, the block B1 can receive two stimuli: m1 and m2. Other stimuli to the B1 block may be extracted from other use case interaction diagrams.

Blocks are implemented as classes in an object-oriented programming language. A service may require more than one class to be implemented, whereas an entity normally is implemented directly as a class. When designing a class corresponding to a block, components are used properly interconnected. The resulting source code of the block is run in an interpreter to verify that it behaves as intended by the designer. If this is the case the block is output for use together with other blocks in Use Case Testing.

4.5 Components

In the future every programmer will have access to one or several component libraries - browsers - similar to the electronic component libraries available to hardware designers today. At present most programmers only have some computer science books and language manuals, but no actual components. Attempts have of cource been made to find component libraries. For instance macro libraries, subroutine libraries, and procedure libraries have been prepared. These libraries, however, have not turned out as hoped for. To quote Gerry Weinberg [24]: "Unfortunately, program libraries are unique; everyone wants to put something in, but no one wants to take anything out." Why, then, has it been so difficult to prepare suitable libraries? It is not very difficult to prepare a procedure package to offer mathematical functions or similar well-defined functions. It is difficult, however, to prepare procedure packages in which the procedures have the side effect of changing more or less global data. Then this data must be defined and the procedures become increasingly complex. We can only develop relatively simple procedures; a small component library.

Components must be powerful, have simple, well-defined interfaces, be easy to learn and use, have a wide area of use, etc. In other words, components are highly re-usable program elements. Components in the software field are for example buffer, queue, list, trees, which are suitable for use in normal programming and to implement algorithms, or for example window, icon, scroll area, which are suitable when developing graphical man-machine interfaces.

Today we only have very few and limited component libraries, and consequently every software house must develop most components themselves. When designing large systems, a very important question is to find, where the border between the components and the blocks should be. The border is not razor-sharp but a compromise between the requirement for many reusuable program elements and the requirement for a surveyable component library

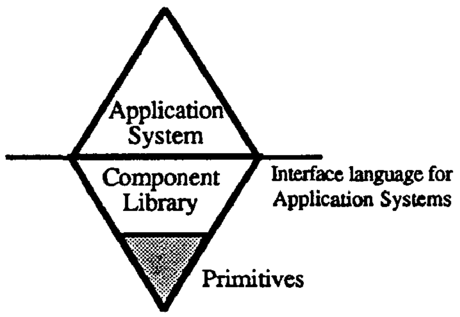

Fig. 14 The application system in relation to the component library.

The components are defined on top of each other (or on top of the primitives); they are designed bottom-up. This is in contrast to an application system which typically is designed in response to a system specification. Application systems are thus defined top-down using the component library.

The component library should ideally be specified before the work starts on the application system. Then the designers know exactly what means of expression are available while describing their blocks. This can be done for an existing system or for a system of a well-known application. When designing a system of a new kind, however, block design and component design must go on in parallel since the set of components to be used is dependent on the application.

During implementation of a block, the designer may recognize the need for a module that is judged to be used in several other blocks. If the module is judged to be sufficiently useful for many other blocks, the designer can propose that the identified module will be standardized and classified as a component. A group of highly qualified application - oriented system specialists - the Component Group - works as a forum for discussing and approving new types of components.

An approved component is specified and designed. Each component is implemented as a class in an object-oriented programming language.

Even if it must be possible to change the component library during the lifetime of the system, such changes should be rare since they influence the whole foundation of the application systems. The components must therefore be chosen with care. It may require several attempts before the chosen set of components for a particular type of application is so stable that meaningful work on a broad basis can be done.

5 CONCLUSIONS

A development method for large systems called ObjectOry™ has been presented. ObjectOry has combined a well proven technique for large systems design, called block design, with conceptual modeling and object- oriented programming. These three techniques are very natural to unify since they rely on similar paradigms aiming at, among other things, reusable software products.

The approach introduces a number of new features that are particularly useful for large systems. The most interesting features discussed in this paper are briefly:

The use case concept which assures a user-driven system development process.

The service concept which facilitates configuration management and supports robustness in case of changes.

The distinction between blocks, which are application modules, and components.

These features facilitate most system life-time activities and increase the productivity for large systems in an industrial environment. Due to the limited space of this paper we have only been able to indicate some other characteristics that we deem important in such an environment. These were:

The factory concept which allows modeling the complex development work on large systems in a realistic manner.

The change orientation of the system development process. Also the first development cycle is a change - a change from 'nothing' to 'something'.

ACKNOWLEDGEMENTS

The author is indebted to the L.M. Ericsson Telephone Co. for funding this research. He is also very grateful to dr. Tom Love, Productivity Products International, for his personal support and for his constructive criticism of an early version of the work.

REFERENCES

[1] Reference Manual for the Ada Programming Language, ANSI/MIL-STD-1815A-1983, 1983.

[2] M. Alford, A requirements Engineering Methodology for Real Time Processing Requirements, IEEE Trans. Software Engineering, Vol. SE-3, No. 1, Jan 1977.

[3] G. Booch, Object-Oriented Development, IEEE Trans. Software Engineering, Vol. SE-12, No. 2, Feb 1986.

[4] A. Borgida, Features of Languages for the Development of Information Systems at the Conceptual Level, IEEE Software, Jan 1985.

[5] J. Bubenko Jr., Information Modeling in the Context of System Development, Proceedings of IFIP 80, 1980.

[6] J. Cameron, An Overview of JSD, IEEE Trans. Software Engineering, Vol. SE-12, No. 2, Feb 1986.

[7] B. Cox, Object-oriented Programming, An Evolutionary Approach, Addison & Wesley, 1986.

[8] C.C.I.T.T., Fascicle vi.11, Functional Specification and Description Language (SDL), Rec. z.100-z.104, Geneva, 1984.

[9] C.C.I.T.T., Fascicle vi.12, CHILL, Rec. z.200, Geneva, 1984.

[10] C. Gane & T. Sarson, Structured Systems Analysis, Tools and Techniques, Englewood Cliffs, NJ: Prentice Hall, 1979.

[11] R. Gardner, Multi-Level Modeling in SARA, Proceedings of the Symposium on Design Automation and Microprocessors, Feb 1977.

[12] A. Goldberg & D. Robson, Smalltalk-80 The Language and its Implementation, Adison-Wesley Publishing Company, 1983.

[13] J. Guttag, Abstract Data Types and the Development of Data Structures, Comm. of ACM, June 1977.

[14] M. Jackson, System Development, Prentice Hall International, 1983.

[15] I. Jacobson, On the Development of an Experience-based Specification and Description Language, IEE Proceedings of Software Engineering for Telecommunication Switching Systems, July 1983.

[16] I. Jacobson, Concepts for Modeling Large real Time Systems, Department of Computer Systems, The Royal Institute of Technology, Stockholm, Sept. 1985.

[17] I. Jacobson, FDL: A Language for Designing Large Real Time Systems, Proceedings of IFIP 86, Sept. 1986.

[18] I. Jacobson, Language Support for Changeable Large Real Time Systems, OOPSLA86, ACM, Special Issue of Sigplan Notices, Vol. 21, No. 11, Nov. 1986.

[19] K. Nygaard, Program Development as a Social Activity, Proceedings of IFIP 86, Sept. 1986.

[20] T. Rentsch, Object Oriented Programming, SIGPLAN Notices, Vol. 17, No. 9, 1982.

[21] D. T. Ross, Structured Analysis (SA): A Language for Communicating Ideas, IEEE Trans. Software Engineering, Vol. SE-3, No.1, Jan 1977.

[22] D. Teichroew & E. Hersey III, PSL/PSA: A Computer-Aided Technique for Structured Documentation and Analysis of Information Processing Systems, IEEE Trans. on Software Engineering, Jan 1979.

[23] P. Wegner, How should Systems be Specified?, Proceedings of IFIP 86, Sept. 1986.

[24] G. M. Weinberg, The Psychology of Computer Programming, Van Nostrand Reinhold, New York, 1971.

[25] The Xerox Research Learning Group, The Smalltalk-80 System, Byte, Aug 1981.

[26] J.J. van Griethuysen (editor), Concepts and Terminology for the Conceptual Schema and the Information Base, ISO TC97/SC5/WG3, March 1982.

™ ObjectOry is a trademark of Objective Systems SF AB, Sjöängsv. 5, 19172 Sollentuna, Sweden. Smalltalk-80 is a trademark of Xerox Corporation. Objective-C is a trademark of Productivity Products International, Inc. Ada is a trademark of the US Government.