TRANSFORMATION OF DATA FLOW ANALYSIS MODELS TO OBJECT ORIENTED DESIGN

Bruno Alabiso

Microsoft Corporation
16011 NE 36th Way
Redmond, WA 98073-9717

ABSTRACT

This paper describes a strategy to transform Data Flow Analysis into Object Oriented Design. This transformation is performed by extracting information from the Data Flow Model, by enriching with Design decision and by finally producing an Object Oriented Design Model. Semiformal transformation rules are described. Also a special notation is introduced to describe the Object Oriented Design Model. The Model used to represent Data Flow Analysis is the one originally proposed by Yourdon, complemented with Ward-Mellor's Real Time extensions (the "Essential Model").

1. INTRODUCTION AND OVERVIEW

In the last few years the software community has witnessed the appearance of a multitude of software development methodologies. Almost simultaneously, products have appeared on the market to support one or more of these methodologies.

Development methodologies address several phases of the development life cycle, with emphasis ranging from requirements specification to system testing and maintainance. The most popular "early-phases" methodologies (also nicknamed "upperCASE" methodologies) are all derivatives of the one originally proposed by Yourdon-De Marco [DEM78] [YC79]: Structured Analysis and Structured Design (SASD), more recently enriched by extensions to support the construction of Real Time Systems by Ward-Mellor [MJ86] and Hatley [HAT86].
According to the Yourdon methodology and its derivatives, the construction of software systems must be preceded by two phases:

1. *Analysis:* during this phase the question: "What is the system supposed to do?" should be answered. The emphasis on what rather than on how implies that Analysis is an activity which goes hand in hand with that of stating the requirements of a system.

2. *Design:* during this phase the question: "How does the system do what is stated in the Analysis?" should be answered. For a software system this is equivalent to exercising strategies to isolate and define precise software components and their interrelations (Modules, Functions, Packages, Objects, or whatever, depending on the particular Design Methodology being adopted).

Both phases are supported by a variety of models which provide essential expressive power to the general precepts of the methodologies.

The most widely used model for Analysis is the *Data Flow Model*, which describes the system in terms of so called *Data Flow Diagrams* or *DFD's*. An example of Data Flow Diagram is given in figure 1.

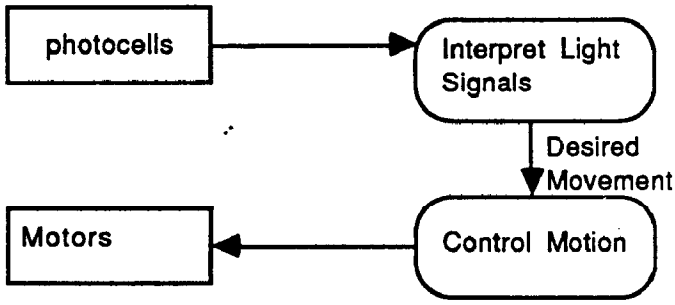


figure1 - Simple DFD

The DFD's are excellent at describing the flow of data to and from loci of functionality (the *Data Processes*, normally shown in DFD's as circles, or rectangles with rounded corners), but they are not very effective at expressing control to be exercised on the execution of functions. Ward-Mellor's and Hatley's methods have remedied to this by extending the basic DFD model with concepts to describe "control". The Model used in this paper is the one proposed by Ward-Mellor (the so called "Essential Model"). The "Essential Model" adds to the expressive power of DFD's by introducing the concept of *Control Process* (as opposed to standard DFD's Data Processes) and *Control Flow* (as opposed to Data Flow). An example of Essential Model is given in figure 2. Control flows (normally shown as dashed lines) carry events which may cause the activation of functions. These events can be generates internally or externally (interrupts are an example of externally generated events). Control Processes are de facto Finite State Machines which exclusively process events.

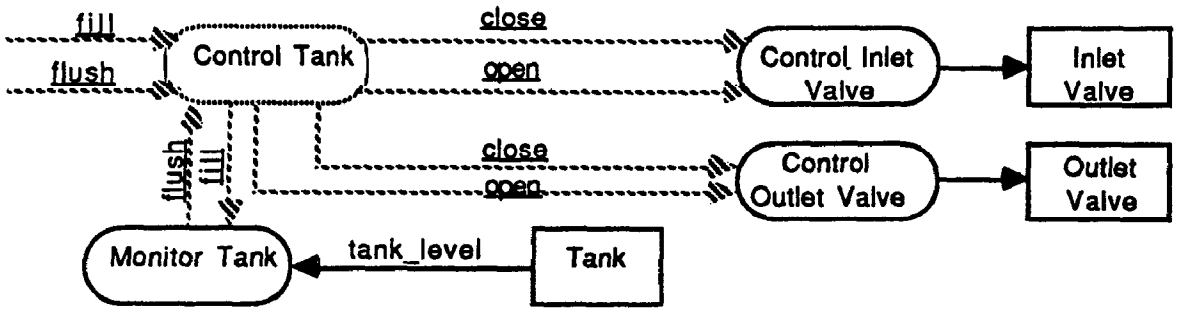


figure 2 - Ward Mellor's Essential Model

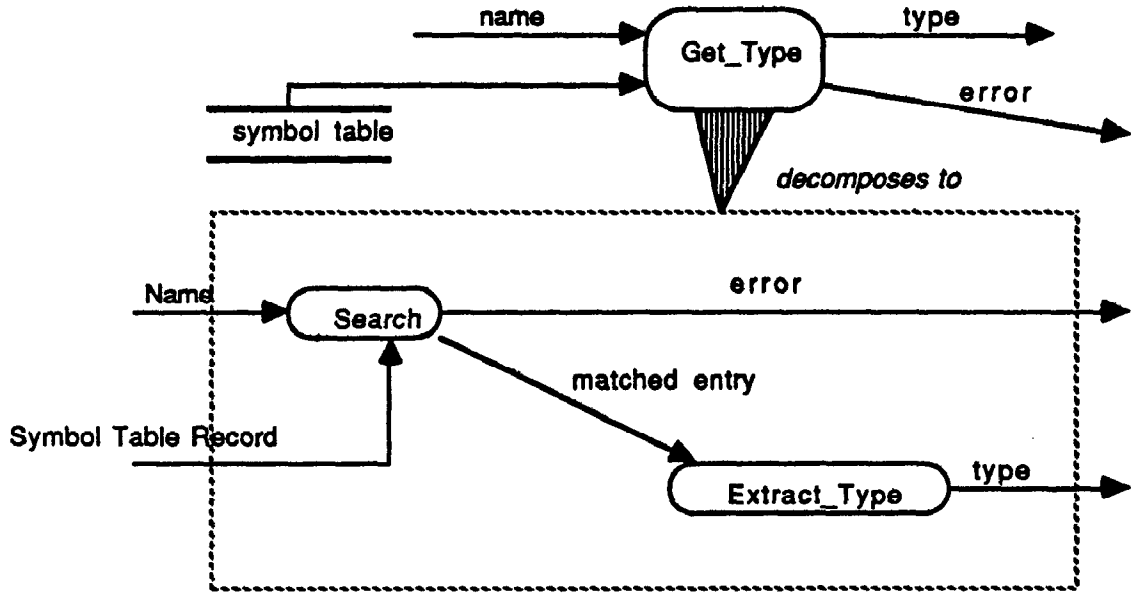


figure 3- Decomposition of Data Processes

A Data Process may be decomposed by describing it in terms of a lower level DFD (see figure 3) or by providing semi-formal specifications ("mini specs").

Control Processes are described in terms of so called *State Transition Diagrams* (see figure 4).

Whereas Analysis Models dwell in the problem state (i.e. they state the problem), Design Models live in the solution space. For this reason the flavor of a Design Model is heavily dependent on the conceptual schema chosen to represent the "solution". In the case where we wish to express our solution in terms of the familiar concepts of Classes, Objects, Methods, Inheritance etc., we talk about *Object Oriented Design.*

There is very little merit in completing the formal Analysis of a system if there isn't a path which will lead us from Analysis to Design. Page-Jones [PJ80] describes in his book a method to migrate from Structured Analysis to Structured Design (a form of functional decomposition Design). In this paper we propose a methodology to transform an Analysis Model into an Object Oriented Model. Before discussing this transformation, we will briefly outline the Object Oriented Model and we will introduce a notation to express it.

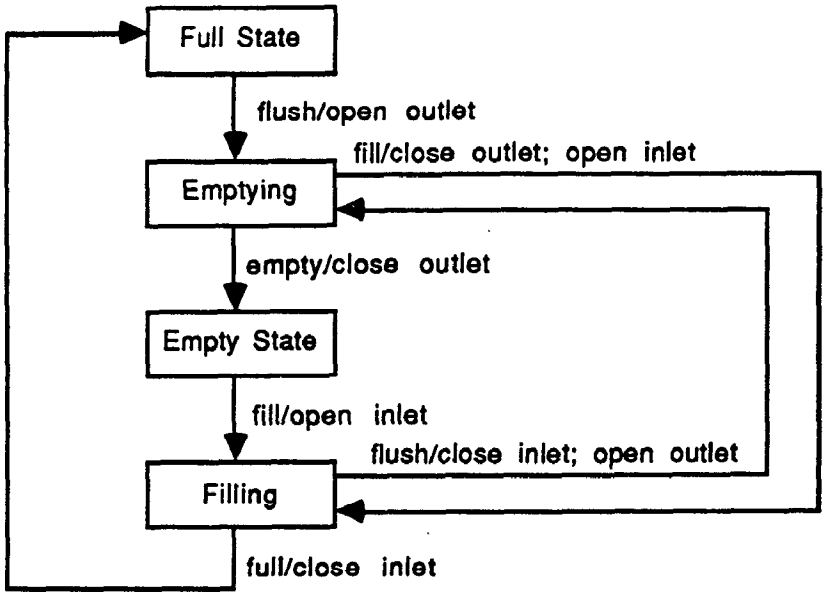


Figure 4 - State Transition Diagram for *Control Tank*

2. THE OBJECT ORIENTED DESIGN MODEL

The Object Oriented Design Model used in this paper is based on the Smalltalk-80™ [GR80] model, with two fundamental differences:

1. Objects in our model are strongly typed.
2. We do away with the concept of Smalltalk Processes. Instead we introduce the concept of *Active Objects* (see also Actors [AGH86]). An Active Object may answer messages like any othe Object, but it also features asynchronous behaviour (i.e. independent execution thread). This special asynchronous behavior of an Active Object may be assimilated to the execution (on a separate thread) of an Instance Method which starts when the Object is instantiated and never "returns"[1]. We will refer to this pseudo-Method by calling the *Executive* of an Active Object[2].

For the physical description of the Design Model we shall use two forms of design charts:

1. The *Functional Design Chart* or *FDC*.
2. The *Object Structure Chart* or *OSC*.

The first type of chart (FDC) is used to express and break down the functional behavior of Objects, i.e. the make up of Methos. The second type (OSC) is used to express and break down the "data" structure of Objects, i.e. the make up of Objects in terms of their Class and Instance Variables and Class inheritance. A FDS is shown in figure 5.

Control Lines (shown by dashed lines) partially or totally order *Actions*. An Action is, for instance, to send a message to an Object.

Object Lines represent Object Stores. For more clarity, a rectangle with an enclosed Object Name may be attached to an Object Line: this is normally used to highlight the receiver of a message.

Note that FDC's can also be used to describe the behavior of more than one Method for a given Class: in this case we talk about Class FDC's. Naturally a Class FDC can always be reduced to a group of Method FDC's (one for each Method of that Class).

An Example of OSC is given in figure 7.

[1] It may, however, terminate.
[2] In our Model Active Objects play the role of Smalltalk Processes.

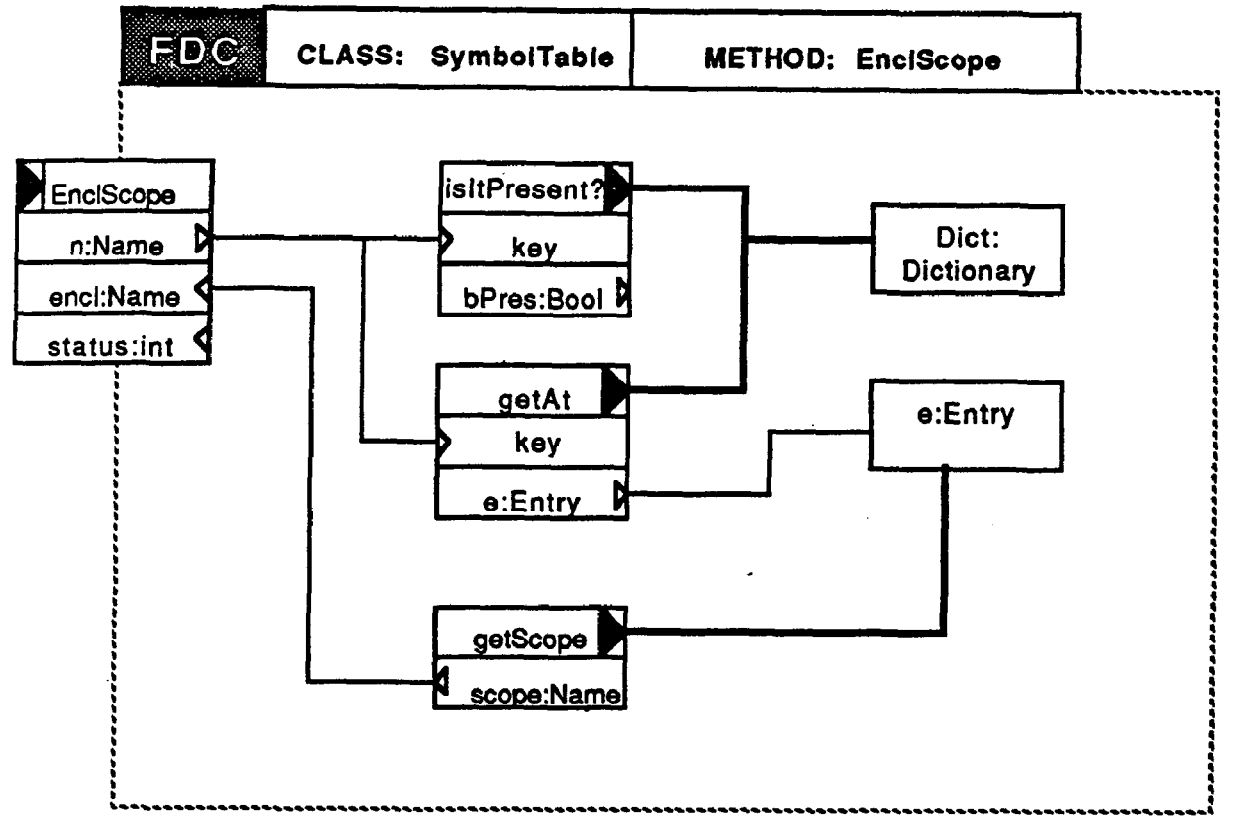


figure 5 - Functional Design Chart

Notice (some of) the conventions used in figure 6.

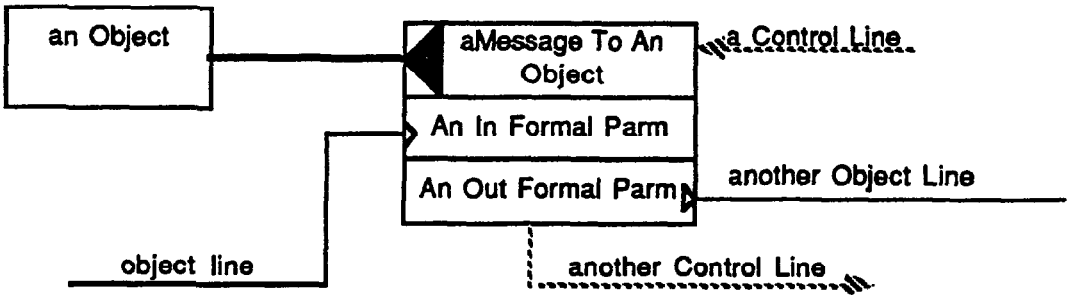


figure 6 - some FDC conventions

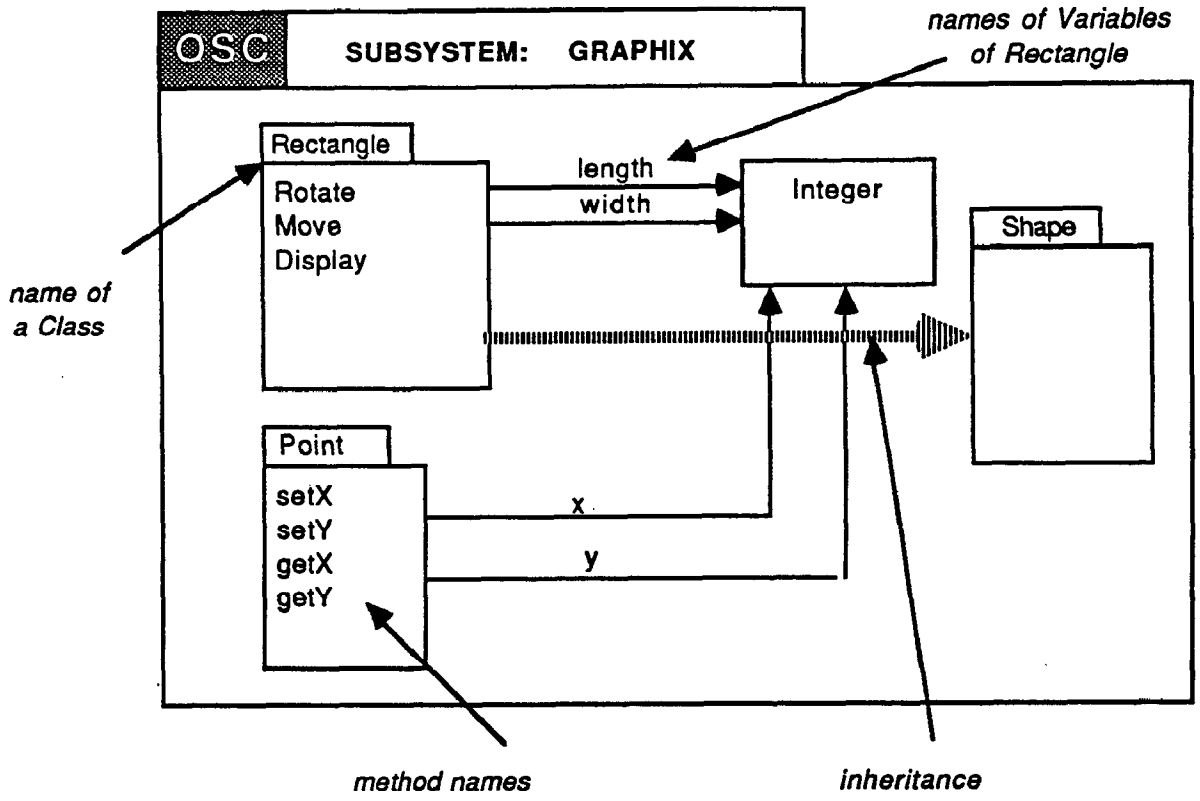


figure 7 - Object Structure Chart

3. THE TRANSFORMATION

We will now describe a strategy to migrate from the decomposition of a system functionality performed according to the methods of Structured Analysis to the design of the same system according to Object Oriented techniques.

This "migration" is truly a transformation between two Models of the same system. We will call this transformation: **Tad** (Transformation from Analysis to Design) (see figure 8).

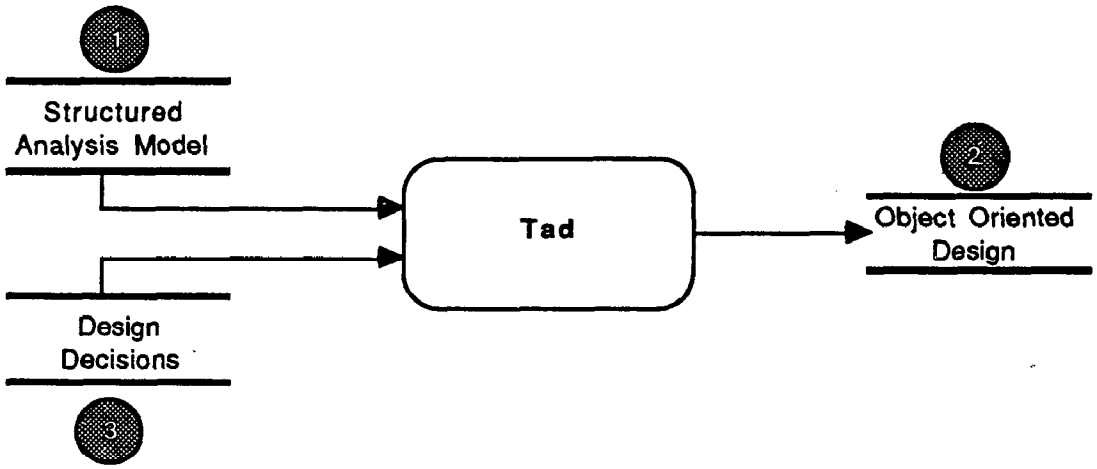


figure 8 - The Analysis to Object Oriented Design Transformation (Tad)

Note the following facts:

- The transformation **Tad** is not automatic (of course): many design-time facts are not expressed during the phase of analysis. Human intervention (3) is necessary to manipulate the input Mode (1) and transform it into the design Model (2).

• Model (1) is the Structured Analysis Model based on Ward-Mellor's extensions of the original Yourdon methodology.

• Model (2) is the Object Oriented Model briefly presented in the previous sections.

3.1. FUNCTIONAL DECOMPOSITION VERSUS OBJECT DECOMPOSITION

The advent and increasing popularity of Object Oriented Design/Programming disciplines advocate Object Decomposition as the primary technique to be used when designing software. According to this technique, the types (Classes) of various Objects are identified, the Methods (operations) listed and referenced or sub-Object components are identified. The process continues recursively until we are left with either very simple Classes (perhaps those which map directly onto types of the programming language[1]) or with Classes which have already been designed/programmed.

In practice, the only flaw with the above technique is that it does not take into consideration the fact that, in order to specify the list of Methods of Objects of a given Class, it is necessary to know who are the client-Methods, and what are their needs: it is the clients who ultimately define which operations need to be supported for the Objects of a given Class. In other words it is still necessary to perform Functional Decomposition.

We see Functional Decomposition as a process parallel to that of Object Decomposition. More strongly we say that neither should take the precedence over the other: the designer should feel free to hop between the two techniques at will.

By nature, Data Flow Diagrams decompose functions. Other common practice analysis techniques, such as dictionary data definitions, provide adequate abstractions to aid Object Decomposition.

3.2. FUNDAMENTAL TRANSFORMATIONS

The Structured Analysis Model is constructed by:

1. Defining "what" the system ought to do, in terms of a hierarchy of Data/Control Flow Diagrams. In this hierarchy lower level Flow Diagrams express the function of a Data Process at the level immediately above.

2. Defining Finite State Machines governing the firing of Data or Control Processes (Ward-Mellor's *State Transition Diagrams*).

3. Defining Data and Decomposing Data.

So, in order to develop a strategy for **Tad**, it is necessary to:

1. Interpret Data, Data Processes, Data Stores and Terminals in terms of Object Oriented concepts.

2. Interpret the DFD hierarchy in terms of Design Decomposition: it is intuitively obvious that the DFD hierarchy and the decomposition of a complex design into design components must bear some relationship to one another.

3. Interpret the significance of Control Processes for the Design Model. Express Events and ordering of resulting Actions in the Design , as defined in the State Transition Diagrams.

4. Use Data Decomposition information to help define Object Decomposition.

3.2.1. DATA AND OBJECTS

In an Object Oriented Design Data is Objects and Objects is Data. It makes no sense to differentiate the two. So it is obvious to map Data defined in the Analysis Model to Objects in the Design Model.

Data Flow lines can be dubbed as *object flow lines* and data names entered in the Data Dictionary, truly correspond to Object names.

3.2.2. DATA PROCESSES

A Data Process in a DFD reads Input Data, transforms it and produces Output Data. It seems at first obvious to map a Data Process to the Method of an Object, and to interpret the activation of a Process as the sending of a Message[2] (figure 9).

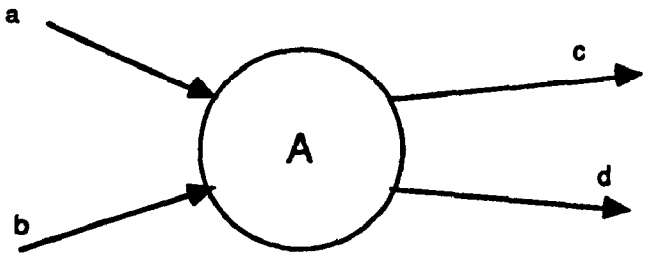


figure 9 - A Data Process

There are two obvious choices for the destination of the message:

1. The destination could be one of the input data (Objects), i.e. **a** or **b**. Often the same object is also present on an output line, disguised under a different name. For instance **a** could be **fileX** and **c** could be **updatedFileX**: the clear choice in this

[1] We are not considering here "pure" object oriented languages like Smalltalk-80™, but rather hybrid languages like Objective-C™ or C++.

[2] Naturally the Data Process may be mapped to the asynchronous behavior of some Active Object. This would generate the Design of a concurrent system. For simplicity we shall not discuss design of concurrent applications in this paper.

case is to make **fileX** the receiver and to eliminate lines **fileX** and **updatedFileX**.

2. The destination could be an Object **x** not described by any of the inputs which we wish to associate with Method **A**.

The choice of the destination will naturally bring us to select the remaining input/output data as candidates for becoming input/output parameters of the Methos **A** (we are assuming I/O cohesiveness here, i.e. that the process **A** reads exactly one single piece of input data for each piece of output data generated. Read further in this paper for a discussion on I/O uncohesiveness).

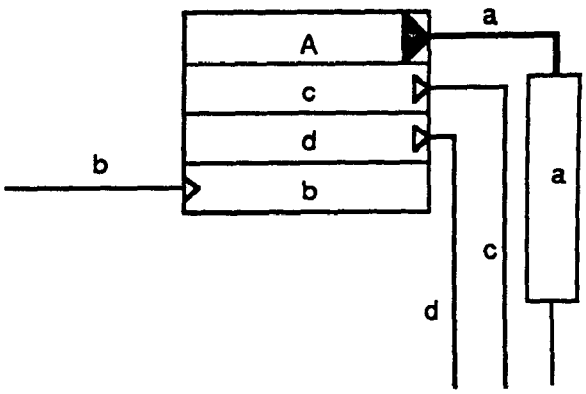
Figure 10 shows how cases 1. and 2. above are expressed in FDC formalism.

Figure 11 shows an example of case 1.

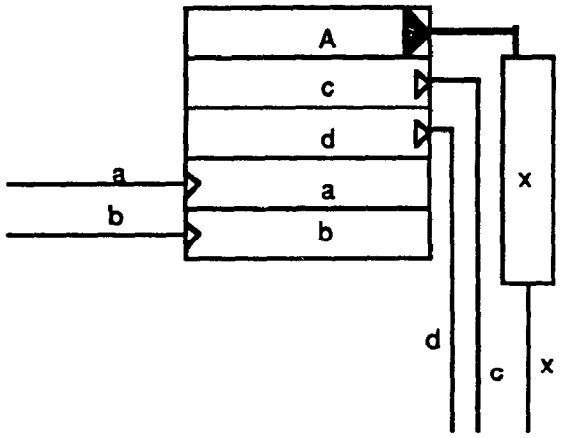
It can be seen (from figure 11) that the Method **POP** is associated with the Input Object (Data item) **myStack**.

As an example of Case 2 consider the following **Tad** transformation (figure 12).

In the example above a new Object (the Class **VectorMaths**) has been introduced to become the destination of the Message **CalculateProjections**. Naturally the designer might have chosen **Vector** to be the destination of the Message like in Case 1[1].



CASE 1



CASE 2

figure 10 - Data Process expressed in FDC formalism

3.2.3. TERMINALS

Terminals in the Data Flow Model represent entities which exist in the world outside the Application System and which produce or consume data (terminals are also called, for this reason, *sources* and *sinks*). The act of extracting data from a source and that of delivering data to a sink are always associated, in the Object Oriented world, with precise Method invocations.

For example, let's consider the DFD shown in figure 13.

An obvious observation is that all terminals can be safely mapped to Objects. Data (Objects) from sources must be read by sending appropriate messages to the sources. Data is delivered to sinks by sending appropriate messages to the sinks themselves.

[1] Note that **CalculateProjection** is a Class Method of **VectorMaths**, whereas the same would be an Instance Method of **Vector**. **Tad** does not provide any hard rules to decide whether operations are to be implemented as Class or Instance Methods: this kind of decisions belong solely to the design domain. Also note that **VectorMaths** resembles closely a *package* in the ADA sense.

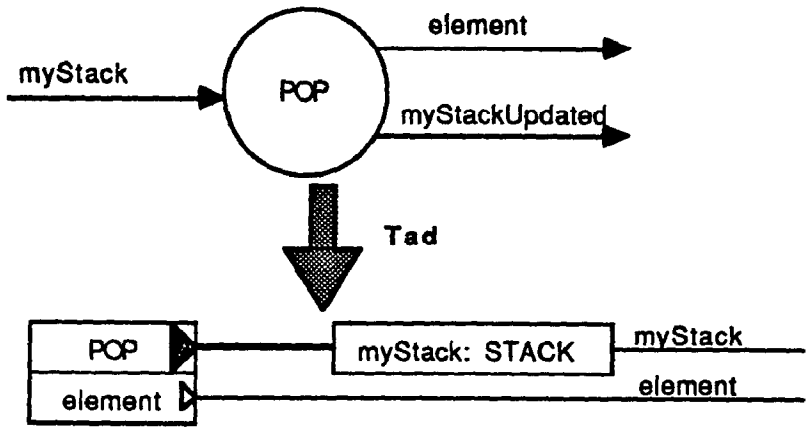


figure 11 - Transformation of process POP

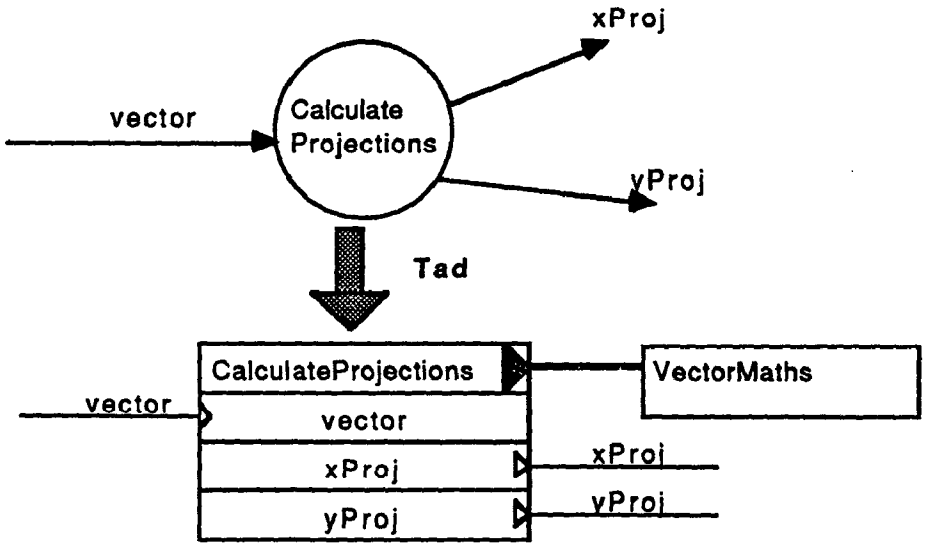


figure 12 - Transformation of CalculateProjections

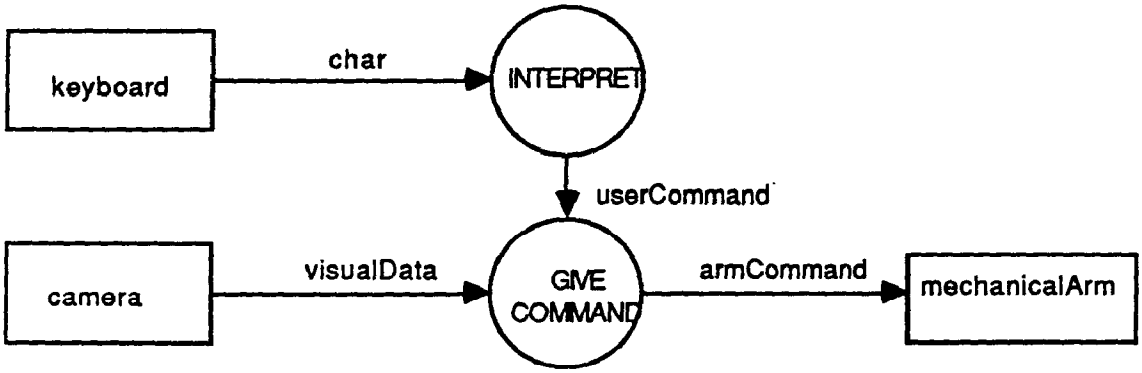


figure 13 - a DFD with terminals

These messages may be directly derived from the Data Processes specified in the Data Flow Diagrams, or new messages can be introduced. In the case of the **keyboard** for instance, the process **INTERPRET** can be **Tad**-transformed into a Method of **keyboard** itself: in this case the line **char** would disappear (figure 14).

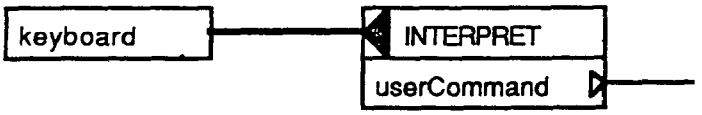


figure 14

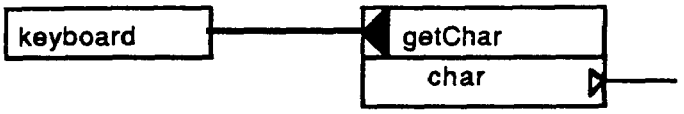


figure 15

Another possible transformation is to introduce a Method which extracts data from the source. In figure 15 the Method **getChar** is introduced for the destination **keyboard**.

Similar strategies can be applied to sinks.

3.2.4. DATA STORES

Data Stores are naturally mapped to Objects in the O^2 Design world. As for the case of Terminals, Methods may be added to extract/place data from/into a data store (see figure 16).

In case the data store is identified as the destination of a Message (Tad-transformed from a Process reading from or writing to the Data Store itself), the **Tad** transformation is simplified as shown in figure 17.

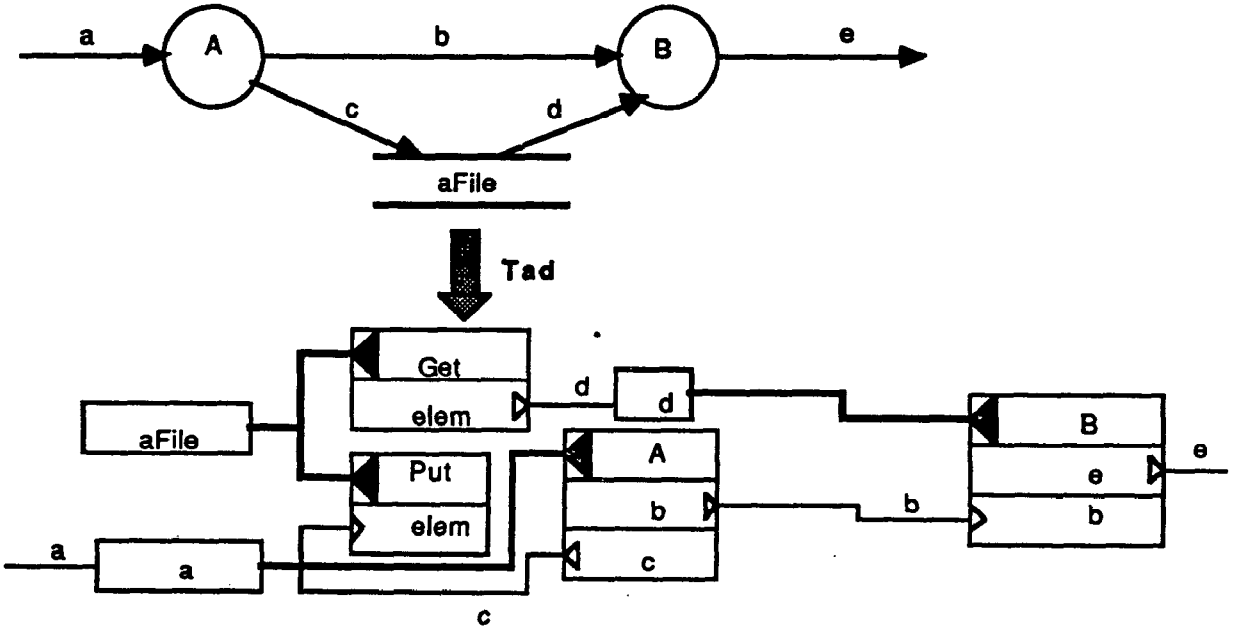


figure 16 - transformation of a Data Store

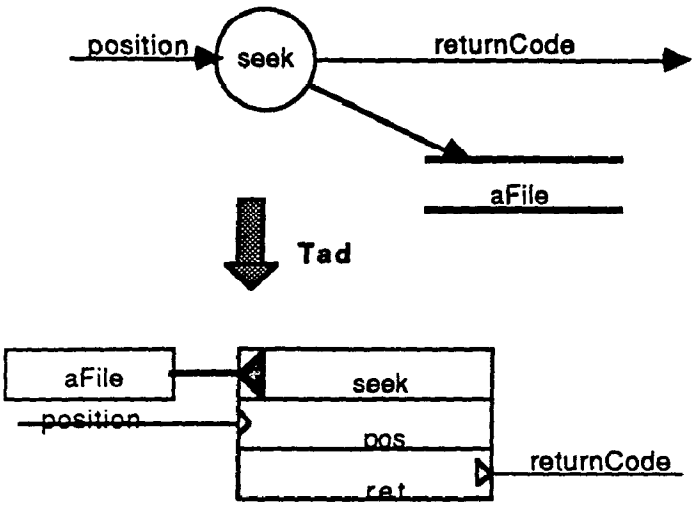


figure 17

Finally a Process which writes to two or more Data Stores may be Tad-mapped to two or more messages as shown in figure 18.

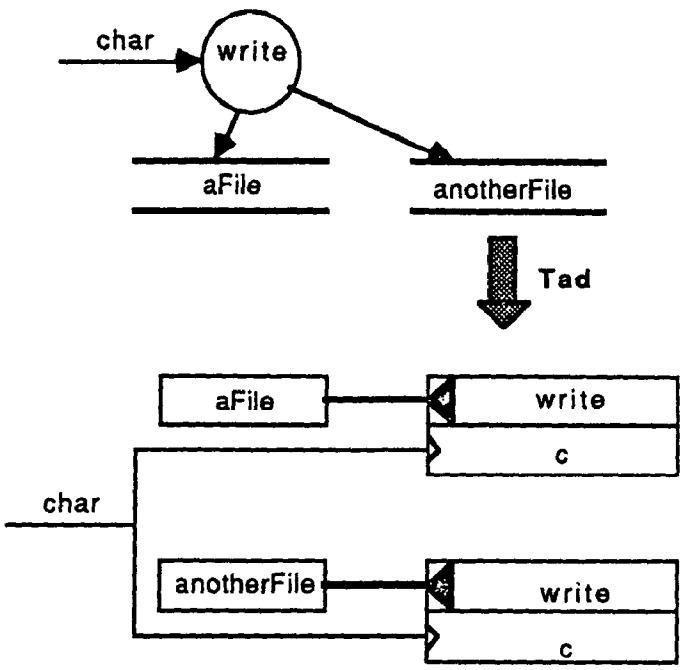


figure 18

3.2.5. THE DATA FLOW DIAGRAM

One of the main benefits of the Data Flow Methodology (DFM) used for the Analysis of application systems is that it allows us to apply a divide and conquer strategy to functionally decompose the complexity of a system. We have seen in the previous sections that it is possible to convert DFD fragments into FDC fragments. This allows us, in principle, to adopt parallel methodologies for functional decomposition in the DFM and functional decomposition in the Object Oriented Design. The example in figure 19 illustrates this.

If we assume I/O cohesiveness for all Processes (see section 3.3), the DFD's in figure 19 are transformed to the FDC's in figures 20 and 21.

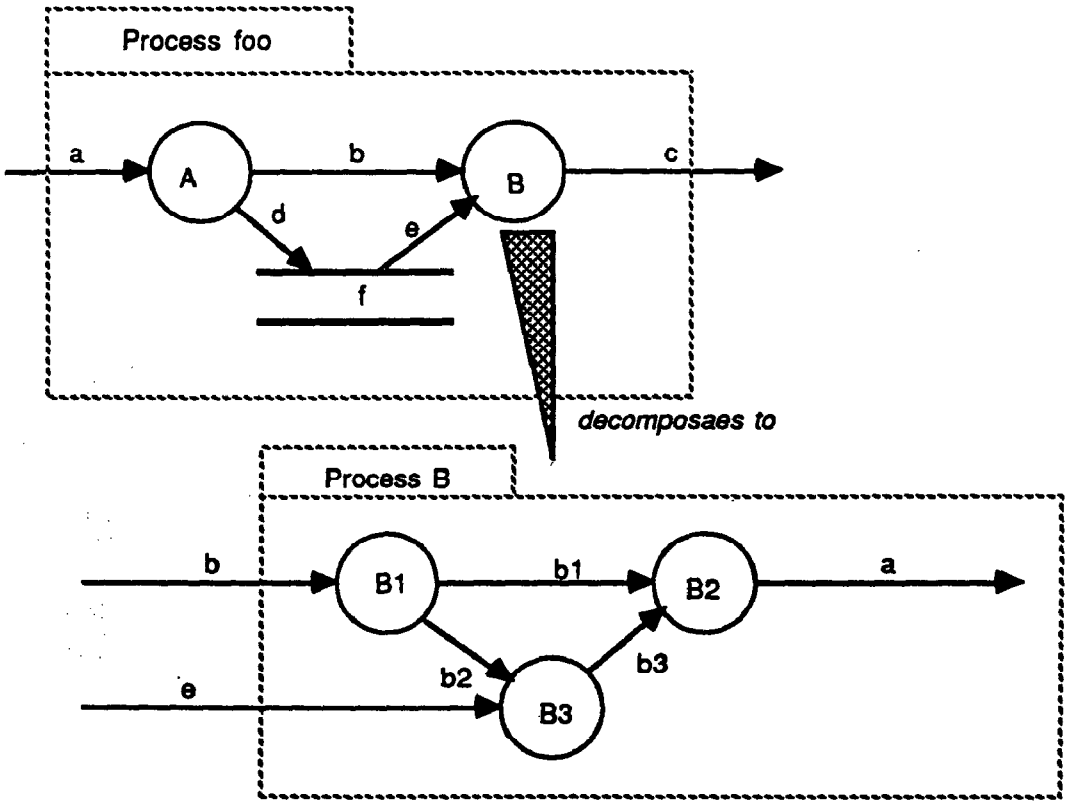


figure 19

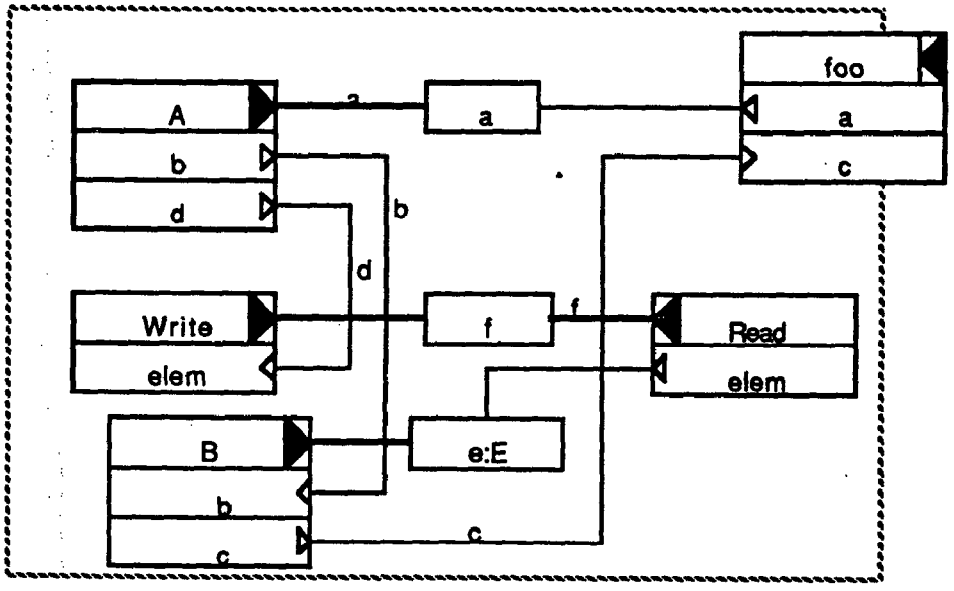


figure 20 - FDC for Method foo, Class X.

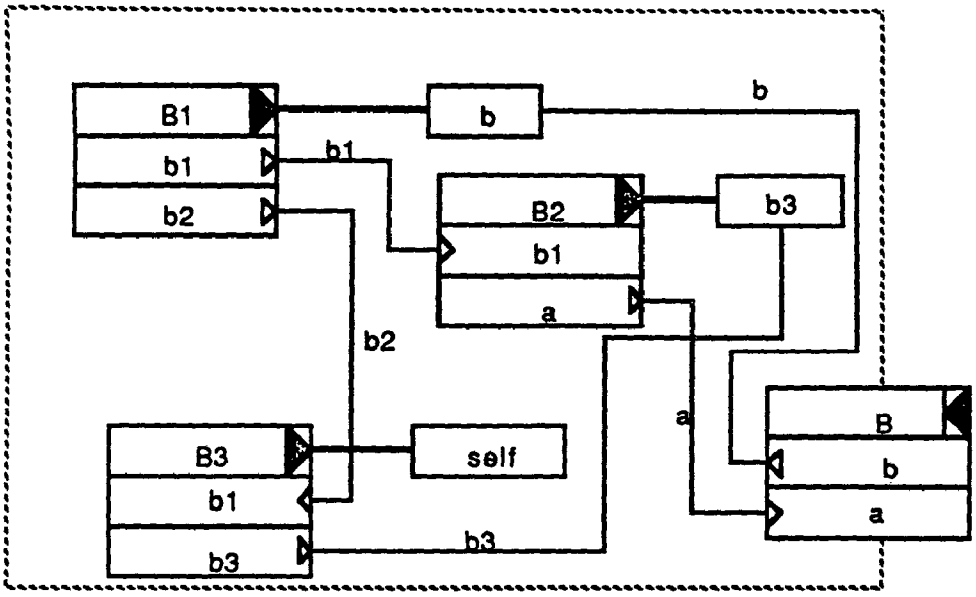


figure 21 - FDC for Method B of Class eClass, the Class to which e belongs

3.3. I/O UNCOHESIVENESS: THE BURIAL METHOD

It would seem from the above example that functional DFD decomposition is always parallel and isomorphic to functional Method decomposition. Unfortunately this is true only for a very limited class of DFD's.

When performing a transformation from a Process Activation to a Method Invocation (message), we may face a problem if the Process is consuming several pieces of input data before generating output data, and/or if the Process generates pieces of output data independently with respect with all other inputs and outputs. We call this potential problem of DFM Data Processes *I/O uncohesiveness*. In case a Process suffers from I/O uncohesiveness, mapping it to a Method invocation is not a straight forward operation, because the Mechanism of Method invocation (message) assumes I/O cohesiveness[1]. For example, take the DFM fragment shown in figure 22.

In this case the Data Process **MAKE_WORD** may chew up several characters (**char**) before generating a **word**.

Obviously one cannot make **MAKE_WORD** a Method of **char**, since it would not be able to return a **word** every time it is invoked. The same problem arises if one selects **MAKE_WORD** to be a Method of some other Object.

Note that the problem is not only inherent to the mapping of DFM's to Object Oriented Design, but in general to the mapping of DFM's (which are intrinsically concurrent) to a "procedural", sequential view of the world.

One choice could be to decide that **MAKE_WORD** is going to be mapped to a task (an Active Object). Naturally this is not always desirable: we shouldn't be forced to make such decision from an inherent property of Data Flow Analysis Models.

In can be however observed that **MAKE_WORD** can be likened to a Method invocation if char were not an input parameter, but rather some private variable directly "buried" in the Method itself, or indirectly "buried" in some other Method which **MAKE_WORD** directly or indirectly invokes.

This suggest that a manipulation of the original DFD can be performed by burying a part of the DFD itself within **MAKE_WORD** in order to render the process **MAKE_WORD** itself I/O cohesive.

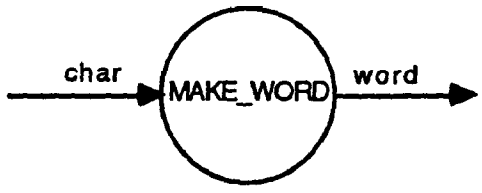


figure 22 - DFD fragment

[1] The Process communicates *asynchronously*, whereas a message is a time-indivisible entity.

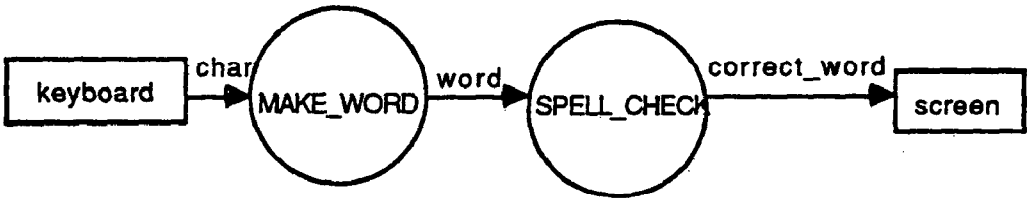


figure 23 - DFD with I/O uncohesiveness

In the DFD in figure 23 the process **MAKE_WORD** is not I/O cohesive. The "burial" technique consists of pushing down that part of the diagram which feeds multiple inputs to **MAKE_WORD**: in this case the terminal **keyboard** and the data flow line **char** are buried within **MAKE_WORD** before attempting to transform (figures 24-25). This is equivalent to stating that the implementation of the operation **MAKE_WORD** will be responsible for dealing with **chars** coming from the **keyboard**.

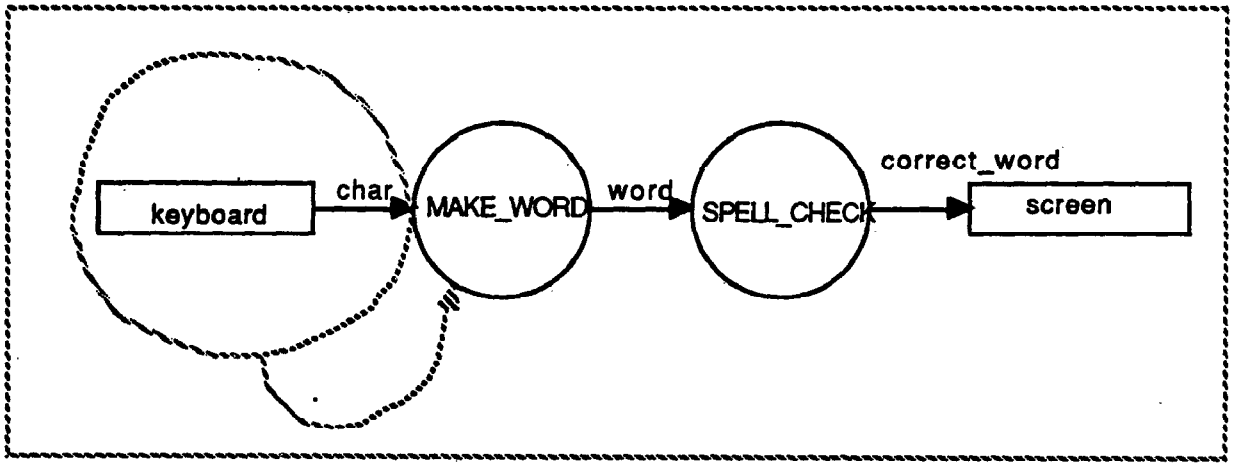


figure 24 - burial of keyboard

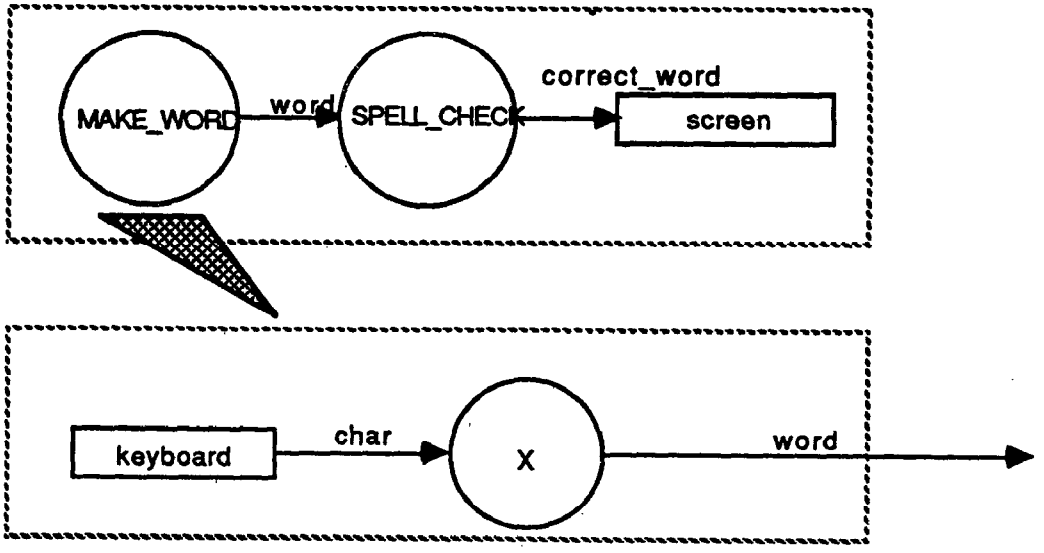


figure 25 - after burial

The question is now: what is the entity **X**? There are two possible answers:

1. If **MAKE_WORD** was originally decomposed into a DFD, then **X** represents that DFD. Note that, by the rules defined for burial, **X** is always balanced with respect to the original DFD.

2. If **MAKE_WORD** was not originally decomposed, then **X** is undefined. In this case we adopt the convention to represent **X**'s input and output data flow lines as shown in figure 26.

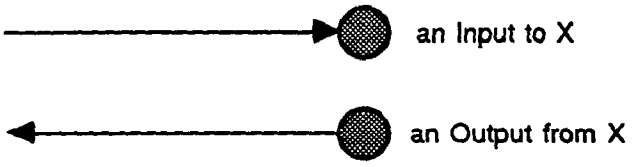


figure 26 - terminators

So, if 2. holds, the DFD of **MAKE_WORD** becomes that shown in figure 27.

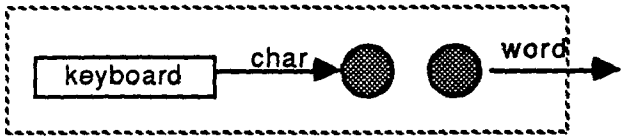


figure 27 - Modified DFD of MAKE_WORD after burial and decomposition

The grayed circle (which we shall call *terminator*) indicates that "something is missing" in the specifications. In the above example it is not specified how **char** will be processed. Equally it is not specified how **word** will be produced.

Note also that by applying the burial method we have allowed certain processes to be pure sources (no input) or pure consumers (no output): this only indicates that the complete I/O of those processes is hidden within the process specifications. In other words, burial is a step towards abstraction and encapsulation.

In any case, after burial it is possible to transform in the usual manner. The terminators will still be present in the FDC's which result from the transformation: the designer may choose at this point to eliminate them by providing further detail in the Object Oriented Design.

The burial method presents other nuances which are not described here for brevity. The most important thing to realize about this method is that it allows to transform functional decomposition expressed by a Data Flow Model to functional decomposition expressed by an Object Oriented Model.

3.4. FROM DATA DICTIONARY TO OBJECT STRUCTURE CHART

During the Analysis phase the Data Dictionary is populated with entries describing the decomposition of data elements. Data identified during the Analysis correspond to Object identified during the Object-Oriented Design phase. Decomposition in the Object Oriented world means to discover what Objects an Object is made of or what Objects it references. Or, if you want, decomposition means to uncover client-server relationships.

The OSC is used to describe both the client-server relationship and the inheritance relationship. So OSC's will be the product of transforming Data into Objects.

The following observations apply:

1. If a Data entry specifies a sequence: **a** is composed of **a1+a2+...+an**, this implies that the Class **a** must feature Variables **a1, a2,...,an.**

2. If a Data entry specifies repetition: **a** is composed of 1 to n **ax**'s, this means that, in the Object Oriented Design world, **a** must contain a **Set, List, Array**, or some other Collection which holds together Objects of the Class **ax.**

3. If a Data entry specifies selection: **a** is composed of either **a1** or **a2** or.... or **an**, this may translate into an inheritance scheme, in the sense that, for instance, **a1, a2,..., an** could be subclasses of a common ancestor-Class **ac.** Then **a** could contain an Object of Class **ac**, or **a** may

In all cases it is quite straight forward to transform Data Dictionary Entries in a set of consistent Object Modelling Charts. The examples in figure 28 illustrates this.

Naturally the resulting OSC's will still lack essential information after the transformation. Namely the lists of operations (Methods) featured by each Object cannot be derived from the Data Dictionary: these lists must be compiled by observing the places in the design (FDC's) where the objects in question play a role. Also more information about decomposition of Objects (i.e. determination of private Variables) may be obtained by observing places in the DFD's where Data is grouped/ungrouped.

Unfortunately Structured Analysis does not help much with inheritance (with the exception of very limited cases, like the one shown in figure 29): the process of organizing Classes of Objects in a hierarchical fashion is truly a design-time task.

3.5. TRANSFORMATION OF CONTROL

In the discussion of **tad** applied to control, let's consider separately:

[1]Depending on whether or not the names used for Data in the Analysis indicate *types* (Classes) or *variables* of a given data type (*instances*). This distinction is very often only loosely made in Analysis.

1. Data Processes featuring incoming/outgoing control lines.

2. Control Processes.

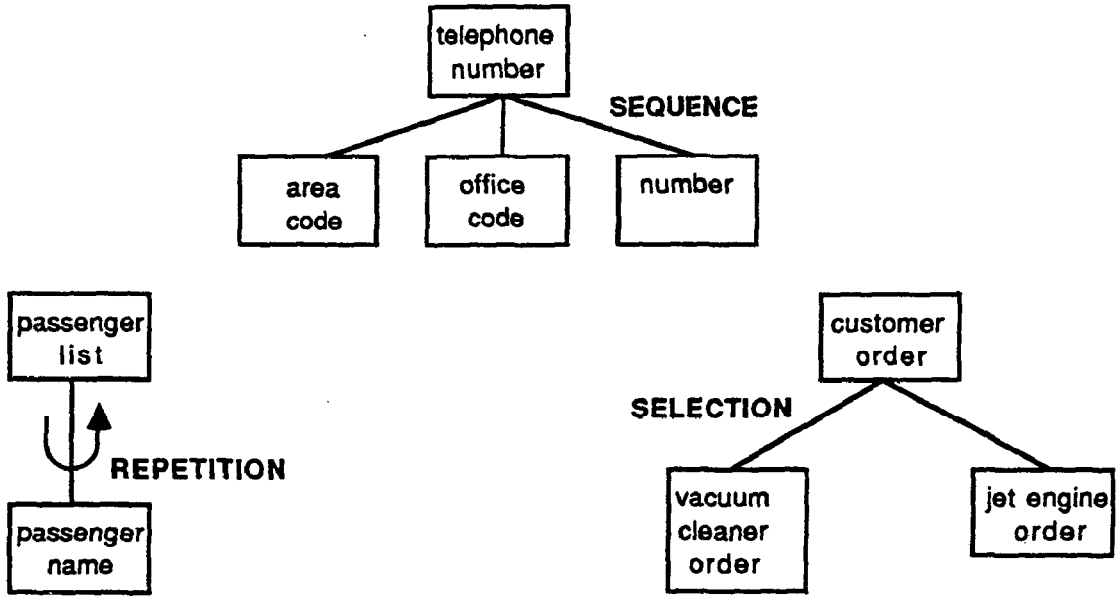


figure 28 - Data Decomposition[1]

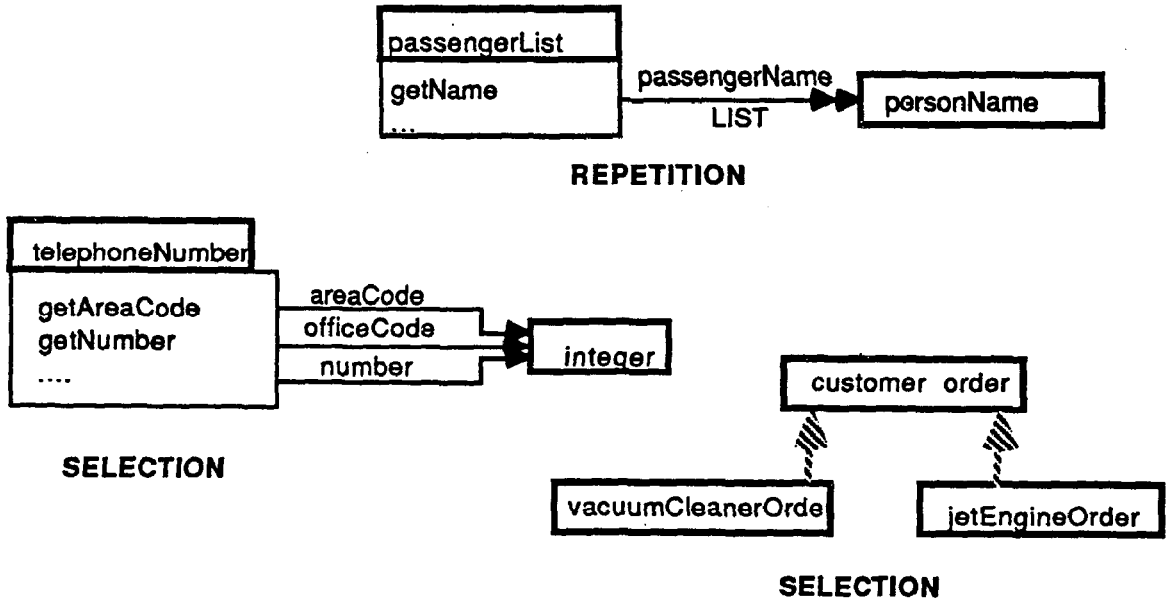


figure 29 - OSC from Data decomposition

[1]Taken form [PJ80].

3.5.1. DATA PROCESSES WITH CONTROL LINES

Let's consider an example (figure 30).

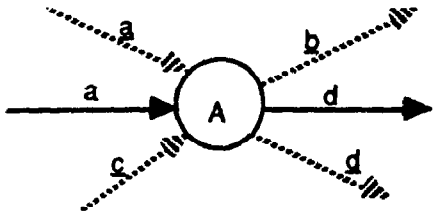


figure 30 - Data Process with control lines

Note that control lines are labeled, by convention, with underscored lower case characters.

It is obvious that the process **A** must "react" to the events **a** and **c**. This suggests that **a** and **c** could be Method invocations (Messages) requesting action. The destination of the Messages would be an Object **OA** which "implements" **A**. The Object **OA** could be that corresponding to one of the input data lines to **A** (a in the example above) or some other Object.

The implication of this mapping strategy is that the process **A** does not map to a single Methods Invocation but to multiple Method Invocations (on the same Object). This poses an immediate difficulty in the sense that it is easy to associate input/output data lines to input/output parameters of a Method. The question is: which Method? The answer is provided by decisions made by the Designer. In general each Method derived from an incoming control line may be associated with a set of parameters which correspond to all or part of the incoming/outgoing data lines of process **A**. Let's consider a concrete example (figure 31).

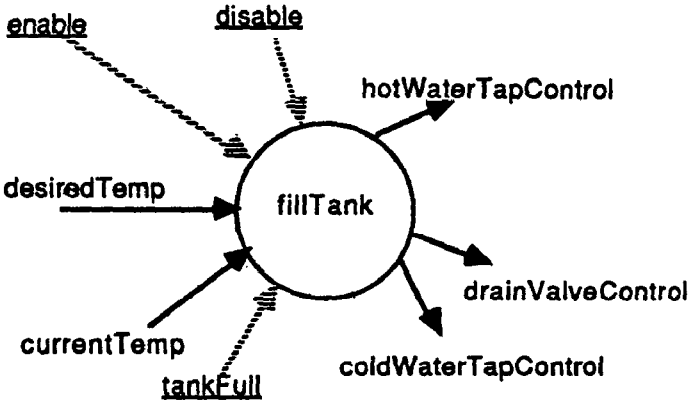


figure 31 - the Data Process fillTank

The process **fillTank** fills a tank with water and it keeps the temperature at **desiredTemp**. By applying **Tad** to the above DFD fragment, we decide to introduce one Active Object: **tankController** which enjoys three Methods: **enable, disable** and **tankFull**. The control lines **enable, disable** and **tankFull** become Method Invocations on the receiver **tankController** (see figure 32).

Clearly the FDC fragment in figure 32 is not complete. We need to add a Method to set **desiredTemp** (figure 33)

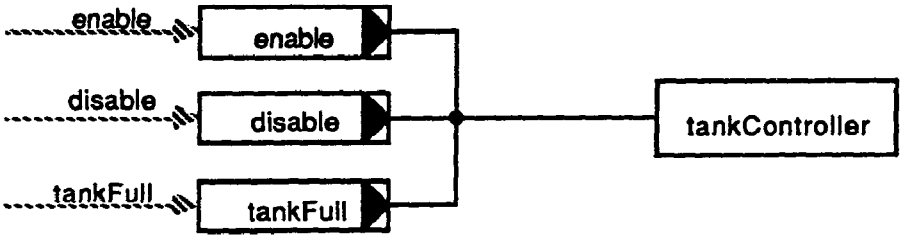


figure 32 - Tad Transformation of Data Process fillTank

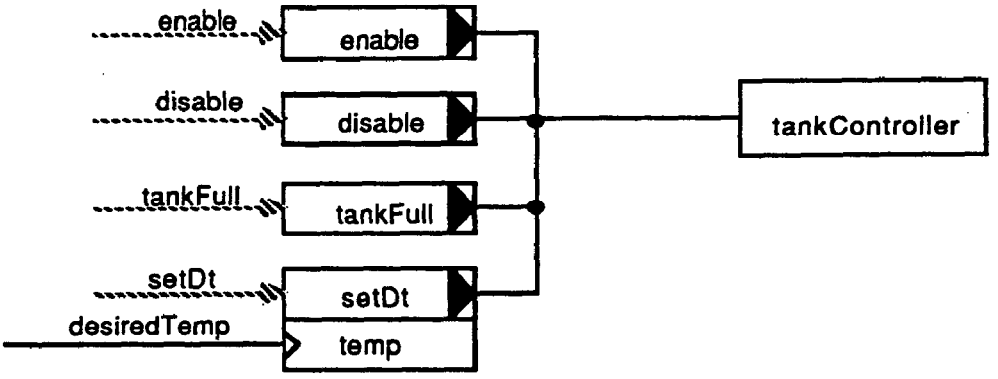


figure 33

The output Objects **hotWaterTapControl**, **drainControl** and **coldWaterTapControl** (presumably integers specifying voltage levels for the operation of electrical valves) are definitely I/O uncohesive with respect to any input data. Hence the Burial method must be applied to these Objects. It is evident from the example that the Objects in questions are produced by the Executive of **tankController**, therefore they will appear in the FDC of the Executive of **tankController** as parameter-values of Method Invocations performed by that Executive on other Objects in the system (like **inletValve**, **drainValve**, etc.).

Similar observations apply to **currentTemp**: this input line is also most likely I/O uncohesive[1].

In the above example no outgoing control lines were present. Output control lines from a Data Process are dealt with again by applying the Burial method. This is done to reflect the fact that FDC's model the functional decomposition of Methods, where control requiring further action is always passed from Client to Server. If we left the output control lines in the FDC which transforms the Data Process featuring those lines, we would be incorrectly showing the inner workings of an Object within the specifications of its Client: this clearly violates the basic principle of information hiding of Object Oriented Design.

3.5.2. CONTROL PROCESSES

A Control Process processes events and generates events. Its definition in the Ward-Mellor Methodology is given in terms of the specification of a Finite State Machine. The State Transition Diagram is used for this purpose.

Since a Control Process is a Finite State Machine, we may map it via **Tad** to an Object implementing the functionality of this Machine. Since this Machine controls all processes which reside in the same Data Flow Diagram, it is natural to bury all Objects which implement the functions of Processes external to the Control Process inside the Object implementing the Finite State Machine. Before we can do this, though, it is necessary to eliminate Control Lines which are both input to the Control Process and output from some Data Process. The Burial method is applied in both cases. Let's illustrate this with an example (figure 34).

The first task is to bury Σ in all processes which control Σ, i.e. **B** (which controls Σ through **b**) and **C** (which controls Σ through **e**). The diagram in figure 34 is transformed into the one shown in figure 35.

Note that, although Σ has been buried within **B** and **C** (changing them into **B'** and **C'**) to allow **B** and **C** to control Σ, Σ must still appear at this level to control process at the same level (**A** through **a**, **B'** through **d** and **C'** through **f**). This makes perfect sense in the Object Oriented world: Σ is an Object Server of both **B** and **C**, whilst both **B** and **C** are servers of Σ.

After the above transformation, **A**, **B** and **C** are buried within Σ to allow Σ to control them. The way Σ controls **A**, **B** and **C** is totally determined by the State Transition Diagram associated with Σ. A computer tool which supports the transformation **Tad** will be quite capable to generate the Design Methods **b** and **e** of Σ automatically.

Finally notice that the FDC's which are going to be used to map the decomposed DFD's of **B'** and **C'** above are Class FDC's, since more than one Method is specified (**c** and **d** for **B**, **f** and **g** for **C**).

[1] Reasonably the process **tankController** will read this value at random times, not prompted from the outside. Uncohesiveness and consequently need for burial would disappear if we chose to communicate **currentTemp** to **tankController** at times determined from outside, for instance at regular time intervals.

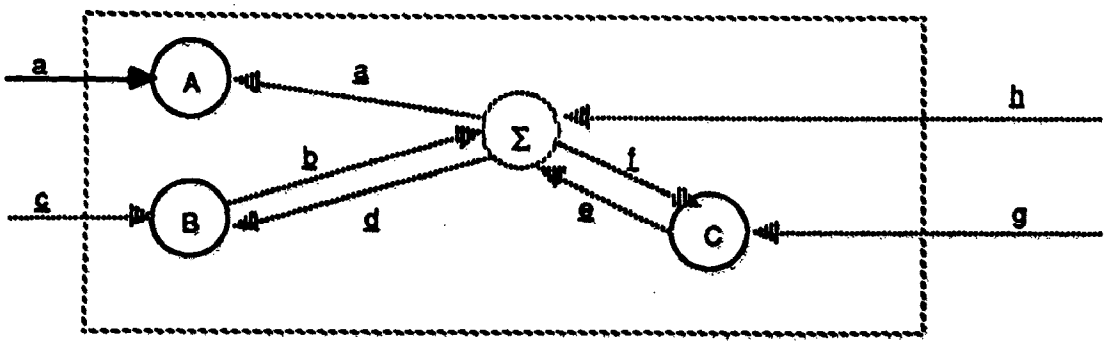


figure 34- a Control Process

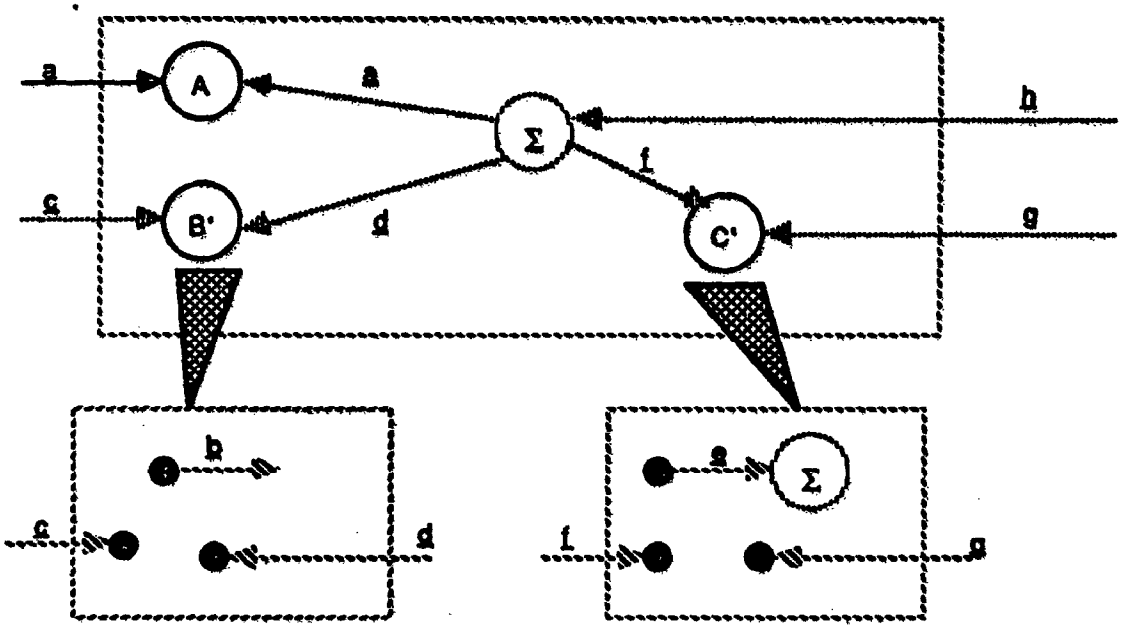


figure 35 - the Burial Method applied to Control Processes

4. SUMMARY

There has been much controversy about whether Structured Analysis techniques make sense at all in an Object Oriented context. The main objection to using SA techniques prior to an Object Oriented Design is that SA techniques ignore the existence of Objects, hence SA may "warp" the resulting Object Oriented design.

The approach outlined in this article shows that a great amount of reconciliation of the two techniques is possible. We also go so far as to say that this reconciliation is useful.

SA techniques have amply demonstrated their value in expressing the specifications of the functional requirements of a system. The major shortcoming of SA is that it requires a sizeable informal quantum jump to reach down from DFD's and Data Dictionaries to any form of software design. This paper hopes to show that it is possible to provide semiformal strategies to reach down from SA to Object Oriented Design.

Undoubtedly the transformation is less painful if the person who compiles the "Essential Model"[1] is "object aware": in this case it will be much easier to isolate and give form and behavior to Objects in the Design phase.

In facts if the Analysis Model itself were slightly modified to deal with "objects, classes and methods", rather than "data and processes", the transformation could be further simplified. In this case much less guesswork would be required to migrate from the Analysis Model to the Object Oriented Model.

We would like to conclude this paper with a philosophical note. We believe that, when it comes to the use of different Models for the description of a complex system, one should never be religious about the use of one Model or another: it is a typical human ability to be able to apply different abstraction Models to the same underlying "reality": the more numerous and the more descriptive the

[1] Ward-Mellor terminology.

Models, the better our understanding. As mentioned above, Analysis Models have well proven their value in pre-design activities, especially when large, complex application systems are involved. SA models are used before the design phase, therefore they should be useful whatever the chosen Design methodology happens to be.

More generally, the problem facing the analyst/designer/software engineer is not one of choice between different methodologies, but one of integration of different models, where each model is a very valuable "view" of the system from a preferred perspective (the program itself is one such model). We believe that the ideas presented in this paper are a step in the right direction and we hope that more effort be dedicated by methodologists and tool builders alike to favor and support multi-model integration.

5. ACKNOWLEDGEMENT

The author wishes to thank Nastec Corporation for making the research for this paper possible.

REFERENCES

[ALA87] Alabiso, B., *Object-Oriented Design,* The Case Report, Nastec Corp., September 1987 and October 1987.

[AGH86] Agha G.A., *Actors: A Model of Concurrent Computation in Distributed Systems,* the MIT Press, 1986.

[BES87] Besemer, Y., *An Introduction to Object Oriented Design and Programming,* Case Outlook, September 1987.

[BOO86] Booch, G., *Object Oriented Development,* IEEE Transactions on Software Engineering, February 1986.

[COX86] Cox, B., *Object-Oriented Programming, an Evolutionary Approach,* Addison Wesley, 1986.

[DEM78] De Marco, T., *Structured Analysis and System Specification,* Yourdon Press, New York 1978.

[DOD83] US Department of Defense, *Reference Manual for the ADA Programming Language,* ANSI/MIL-STD-1815A-1983.

[GR83] Goldberg, A. and Robson, D., *Smalltalk 80, the Language and its Implementation,* Addison Wesley, 1983.

[GS79] Gane, C. and Sarson, T., *Structured Systems Analysis: Tools and Techniques,* Prentice-Hall, Englewood Cliffs, NJ, 1986.

[HAT86] Hatley, D. J., *Structured Method for Real Time and General Sysstems Development,* IEEE Proceedings, 10th Anniversary COMSAC, Chicago, IL., 8-10th Oct. 1986

[JAC87] Jacobson, I., *Object Oriented Development in an Industrial Development,* Proceedings of OOPSLA'87, Orlando, FL, 1987.

[MW86] Mellor, S. and Ward, P., *Structured Development for Real-Time Systems,* Prentice Hall, Englewood Cliffs, NJ, 1986.

[MEY87] Meyer, B., *Reusability: the Case for Object Oriented Design,* IEEE Software, March 1987.

[MEY87] Meyer, B. *Eiffel: A Language and Environment for Software Engineering,* Interactive Software Engineering, January 6th, 1987.

[PJ80] Page-Jones, M., *The Practical Guide to Structured System Design,* Yourdon Press, New York, 1980.

[STR86] Stroustrup B., *The C++ Programming Language,* Addison-Wesley, 1986.

[YC79] Yourdon, E. and Constantine, L., *Structured Design,* Prentice Hall, Englewood Cliffs, NJ, 1979.