CASE - A Way out of the Software Trap

Geoff Davies

RAET Software Products BV

The Netherlands

1. Introduction

One of the hottest topics in the computer industry today is CASE, which stands for Computer Aided Software Engineering (or Computer Assisted Systems Engineering). Is this another meaningless acronym you see for a while, that disappears in a short time, when the excitement dies down? The acronym could disappear, but the concept will certainly not go away, because design and construction of business application systems is so vital a dimension of commerce and industry today, that our "profession" finds itself under increasing pressure to behave as Engineers.

Engineers today are assisted by computer technology in the design, visualisation, manufacture, testing, service, and quality control of the products of their discipline, such as bridges, buildings, automobiles, weapons, satellites, computer chips, and so on. The terms CAD/CAM (Computer Aided Design / Computer Aided Manufacturing) and CIM (Computer Integrated Manufacturing) are widely known.

So it will go with Application Software development. CASE is the term rapidly gaining acceptance for the automation by computer of the Software Development process. Automation is being hailed by many as a way out of the **Software Trap** - the trap we find ourselves in when we can't make vital new systems quickly or well enough, because of the burden of holding together inadequate systems developed in the past.

This paper will explore the background to CASE, the benefits it can deliver, and suggest an ideal toolset, especially with regard to the HP3000. Finally, some industry trends are examined for a view of where CASE is headed.

2. MIS Quality and Service goals under pressure

The march of technology in our business has not diminished in any way the pressure on MIS to improve its performance in the delivery of support to the business activities of our corporate masters. Everyone today is aware of the incredible price-performance gains in the computer, with a microcomputer being almost as common a feature of the middle class household as the television set.

Corporate leaders are therefore asking themselves why they do not perceive similar advances in the delivery of quality support systems for business activities. It's becoming such a glaring deficiency that general business publications are examining the problem, and discussing the effects and possible remedies. Our dirty linen is being washed in public.

Are we doing such a bad job? Corporate analysts say that, on the whole, we must be able to do better.

Data Processing was once a black tower, where magicians wove their secret spells, and spoke in strange language that awed and mystified the tremulous user. The minions of this domain were known to be fickle and some even had a reputation for blackmail - the secret knowledge possessed by them would leave their employers paralysed if they left the enclave for pastures new.

And indeed, little has changed in that respect. Today, experienced programmer/analysts, operators, and management are as scarce as ever - which is good news if you are one of these.

In our efforts to keep up with demand, and to maintain quality in the delivery, support and maintenance of business systems, we have become a serious drain on corporate finances, yet seem to get no closer to answering fundamental questions about the service we provide. Why do development projects so often run behind schedule and over budget? Why can we not repair software defects as quickly and easily as an engineer can correct hardware problems? Why must important enhancements to critical business systems, necessary for competitive advantage, wait so long to be scheduled and implemented?

It's generally agreed that the reasons for this, at least in the area of business applications, are to do with the ability of MIS to take the expression of a business problem and solve it with a computer-based solution. Among the reasons most commonly cited:

User to MIS communications is poor

MIS understanding of users' needs is poor

Users keep changing their requirements, increasing the maintenance load

Good programmers are scarce

MIS ability to plan and estimate is poor

MIS development productivity is low

MIS quality control is almost non-existent

There are many other possible explanations (excuses?), but the overriding impression is that MIS, who are supposed to be able to provide the total service for users, simply are not sufficiently professional in the delivery of their service.

User to MIS communications: shouldn't the responsibility rest with MIS to behave as business analysts and CLARIFY a user's requirements before a single line of code is written?

Scarcity of technical resources: where is the real problem - is it programming or analysis?

MIS planning and estimating ability: other departments (engineering, manufacturing, distribution, etc) can submit well-managed business and project plans - why can't MIS?

MIS development productivity: little gain has been made in development productivity in the USA in recent years. Few can even measure it, in fact, system development is widely regarded as the last uncontrolled business activity, and one for which few benchmark measurements exist.

MIS quality control: contrast the quality control procedures (if any) of the MIS department with those of manufacturing, and there is a yawning gap. And that should come as no surprise - QC in engineering departments is based on a rigorous discipline stemming from the recognition that, ultimately, customer satisfaction (and safety) will determine the success of the corporation. MIS has only internal customers, we have an informal relationship with our customers, and they have only one choice of supplier. But MIS can build systems, on which the business must depend to survive, with no QC rules - and those systems can be grown and extended over decades to massive networks of thousands of terminals and enormous transaction volumes.

So I would suggest that we, as MIS professionals, whatever the size or budget of our department, should decide firmly on a strategy to upgrade the service that we offer, and aggressively implement that strategy while we still have a choice in the matter.

3. Strategies for MIS to reach Quality and Service objectives

Very briefly, we will look at some steps that MIS could take to approach the objectives of Quality and Service that are desirable if we are to be seen as an asset to our employers, including CASE methodologies and tools.

There are many different ways of dividing up the components of a development project, and of course most MIS departments have several projects under way simultaneously. The following four stages are a simple model, and we'll assume that it's already been decided to proceed with a project.

Design: Visualising the finished application, analysing the data and flow of data, and setting out the programming and data management requirements. Normally a pen-and-paper job, done by an analyst.

Program: Actually creating and editing programs in the selected language, and submitting a succession of revisions to a compiler or interpreter, until each program is finished.

Test and document: Iterative procedure of verifying that the programs work, going back to programming to make corrections, and finally establishing on paper or in-code how it all works and what it means (for MIS and users).

Maintenance: Does it ever end? The amount of maintenance depends on how good a job you did in the first place, in terms of NEED for maintenance, and EASE of it.

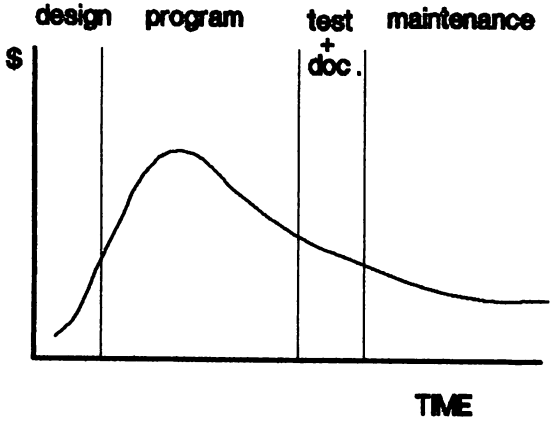

Figure 1: Traditional Approach

Figure 1 shows a theoretical model of a development project using traditional approach and the four stages (although there is no sharp line between them). The area under the curve would represent the total investment made in human and other resources in the project.

This graph emphasises that traditionally little resource in total goes into design. Why is this? Because there is little that one could do to significantly refine a design to make it more useful, once you have a few flowcharts, a list of data items, and some screen and report layouts. "Let's get on with the programming", and of course we tend to design as we go. So the bulk of cost is indeed in programming.

Testing and documentation are perhaps given MORE weight than is real in the traditional DP world. It tends to be a haphazard activity, the programmer tests his own programs, and documentation follows later - after all, we don't like to write pages of notes if we think the user might reject the system until more changes are made! Of course, if we got the design at all wrong up front, we could be a very long time putting it right later.

And maintenance, where 80% of America's programmers are busy (as Wall Street Journal would have it), just never ends. Unless the documentation was comprehensive (even through earlier maintenance), we have a lot of code to read.

What we would like to accomplish is to get the curve flatter (it makes scheduling easier), lower (it reduces costs), and for the delivery point to be nearer to the start point. There are some ways to help this.

4GL

A fourth generation language can reduce the programming load. 4GL's boost productivity enormously in low transaction volume applications, and also are very effective in rapid prototyping. They have a great ability with data manipulation, and for ad hoc report and inquiry applications. The penalty is paid in performance, and in transactions with any complexity of data management.

RelationalDatabase

Information-retrieval-intensive applications benefit from relational data management, and the associated retrieval language is usually easier to code with than Cobol. Again, the penalty is paid in high-volume applications, and you still need traditional or fourth generation language for full-function application development.

Code analysers and restructuring

There is substantial growth in this segment - the advantage comes in making old code maintainable. Obscure algorithms developed twenty years previously can be made readable for today's analyst.

DataDictionary

Implementing a standard data dictionary is a very real way to introduce some productivity - making data definitions re-usable by programmers on a team, or maintenance programmers in the future, reduces the amount of redundant coding considerably.

Generators

Code generators and report generators are a good way to re-use programming done by somebody else. Usually driven by a procedural language, you can get skeleton or even complete programs from a few statements.

The problem with all of these, and the many other productivity tools available today, is that few are integrated, and there remains an almost obsessive emphasis on the **program** as an object of attention. It seems to be overlooked that the objective is to build **application systems**, that's what the business needs us for.

Furthermore, how do we measure that we are in fact gaining in productivity at all, and to a sufficient degree that we can assert that our service is improving? Development productivity, as mentioned earlier, is an almost immeasurable quantity. The most commonly cited measurement is "lines of code per day".

Two problems with this: first, what constitutes a line of code? A line of Cobol? A field defined on a screen? A line of a DBSCHEMA? Is a replaced or deleted line a line of code for productivity measures? What if I copy 1000 lines from another program, for a "same as except" purpose?

Second, if a line of Powerhouse code can do what ten lines of Cobol do - am I ten times as productive? Is this true if the other tasks surrounding the programming (design, testing, problem resolution, editing) take the same amount of time anyway?

Understanding the productivity average for your development center is important if you are to be able to truly know that you have made improvements. Knowing productivity by **function** and by individual **personnel** can be very helpful in determining what resources to apply in a development task. How useful would it be if you **knew** the average time it took a skilled analyst to produce a transaction of medium complexity, when estimating time and cost for a new project? And the time it takes a trainee programmer to produce a new screen display for an existing system, including testing and documentation update?

Your manufacturing department has this sort of information, relevant to their operation.

In all of the tools available to the HP3000 user today, there are very few that help with design. You can obtain PC-based applications that help you understand the data and flow of data in business systems, many even produce diagrams to use as a starting point for programming.

And yet, it is in failing to get the design **right** at the very beginning that our problems begin. A business application system, including all of its screens and menus, all its reports, all its transactions, data management activity and system management, form a critical structure, supporting the corporate activities. Compare it with, for example, your head office building.

If we erected buildings the way we put application systems together, we would start from a sketch plan, hand-craft the building from the roof down, every room would be a different size, shape, height, we would make all the fittings ourselves instead of using standard ones, express surprise when the owner said they wanted 12 floors, not 3, and finally we would hand it over and say "use it for a while, and tell me what you think - if it's not quite right I can make some small adjustments!" When we thought it was all finished, we would get around to drawing up the "real" plans - if we hadn't started another project. Maintaining our building would entail going into the building and rearranging it until it "felt" right.

Now let's use CAD/CAM as an example. Who can deny the value this has been to the engineer, who can now construct at a workstation a complete specification for a machine, inspect and adjust it, before having manufacturing put it into production. He doesn't tell the computer every item of detail of a gear wheel, for example. He tells it he wants a "32 tooth spur gear of radius 4.25 inches" - and then adjusts it and moves it around. Engineers quickly became familiar with their new tools in spite of resistance by MIS (who I have seen challenge the competency of engineers to select and use them).

To conclude this part, let's look at our graph again (Figure 2) and see how it might look with an **engineering** approach to development.

We have a flatter curve, with the earlier delivery of the system. The programming phase is compressed, because in an engineering approach, rather than telling the computer how to do everything (programming) we concentrate on telling it what we want accomplished. We then leave it to the computer to assemble as much of the design into our desired executable application system as possible. We give the computer the task of coding the solution, using highly re-usable code structures.

How do we implement design on an HP3000? We have to select tools that will allow us to interact with our HP 3000 terminal as a computer aided design workstation, and which will interact with all the other parts of the development cycle, giving us a complete "software factory". This is the objective of CASE.

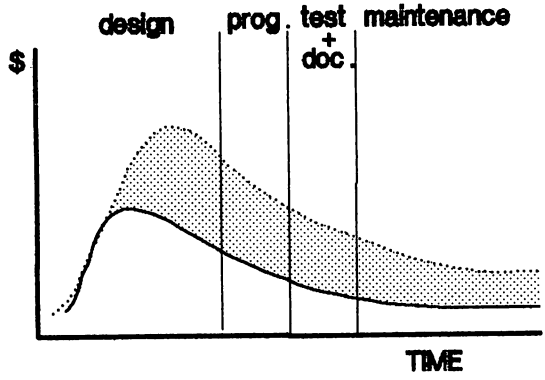

Figure 2: CASE Approach

4. An integrated CASE toolset

In this section, you're asked to forget about writing **programs** to deliver business applications. Thinking **programs** makes you relate to CASE at a level of **lines of code**, and to do so is to be fettered by tradition.

Instead think in terms of the **systems** you need to produce, and of those systems as made up of components, and sub-assemblies, rather as a manufactured product might be.

Our proposed CASE toolset may not correspond exactly to how you might perceive the vital parts of an integrated software engineering environment. There are many different ways of representing a CASE toolset, this is just one. The toolset you see here (figure 3) is oriented to a total application development environment.

In this diagram, the CASE tools are visualised in a "case" - and tidily packed away. This has no bearing on the order you might use them in. It helps you to see the interaction between each component tool.

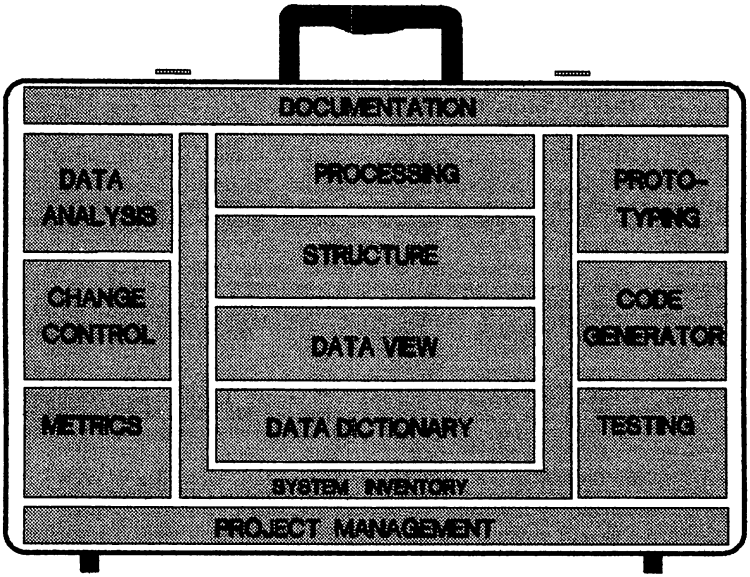

Figure 3: CASE Toolset

DATA ANALYSIS

Sometimes called "front-end" CASE tools, these have been around for some time. Used to analyse the behaviour of data in an existing or planned business application, they help the development professional's understanding of the future design and data management requirements.

Large installations, especially mainframe users, tend to favour these products more, perhaps because they seem to fit very well to relational data management.

The usual approach is for a skilled analyst to survey the user's application area, capture into the PC the data entities identified, and as much information as is known about these entities. Progressively, the whole story is built up, including the logical organization of the data, the updating points, the relationship to other entities, the properties of the entities (editing, type, size, and so on).

Output from these tools is typically presented in graphical form, in **entity-relationship diagrams**, and **data flow diagrams**. Some tools are directed at specific structured programming techniques, and produce diagrams in a compatible form.

Increasingly, use is being made of the output directly, by accepting the entity-relationship information as an initial data model for the Data Dictionary, especially on mainframes.

CHANGE CONTROL

Whether it's the introduction of a completely new application, or the adjustment of an existing system, Change Control (or Change Management) is a vital aspect of controlling a lively development environment.

Volumes have been written about effective control of change and its impact on existing Management Information Systems. Micro and minicomputer users are notorious for their cavalier attitude to changes in a running application. Mainframe users have been in the business long enough, and in a hitherto more complex environment, to know that these systems are fragile things. Quality Control is closely linked - although we don't tend to think of that.

Most old hands have had the experience of a "minor change" causing massive disruption to a critical system. It's all too tempting to think we know all we need to, and make a "quick fix" that later turns out to have an effect we did not expect. The problem is exacerbated with interpretive code environments - we often want to try it and see in a live usage. Even when a backup is available to undo the damage, the disruption can be fearful.

So a management information system for MIS itself is needed - whereby procedures are in place to control changes to sensitive systems. To return to the manufacturing analogy - change to established design normally goes through an EC (Engineering Change) approval process, with QC, Engineering, Marketing and Shop Floor inspecting the change and commenting on or planning for the effect of this.

Our ideal Change Control system will include forms for submitting problem reports and change requests; a submission and registration procedure; a design review, comment and approval procedure; quality assurance procedure (by which sign-off is given that impact analysis, check on relational integrity within programs,testing, documentation, and user-advice have all been done); and a release procedure. These procedures can be real-time and continuous, in a DP shop there does not even have to be any paper. Why not give a change/problem report facility entry point to users on all application systems?

A further issue arises here, and that is controlling the versions of software in use, especially in a distributed systems environment. Change control should therefore take account of the version in use, at the point where change is needed, and being aware of the effect on other current or future versions.

By dealing with the very natural process of change in a professional and **engineering** mode, we enhance the quality of our service, and gain greater confidence from our clients.

METRICS

Repeating the earlier assertion that productivity of the development center is virtually impossible to measure objectively, we should look for tools that will help with this, if our development resources are substantial.

There are many HP 3000 users in the world with multiple development centers, with many development staff at each site.

Our ideal integrated CASE toolset will permit us to **measure** our production and productivity. This is very important, if we want to collect project progress information, but we will come to that later.

Here we raise a concept that has not yet been mentioned - **"Object Oriented Programming"**. This concept is emerging in a PC development environment - we see an object-oriented interface in the HP-NewWave environment, and the famed Macintosh interface. At the start of this section you were asked to forget about lines of code, and programs, but to think in terms of systems.

Object-oriented **development** is more encompassing than **programming** but the idea is the same. Development activity and progress is easier to express in user-oriented terms, to the users themselves, why not measure in the same terms?

Our CASE tools will help us develop **"objects"** such as screens, reports, transactions, menus, database definitions, and so on. Let's measure our progress using those terms. Integrating metrics to object development means that our Metrics module logs resources expended in accomplishing the development of a unit. For example, let's take a screen as an object of development.

Metrics logs for the screen, the total sign-on time (programmer resources), CPU cycles for design, CPU cycles for forms generation, and also can study the make-up of the screen to arrive at a complexity measurement. If you want, it can also identify who did the job. Now, this is not such a horrifying idea - in most other disciplines these measurements are indeed done. Comparative data on the performance of individuals may be more draconian than we could bear, but of **classes** of personnel (trainee, advanced, expert) may be very useful.

Commercial software specialists who do development work by contract would find this very useful as a basis for charging, as indeed would internal corporate development centers who would then have a practical basis for cost distribution to client departments.

SYSTEM INVENTORY

Now we come to the heart of our toolset. As with a manufacturing product data management system, our CASE tools will allow us to keep **all** system specifications in **one place**. where the developer(s) can all access it uniformly. In terms of Data Dictionary, this is not such a novel concept - but for the other items, it's quite uncommon up to now.

We will look at each of the four categories of information retained in this repository shortly. The System Inventory can only be useful if it is accessible interactively for **system designers** (note that we don't say **programmers**).

Adding, editing and manipulating specifications requires a very active interface commonly referred to as the **Designers' Workbench**. A day in the life of an analyst or programmer changes radically when using such a tool. You spend little or no time using full-screen character-mode editors; the Workbench presents your SI to you in a formatted and organised way. As soon as an item is added to the inventory, it's available for another designer to use.

A good workbench will provide standard objects, from which you can derive further standards of your own (such as standard screen and report layouts). Objects can thus "breed", through "same-as-except" derivations, and the components of a target application system can come together very quickly.

Progressively, as you describe your design, you can refine and enhance it. Alteration of specifications organised in this way is light-years ahead of reading source code to locate where changes must be made. Because our tool has a very active inventory, the effect of changes proposed can be detected and propagated quickly.

Documentation of design, long a despised task of programmers, becomes a task of attaching an annotation to our "objects" describing only what makes it different from any other similar object.

The SI thus forms a complete design specification for the target Application, or in manufacturing terms, a **"Bill of Materials"**. And like any manufacturing product data management system, you can obtain very useful information to aid decision-making: where is this data field used? what if I extend its length? A bill of materials can be "exploded" to show down to the lowest level all components and sub-assemblies.

Productivity gains in analysis of change effects and in re-use of standard objects are quantum. Factors of 10 to 20 in this area are not uncommon.

A designer familiar with the System Inventory becomes concerned only with differences in objects - designs produced by others are thus accessible and transparent, making maintenance a process that is not as arduous as poring over listings. Breaking a problem application down to detect a fault requires no program libraries, no UDC listings, no programmers' notes - you break it open in the System Inventory, and examine the component objects, self-contained pieces of the design.

Because the whole design is available to all design staff, division of work to specialists becomes a simple matter, for example into screens, reports, processing, structure. Measurement of progress (even in the absence of Metrics) is facilitated - counting components at least gives an objective measure. Try coming up with a simple and repeatable way to estimate the percent complete of a source-code program!

And of great importance, "programming style" becomes minimised in impact. The "software Picassos" among us may be perturbed by that - we believe that our own style is outstanding, and want to leave our signature on our works of art - but we can't stand maintaining code of others, because they are never as competent as ourselves. The word "elegance" turns up frequently when programmers are describing arcane coding problems.

We will now examine the contents of our System Inventory, and how it helps us produce better systems faster.

DATA DICTIONARY

This is a repository of "Data about Data". Data Dictionaries have been around for some time, most vendors offer one, HP 3000 users even have a choice.

The Data Dictionary keeps all we need to know about data in an accessible location, and all developers use the DD to reduce redundancy and error in their use of data items. The information kept here is **fundamental** or describing the properties of data items, **organisational** defining relationships, and **physical** describing for example disk data management.

In a CASE environment, where re-usability is a major objective, and where the properties of a data item are part of the "object definition", we need to know a whole lot more.

Added to the usual descriptive information in the DD, such as **Identity (name), type, and length**, etc, we also want to define other properties, that will be available to our application. So we also want to know **headings** for use in reports and displays (a one-character code with a long name might justify a short heading); **security** or ways to identify create/update/read access; **ranges** for automatic input validation; **tables** again for input validation; **editing** for input and output; **locking** if required to prevent simultaneous update by two transactions; **defaults** when not filled-in; **HELP** to display at

input if the user is uncertain; **structure** if an entity is part of another entity; **sub-fields** if an entity has them.

This is not an exhaustive list. The important point is that in our ideal toolset, all of these are properties of the data item, and you do not have to code them to have them available in your application. If the data item is accessed, all its properties are automatically there.

Because defined data about data is all in the inventory, useful information can be given online or off-line to the designer. Decisions Support is given by cross referencing; where-used; search and retrieve. Defining new data objects is rapid with "same-as-except" activity.

Maintenance happens faster and with better results - because the designer making a data change can inspect the ripple effect of that change. Much maintenance of finished applications can be carried out simply by selecting and modifying a simple data entity definition (for example ranges, editing, prompting, HELP).

The ultimate beneficiary is the application user - data items are presented, prompted and handled more as he intended when he first explained to the designer of the existence of the items. He gets consistent treatment from his application - because editing, validation, prompting, annotation are always the same, rather than a different programmer's interpretation each time.

DATA VIEW

Defining the Data View is the process of telling your CASE toolset how the User wants the application to **look**, and to some extent, **feel**. From the on-line viewpoint, this is the layout of Screens, the way exceptions and errors are handled online, the nature of user prompting, and menus.

For off-line operations, the layout of reports is part of the data view.

And navigating through it all is also part of Data View. What's different from traditional programming is that we store standard and special Data Views in the System Inventory, as **objects** for managing and maintaining. All the facilities of the Data Dictionary are available for constructing the Data View - headings, edits, standard data validation, and so on. And because we've defined data entity relationships, associated data items can be mapped into screens and reports together (i.e. groups of elements are presented together for input/output).

Now we should also attach to our Data View objects some other information, mainly a reference to special processing, if any (such as related-item validations, for example, three input fields might have to add up to 100), and, of course, HELP describing how and what to do at a menu or screen. This approach separates **screen** processing from the driving transaction - and again makes maintenance easier by breaking the overall application into manageable "chunks".

You should have tools in your Workbench with which to define and edit Data Views, rapidly re-use existing components (same-as-except) and with which to inspect and adjust the actual appearance of it (e.g. screen painting). You should never have to enter code-inspection.

What are the benefits of separately defining Data View? Because it's here that 80% of user acceptance problems occur, then by reviewing your design with the user, BEFORE the tougher part of design begins (processing), you can adjust the design, even before the user's very eyes. It is even worth considering giving the user limited access to this part of the design to make the minor adjustments or type in the online guidance themselves (HELP text, etc).

STRUCTURE

The advantage of CASE's object-oriented approach is the separation of the desired application into components which are easier units to manage and maintain than programs. Systems built in traditional programming methods rarely have a clear architecture or **structure**, with the consequence that a great deal of effort goes into laboriously writing code to facilitate navigation (menus, related transactions, etc); and to handle the interaction of data management (database and files), manipulation, and data presentation (or Data View). What is odd about this is that almost all of this has been done before, but only sometimes do we take the trouble to establish re-usable code to make it easier the next time.

Enter CASE, in which a catalog of standard structures for all standard programming logic is already available, eliminating the need to program it. This does not mean that you now have to manipulate Cobol with the aid of copy-books.

CASE gives you, once again, **objects** which are pre-programmed structures for System Management (menu tree structures, for example), applications (where the component transactions are defined), and transactions themselves (online and batch). These structures can be thought of as **models** that you select, manipulate and customise through parameter settings.

Within each **type** of structure, all the management is provided automatically by model in use. Take for example an online "browse" through historical records. We select a "BROWSE" transaction model, and proceed to specify the customisation necessary to make it unique for our user. This would involve naming the Data View(s) attached, and the Data Management required (data base(s) and/or files); naming the **processing** objects (next section) to be invoked at the **sockets** in the transaction; and the relationships between the transaction and the application, and other transactions.

What you get automatically is management of database activity, function key recognition and action, correct entry point and initialisation, correct housework at end, management of appropriate HELP to the right point at the right time (carried in from DD and DV), and sensible error handling.

It's akin to selecting standard foundations, columns, bearers, partitions and roofing when designing a house.

Now we benefit in design and maintenance by isolating system navigation and transaction logic problems or changes to parameters, rather than having to handle source code and Job Control Language. Our user is happier, because getting into, around, and out of a system is **always** the same, regardless of which system it happens to be. Peculiarities and quirks can are sometimes amusing, often downright annoying, are a thing of the past.

PROCESSING

Processing, or calculation and data manipulation logic, can also be reduced to components of a structured application suite. The component, or **"logic object"**, is a self-contained and re-usable System Inventory item, with associated properties. The nature of those properties is defined by the designer, but the properties include, besides an identity: a **description** for the original and subsequent designers of purpose and technique employed; a definition of the **error handling** (e.g. display message, re-set work areas); the **work area** or **common areas** that the logic accesses; **linkages** to other processing logic objects, depending on success or failure; and other possible properties.

Of course, the actual data manipulation and calculation logic itself is also a property of the object.

In any CASE environment, it is almost impossible to eliminate the need for a high-level or macro definition language. Our ideal CASE toolset includes an interactive editor to allow you to enter new logic, or copy other logic (same-as-except) as a starting point. The editor must include features to ensure that the logic is syntactically correct, and that we work with real data (defined in the DD).

The CASE methodology relieves much of the tedious coding chores from the designer, through its Data Dictionary, Data View and Structure facilities. So it's probable that 80% of the design coding effort will be expended in this part. Our CASE Process Definition Language must therefore provide a very high level of macro capability, reducing to a few keystrokes what normally requires a substantial coding effort.

The highest-skilled analysts in your installation can preserve in a re-usable form their skill for access by less experienced personnel. Because our CASE processing puts a "fence" around a piece of logic, it becomes simpler to read it, understand it, use it and learn from it,

Again, this separation makes maintenance far simpler. You do not have to be the originator, nor have any written manuals present, to be able to isolate a problem piece of logic and fix it. And in development, the accomplishment of a logic object or process definition means that you need never handle that code again,

Contrast this with traditional programming, in third or fourth generation language. Construction of large and complex programs tends to be an iterative cycle of edit, compile until clean, add more complexity. Even parts of a program that are running OK get compiled again and again. Sometimes, the subsequent re-editing messes with code that was running satisfactorily.

Once again, it's pointed out that the chief beneficiary in the end is the user - who gets a better response from DP for maintenance and new system development.

PROTOTYPING

We now leave the System Inventory, at least in regard to changing its contents. But Prototyping, a proven technique employed by many for verifying design, can participate in the SI as well in our ideal CASE environment.
If we did not get the **look** of our system right using the Data View facilities, we can generate a prototype application and actually run the application. When does a prototype stop being one? The answer is probably, when you feel like the design is nearly finished.

To truly prototype what the finished application will look and feel like, and picture what a day in the life of the user will entail, you have to be able to go from transaction to transaction as the user would, not by starting Data Entry under Formspec. The user needs to see, and you also, what the screens and reports look like when there is data appearing in them.

In our CASE environment, therefore, you can check the look in the Data View parts of the workbench, as well as the prototype. But for the feel you need to start the application from the Operating System, as in "real life".

Which is why, the prototype is best made as the word intended - not in some special make-believe environment (such as by using dBase), but is in fact the **first effort** from the toolset, and in fact is the first reviewable version from the code generator (next).

As with verification of the look in Data View, up to 80% of rework can be eliminated from the post-delivery phase if you do this phase right. And that makes us all happier.

CODE GENERATOR

Ideally, you should use the code generator to make the prototype, because then it is a **true** prototype.

Why a code generator? Its not the only choice, there are quite a few CASE environments today integrated to a 4GL. But 4GL's often give real performance problems in highly transaction-intensive systems. A code generator that makes compilable code gives you some fringe benefits.

First, if generated code is recognised third generation language, such as Cobol or Fortran, then you maintain independence of development environment, your 3GL code is still maintainable; second, distributing applications based on a 4GL inevitably leads to high charges for runtime systems; third, distribution of **compiled** 3GL code gives a large measure of protection against copying of proprietary source and reduces size of application libraries on smaller disk systems.

A fundamental assumption here is that, because the CASE tools are integrated, the Code Generator can read the specifications from the SI.

The Code Generator makes more than just executable code - it should also create source code, data management schemas, initialise databases, build screen forms (e.g. VIEW), write the linking job control, bind message and HELP text files into the whole, and act on your instructions for organization of the executable libraries (USL's etc).

A number of benefits accrue from the CASE Code Generator. Reliable and correct code generated does not have to be debugged - only SI specifications need to be; consequently, the old cycle of edit/compile over and over becomes less prevalent. CASE designers don;t feel the need that programmers do, to get one program perfect before going on to the next one. In the CASE environment, you could design for weeks and never generate or compile in all that time. Then, as you get to prototyping and refinement, you begin to need to generate.

It becomes possible, and I have seen it, that contented design staff schedule all their generate/compile activity for the back-shift. It's more conducive to good design to simply work with the interactive tools through the day, bypassing the edit/compile programmer's cycle with it's trips to the printer, sessions with spook, and "just one more small change and it will be right". Less erratic, unscheduled, and CPU-intensive activity makes for good response times.

TESTING

The automation of Testing is a late-emerging part of CASE. There are some tools available that allow you to automate an interactive terminal session that signs on, enters transactions, creates reports, deliberately makes mistakes, and so on, according to your own scripts and then gives you a report on what was different from the last time it did it.

Specification of standard test procedures like this is vitally interesting to software package developers, or to those with colossal user populations.

You could check for code that was never entered (why do we need it? Maybe our test was inadequate, or we have a logic error?), and try all the "ridiculous" values for input ("I never would have expected a user to enter minus 5 for the month").

Another interesting area we want for our ideal toolset is an extension of the Data Dictionary, to describe the intricacies of data **behaviour** in the finished application. This might for example describe the usual distribution of number of order detail lines per order header; the usual distribution of letters in the NAME field of a NAME & ADDRESS group.

A test data generator would then construct full data bases based on the statistical information you predict, for you to test. One of the hardest parts of testing is to get enough realistically filled records to make all screens and reports look as they would when the application has been in use some time. Or even, to be able to estimate the response time of an inquiry application accessing a data base with a million records, and a complex structure. Test data generators can do this.

DOCUMENTATION

There are already a number of tools on the market to help us do what we least like doing, documenting our systems. But they rarely are able to shine light on purpose behind logic that they scan. Our System Inventory is self documenting. The designer is disciplined more to create a small amount of descriptive information when creating Processing objects, where he is outside of the boundaries of pre-structured or special-purpose objects.

All the pre-structured and special purpose objects (screens, data elements, reports, transactions, menus) have such a defined logic and purpose that further manually produced documentation is redundant.

An active System Inventory therefore carries all its documentation within it. The CASE toolset only needs to provide access to the design for management reporting and designer review. This is accomplished by reports and inquiries, and is an application that should offer choices of levels of depth and complexity.

Calling for a full set of reports describing all structures, processing, messages and text, screens reports and data dictionary - is asking for a System Reference Manual. It's always up to date.

Calling for menus, screens, and report layouts, annotated with the validation rules, ranges and associated HELP text for all input fields and action screens - is asking for a User's Manual. And it's also always up to date.

DP staff benefit - that burden of guilt for incomplete documentation lifts from your shoulders. Your client user benefits also - machine-produced documentation is able to be customised and formatted - I've seen very smart manuals produced with corporate logos and other frills, using a laser printer and CASE documentation tools.

PROJECT MANAGEMENT

These tools are not exclusively the preserve of CASE, but when you run the rest of you development environment so well, why not underpin it with an integrated toolset that helps you maintain control over large development projects and maintenance? All project management tools include a Critical Path algorithm, defining the shortest path between the start and finish points, and many include cost control and resource management facilities.

Integrating Project Management to the rest of the toolset means measuring progress (accessing the SI) and determining if required stages have been passed yet. It also means that Metrics can be actively employed to determine, based on previous performance, when a project will be completed.

This can only lead to trust in MIS, and more satisfied users.

5. Industry Trends

We shall now review recent developments in Computer Aided Software Engineering, and see whether any trends are emerging.

Technology

CASE workstations driven by proprietary CASE software are expected to become a growth area. Analogous to CAD workstations, the CASE workstation will use icons to ease the selection of objects for design activity, and windows to permit rapid navigation through the toolset, and to run testing and design side by side. Some engineering workstation vendors are already producing CASE workstations for UNIX systems.

Workstations which are fully compatible with the target environment are today quite common and the new HP 3000 LX models could be considered to be development workstations for HP3000 corporate systems.

Start-ups

Business Week reported in May that the worldwide market for CASE tools could hit $2 billion in 1992, and in the US alone, $800 million.

It's no surprise, then, that there are many people getting into the act. In the same article: a San Francisco consultant sees two new CASE start-ups a day, and an analyst reports that he knows of some 100 CASE companies, mostly less than two years old, and most funded at more than $1 million.

Perhaps we'll see the same explosive sort of growth that the micro started.

Standards

It only becomes interesting to attempt to establish industry standards for any new technology once there is a discernible movement to embrace that technology by large numbers of pioneers. There are no standards yet for specification of software design, and no standards for productivity measurement. The IEEE has struggled for several years to come up with a "single perfect measure" of software productivity, but few believe it is possible.

What is more likely is that the new CASE technology and object-oriented design and programming will permit accurate measurement of development effectiveness, but comparison to former methodologies will not yield any accurate figures, because the old methods of measurement are so imprecise.

Alliances

Recognition by the computer industry that growth is stunted by the inability of corporate clients to take full advantage of the power of new computers, because of the maintenance and development backlog, has greatly accelerated the interest of vendors in having CASE tools available.

lowering the cost and improving the quality and service offered by MIS might make more budget available for more hardware purchases, and growth in user populations can only lead to increased peripheral and capacity purchased.

Consequently, we are seeing major hardware vendors and software companies teaming up, and manufacturers of discrete CASE tools getting together. Even major corporate computer users are invited to have their say.

In 1985, fourteen leading aerospace and defence contractors formed a limited partnership called the Software Productivity Consortium. They have an invited group of CASE vendors called the Guest Systems Council, and jointly they are attempting to formulate complementary strategies for their future mutual benefit. The last press statement I saw was optimistic, but nothing concrete had emerged.

Education

MIS is traditionally adverse to risk, and as a consequence is not yet ready to embrace the new technology. Few companies want to be pioneers, and few in corporate MIS want to lead the way for their colleagues. As Computerworld put it, we need a "hero in the programmer's shop".

However, a growing number of CASE evangelists such as T. Capers Jones, James Martin and David Yourdon are defining the CASE environment thoroughly, and some excellent publications from some of these authors are available. There will probably be an increase in the exposure of computer science students to CASE methodologies.

Cost

There is a huge variance in the price of CASE tools, just as with 4GL's and databases, depending on whether you're an IBM mainframe user, or a small mini user.

A total life-cycle CASE environment recently announced for mainframe IBM by one of the "Big Five" accounting firms, includes an IBM PC based methodology front end at $50,000 for a site license; a design interface for filling the repository at $7,000 per networked micro workstation; and a generating/implementing back-end that costs a hefty $200,000 for a single license.

At the other end of the spectrum, the most popular data analysis PC based front-end is around $8,000 per copy, and a full application development and documentation environment can be had on the HP3000 for just $30,000.

The cost of the tools is, of course, offset by the gains in productivity, and the competitive advantage to the corporation of having high-quality and maintainable systems available faster.

CASE is here, and it's growing in importance. The signs are in the industry that it is going to reach all of us, very soon, and very pervasively.

---

6. Summary

We've reviewed the pressures on MIS to upgrade the quality and service which is its responsibility, and we've seen how executive attention is becoming focused on software development productivity as being at least part of the problem, where it exists.

One of the possible strategies to help us drive towards MIS quality and service objectives has been examined up close, and one ideal set of tools has been proposed. The ideal set of tools would transform our development center from a craftsmen's workshop, into a professional software engineering center, with consequent advantages already enjoyed by engineers in other disciplines.

And finally we have glimpsed a few of the developments relating to CASE in the computer industry, which augur for a healthy period of penetration and growth for this technology.

Attendees are invited to discuss the CASE approach further at the RAET Software Products exhobot, number 1015.