DEPARTEMENT TOEGEPASTE ECONOMISCHE WETENSCHAPPEN

ONDERZOEKSRAPPORT NR 9629

A GENERAL FRAMEWORK FOR POSITIONING, EVALUATING AND SELECTING THE NEW GENERATION OF DEVELOPMENT TOOLS

by

J. Vanthienen

S. Poelmans

Katholieke Universiteit Leuven

Naamsestraat 69, B-3000 Leuven

ONDERZOEKSRAPPORT NR 9629

A GENERAL FRAMEWORK FOR POSITIONING, EVALUATING AND SELECTING THE NEW GENERATION OF DEVELOPMENT TOOLS

by

J. Vanthienen

S. Poelmans

D/1996/2376/29

A General Framework for Positioning, Evaluating and Selecting the New Generation of Development Tools

J. VANTHIENEN & S. POELMANS
Katholieke Universiteit Leuven
Department of Applied Economic Sciences
Naamsestraat 69, B-3000 Leuven (Belgium)
E-mail: Jan.Vanthienen@econ.kuleuven.ac.be
Stephan.Poelmans@econ.kuleuven.ac.be

Abstract

This paper focuses on the evaluation and positioning of a new generation of development tools containing subtools (report generators, browsers, debuggers, GUI-builders,...) and programming languages that are designed to work together and have a common graphical user interface and are therefore called *environments*. Several trends in IT have led to a pluriform range of development tools that can be classified in numerous categories. Examples are : object-oriented tools, GUI-tools, upper- and lower CASE-tools, client/server tools and 4GL environments. This classification does not sufficiently cover the tools subject in this paper for the simple reason that only one criterion is used to distinguish them. Modern visual development environments often fit in several categories because to a certain extent, several criteria can be applied to evaluate them. In this study, we will offer a broad classification scheme with which tools can be positioned and which can be refined through further research.

Keywords

visual development tools, programming environments, visual programming, object-oriented programming, software engineering, client/server

1. Introduction

This paper presents a broad classification-framework of development environments that are characterised by graphical, user friendly interfaces and may be intended for robust industrial-strength code or for rapid prototypes or "one-off" applications that will be discarded after a few users.

The success of these tools increased gradually and was encouraged by several tendencies. First, with the trend of downsizing in the eighties and the backlog end-users already faced, end-user computing became a vigorous trend and languages were needed that were easy to use and if possible non-procedural (Martin, J. 85, p. 2). Basic fourth-generation languages meet these requirements but are domain-specific and are mostly limited to be used in office-automation environments (Bodker, S. 91, p. 131; Beek, G.V. 1987, p. 889; Jande, H.J. and Achterberg, J. 1988, p. 1006).

Together with this phenomenon, the object-oriented technology lifted out of research projects and into real-life organisational surroundings. Numerous traditional (third generation) languages transformed to the OO paradigm and integrated OO concepts, which resulted in *hybrid* languages like Object Pascal, C++, Objective-C, OO-COBOL,.... (Winblad, A.L. 90, p. 59; Harmon, P. and Taylor, D.A. 93, p. 33; Hopkins, T. and Horan, B. 95, p. 7) On the other hand, pure OO-languages were designed on the basis of OO-concepts and languages like ADA, Eiffel, Smalltalk and CLOS emerged. The use of OO-classes led to the standardisation of class-libraries that support the integration of distributed applications, are less platform-dependent and support to a certain extent language-independency (e.g. DCE, CORBA, DSOM,...).

In the beginning of the nineties, a new trend occurred as a lot of research was conducted on the possibilities to program in a visual way.(Burnett, M. 95; Glinert, P.E. 90; Shu, N.C. 88) Attempts were taken to present and manipulate program-structures using pictorial elements and graphical interfaces.

The convergence of these tendencies led to the creation of visual development tools containing subtools (report generators, browsers, debuggers, GUI-builders,...) and programming languages with a compiler or interpreter that are designed to work together and have a common user interface and are therefore called *environments. (*Taylor, D.A. June 95, p.47; Taylor, D.A. 92, p. 152) As a result, a pluriform range of development tools can be classified in numerous categories such as object-oriented tools, GUI-tools, upper- and lower CASE-tools, client/server tools and 4GL-environments (Verhoef, D. 95, p. 16). This classification does not sufficiently cover the tools subject in this paper for the simple reason that only one criterion is used to distinguish them. Modern visual development environments often fit in several categories because to a certain extent, several criteria can be applied to

evaluate them. In this study, we will offer a broad classification scheme with which tools can be positioned and which can be refined through further research.

Based on the idea of (Howatt, J. 95) a distinction is made between software-engineering criteria, human-factor criteria and criteria that relate to consulting, support and other costs. The former category consists of criteria that are inherent to the functionality of a tool as it is presented by a vendor and concerns the object-orientedness, client/server support and productivity of development environments. Human-factor criteria point to criteria that are dependent on the way the tool is perceived and learned by the user. Finally, some attention is paid to the way experience, consulting costs and vendor support may affect a selection. When evaluating a tool, it should be assessed along these dimensions and then be selected to fit the needs of the project at hand.

2. Software-engineering criteria

This paragraph focuses on three major criteria. Section 2.1 elaborates which elements are decisive when regarding the object-orientedness of tools. Section 2.2. explores the purpose of development environments, whereas section 2.3 indicates the issues that are related to the client/server functionality of tools. Finally, performance and efficiency aspects are highlighted in section 2.4.

2.1. object-based vs. object-oriented

Before comparing languages that meet certain features of the OO paradigm, it is necessary to clearly outline the cornerstones of object-oriented programming.

(Cardelli, L. and Wegner, P. 85, p. 481; and Booch 94 p., 38) state that a language is OO if and only if it satisfies the following requirements :

- the language should support data abstractions with an interface of named operations and a hidden local state;
- objects should have an associated class;
- classes are members of a hierarchy, united via inheritance relationships.

These conditions are important because some programming tools claim to be OO but only refer to abstract data types or objects without classes or an inheritance structure and hence lack the possibility of polymorphism. Such languages can not be considered OO but are usually referred to as being object-based. (Stroustrup 91; Booch 94; Agha, G. A. and Wegner, P. 93)

In order to be able to classify tools according to the degree of object-orientedness, we call on the object-model, elaborated by (Booch 94, pp. 27-81). Booch presents a model encompassing all the elements necessary for a language to be considered truly OO. He makes a distinction

between *major* factors (abstraction, encapsulation, modularity, hierarchy) and *minor* elements (typing, concurrency and persistence). The model denotes that a language lacking any of the four major factors cannot claim to be object-oriented and should therefore be regarded as object-based. The minor elements can give an indication to which degree a development tool is object-oriented

However, since modularity is not typical of OO-concepts (non OO-languages (such as C or Pascal), may provide a modular structure (files and units respectively)) and is implicitly present when the three other conditions are fulfilled, it will not be considered as a distinguishing factor in the classification scheme of this paper. As a result, a language should then provide the three remaining factors to be truly OO : abstraction (class and/or instance variable or methods), encapsulation (using private and public variables and methods) and hierarchy (single or multiple inheritance and metaclasses[1] and/or generic classes[2]).

When further interpreting the model as presented by Booch, the relationship between typing and dynamic binding should be considered. The way dynamic binding is implemented depends on whether the language is statically or dynamically typed (the latter term is equally referred to as being "untyped"). A distinction is made between statically-typed dynamic binding and dynamically-typed dynamic binding. In the first form, it is not known which function will be called for a virtual function at run-time because a derived class may override the function, in which case the overriding function must be called. When the complete program is compiled, virtual functions are resolved (statically) for actual objects. These functions can be accessed (at run-time) by using virtual table function pointers in the actual objects, thus providing statically-typed dynamic binding. When dynamically-typed dynamic binding is provided, the lookup for methods is performed at run-time (dynamically). This kind of binding not only increases flexibility and loose coupling but is often required in many applications including databases, distributed programming and user interaction. (Garfinkel, S.L. and Mahoney, M.K. 93 p.80)

Although an dynamically typed language beyond any doubt provides more flexibility and fits better the OO-model, it is important to know that a number of important benefits can be derived from using (statically) typed languages. (Tesler, L. ,Aug. 1981, p. 142) points out the following considerations :

- "Without type checking, a program in most languages can 'crash' in mysterious ways at runtime.
- In most systems, the edit-compile-debug cycle is so tedious that early error detection is indispensable.
- Type declarations help to document programs.

---

[1] A metaclass is a class whose instances are classed themselves. (e.g. Smalltalk-tools and CLOS possess metaclasses.)

[2] A class that serves as a template class for other classes, in which the template may be parameterized by other classes, objects, and/or operations. In this way, new classes at the same level of abstraction in an inheritance hierarchy may be created by filling in parameters)

- Most compilers can generate more efficient object code if types are declared."

Finally, the following scheme is presented to evaluate the scripting language of a development environment :

necessary conditions (major factors)

abstraction	instance variables instance methods class variables class methods	yes/no yes/no yes/no yes/no
encapsulation	of variables of methods	public/private public/private
hierarchy	inheritance metaclasses generic classes	single/multiple yes/no yes/no

conditions to determine the what degree a language is OO (minor factors)

typing	strongly typed binding polymorphism	yes/no static/dynamic single/multiple
concurrency	multitasking	yes/no/indirectly
persistence[1]	persistent objects	yes/no

source : modification of (Booch, 94, pp. 473-488)

Given the fact that not all characteristics are interdependent, one can make several subdivisions in OO-languages dependent upon the reason for the classification. If for instance a software engineer is only interested in the fact whether or not the tool provides multiple inheritance and polymorphism, without caring for flexibility arguments, a ranking might result that does not take into account the typing of a programming environment, since single and multiple inheritance and polymorphism can occur in static binded as well as in dynamic binded languages. Another possibility consists in putting forward OO functionality and flexibility that is determined by the way of typing in a language. Features like static or dynamic binding, single and multiple inheritance or polymorphism can be absent or present regardless of the typing used in a language, but differ in the way they are implemented according to the typing used.

In the ranking below, this approach is preferred and a ranking is presented that first makes a distinction between conventional programming, object-based and object-oriented and further distinguishes OO-languages on the basis of typing, binding and polymorphism, in this order of importance.

---

[1] a means of maintaining the stat of an object across invocations

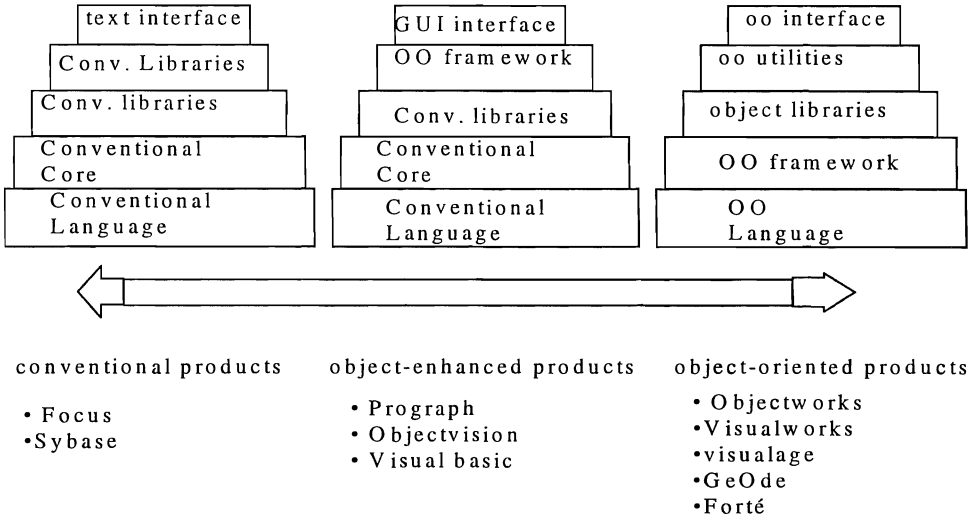
The following categories result :

conventional coding	*object-based*	*object oriented*		
		limited OO features		***extended oo features***
no notion of abstraction, encapsulation and hierarchy	lacking any of the following features : - abstraction; - encapsulation; - hierarchy.	strongly typed and static binding +(no polymorphism) + single or multiple inheritance	statically-typed dynamic binding + polym + single or multiple inher.	dynamically-typed dynamic binding + pol. + single or mult. inher.

An example of an extended OO language is CLOS, which originated from Lisp. CLOS fulfils the three major factors, but does not possess generic units. It provides dynamically-typed dynamic binding, multiple inheritance and polymorphism. Static typing can be used optionally. Smalltalk-tools (e.g. Visualage) also support dynamically -typed dynamic binding, but provide single inheritance and no generic units. C++ tools are hybrid tools that offer statically typed dynamic binding and provide multiple inheritance as well as generic units. ADA is an example of a pure but object-based language. It is strongly typed, but does not allow inheritance or polymorphism (abstraction and encapsulation, but no hierarchy). Delphi is a development environment with Object Pascal as its base code. It is object oriented and provides statically typed dynamic binding with single inheritance. Tools like Visual Basic, Access (Basic) or Prograph are not fully OO (lack hierarchy and polymorphism) and should be regarded as object based.

When interpreting this classification-scheme, one should keep in mind that there is no such thing as a perfect classification. There are potentially at least as many ways of dividing the world into object systems as there are scientists to undertake the task (Booch 94 p. 150) and any classification is relative to the perspective of the observer doing the classification. This certainly is true in this categorisation where some independent features are brought together in one continuum. Each independent feature - static and both forms of dynamic binding and single or multiple inheritance - can give rise to another subdivision or ranking if it is focused and given a deterministic meaning. Moreover, several other features - pre- and postconditions, the possibility of data-hiding, the fact that structural independent procedures or functions can be programmed, etc.- are not considered but could be important depending on the project at hand. Considering this argumentation, it is important to note that the above dimension should not be seen as an ordinal ranking from "less capable" to "excellent". It only claims to distinguish tools on certain OO characteristics that can be useful but also hindering, depending on the target project (e.g. a controversy exists on whether or not multiple inheritance can be seen as an advantage or disadvantage of a language (Templ, J. April 1992).

In this respect, (Harmon, P. 93) developed a comparable but less specific classification. In his framework, he focused on the language in which the development it self is written :



If the product claims to be purely object-oriented, it is likely being developed in an object-oriented environment. In this way, the tool becomes a kind of meta-tool of which the developer can adapt the base-classes and structures of the kernel to result in a different tool that better fits his needs. Most of the application generators on the market are developed in non OO languages like C and OO characteristics have been added on top. Or, even if they are written in an OO environment, it is possible that the underlying classes and frameworks are locked so that they cannot be accessed.

In this scheme, the two columns to the left correspond with the groups of conventional and object-based tools. The third column is a generalisation of (OO) tools that can be put in one of the three right categories in the classification presented in this paper.

2.2. general vs. specific purpose

As tools become more sophisticated, they allow the developer to write less code that is syntactically machine-dependent and more code that is related to the way humans think of application logic. This means that they assume more about the nature of the application, provide more built-in features, utilities and class libraries and in that way constraint the developer to a greater degree.(Harmon, P. and Taylor, D.A. 94, p. 27) This has the advantage that tools can be used by end-users who directly perceive the need of an information system. General purpose tools often possess less semantic guidance or constraints and allow easier manipulation of hardware functionalities

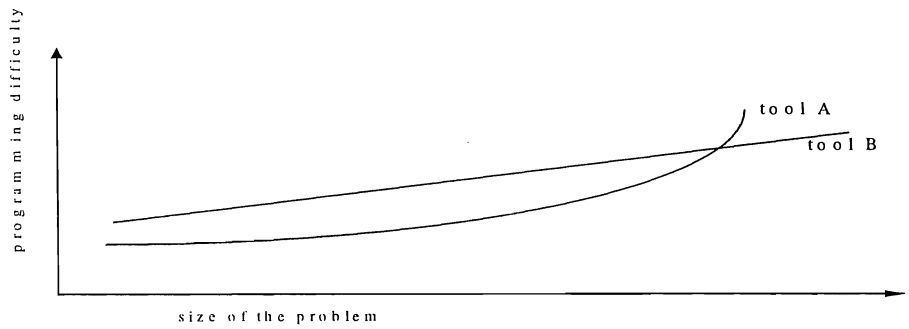
Basically, tools can be evaluated on the basis of their core programming language and the presence and seamless integration of subtools. Subtools are specific-purpose and should be checked whether their statements, functions or visual representations match sufficiently the

semantic meaning of the application-logic. Because every development environment possesses a core scripting language that often is responsible for the co-ordination of the subtools, it is of primordial importance though, to verify whether the code is flexible enough to convert general and complex problem logic in an operational application. (e.g. a non-procedural 4GL is generally speaking less flexible than a full OO language).

Based on a literature study (Verhoef, D. 95; Collins, D. 95; Howatt, J. 95; Martin, J. 1985; Booch 1994; Hopkins,T. and Horan, B.; Burnett, M., etc.) and practical descriptions and manuals of development tools, several characteristics that determine the purpose of a tool can be deducted :

the scripting-language of the tool

The scripting-language of a tool is responsible for the creation of source code and the compilation or interpretation of source code into object code. In practice, the scripting language of a tool can take several forms. According to the origin of the tool, OO-wrapped 3GLs as well as full-function 4GL or object-oriented programming languages can be used. Hybrid (originally 3GLs, e.g. Object Pascal, C++, ...) and pure OO-languages (ADA, Eiffel, CLOS, Smalltalk) are by definition procedural - in this paragraph "procedural" is used to indicate the fact that a programmer has to outline how an algorithm should be executed (using data-flows and user defined functions) and should not be confused with non-OO features - and can generate complete applications, which cannot be stated to the same extent when 4GL are considered. In fact, 4GL languages can vary from simple report generators to complete full-function high-level languages and they can be procedural, non-procedural or both (Martin, J. 85 p. 10)



source : Collins, D. 1995, p. 443: Comparison of two hypothetical tools A and B. B is a general-purpose tool, and A is only efficient and usable in (relatively) small programs

A combination of both procedural and non-procedural statements may be desirable because nonprocedural statements speed up development time and improve the ease-of-use of the language, whereas procedural operations enhance performance and extend the range of applications that can be tackled.(Martin, J. 85, p. 8).

the degree to which visual programming is made possible

A lot of visual development environments as defined in this study, do not match (despite their names) the definition of visual programming languages. Instead they are often textual languages which use a visual environment where graphical tools (GUI-builders, report generators, debuggers, browsers,...) can make programming easier on the programmer. As long as the syntax and semantics of elementary programming structures such as dataflows (selection, sequences and iterations) and datatypes have to be coded in a one-dimensional textual manner, no visual syntax is provided and the term visual programming language (VPL) is not appropriate.

Languages with a *visual syntax* include diagrammatic languages (in which nodes and arcs are the basic elements) and iconic languages, based on icons that are used to define the composition of tokens or the pre- and postconditions of actions rules.(Burnett, M. & Baker, M.J., June 1994; Burnett, M., Goldberg, A., Lewis, T., 95, p. 10-17)

Visual environments are tools that can possess a visual or a textual syntax and can therefore be based on either a real VPL or a textual scripting code. In a visual environment (whether or not a visual syntax is provided), graphical techniques are used to manipulate pictorial elements and display the structure of a program (that is originally expressed textually or visually). Sample techniques for constructing programs include point-and-click for action, invocation or selection and wiring for relating objects to another by drawing lines.(Burnett, M., Goldberg, A., Lewis, T., 95, p. 10-17). Many visual environments also include methods for displaying program information such as dataflow diagrams, dependency graphs and state transition diagrams.

The distinction between real visual programming languages and visual development environments that use graphical interfaces and techniques is fading though. Some tools such as Visualage provide visual syntax features to generate application-logic code, but the user still has the option to use textual formats as well.

Although visual programming may be appealing because it is easier to understand and memorise, provides more information in less space and makes structure more visible and clearer (Petre, M. June 95 p. 39), visual programming languages often constraint the programmer because visual representations are abstract concepts that represent a number of sequential statements which cannot be accessed individually, and are therefore less flexible. It can be stated that specific visual representations support the conventions that language-designers had in mind. VPL are therefore often less general-purpose and are directed towards a certain range of applications. Or, as Marian Petre puts it : "Graphical representations appear to offer potential for 'externalising the objects of thought' - for providing a more direct mapping between internal and external representations by providing representations close to

the domain level that make structures and relationships accessible."(Petre, M., June 95 nor. 6, p. 40).

If a tool limits the possibility of textual coding and offers visual representations instead, the degree of visual programming becomes an important factor that should be considered when determining the general-purpose character.

interfaces to routines, procedures or utilities of other languages and/or the possibility to export libraries or objects to other programming environments

It is obvious that a tool has a general-purpose character, if it is able to generate general objects or procedures that can be imported by other tools (e.g. VBX-files can be written in Visual C++ and imported in Visual Basic; C++ libraries can be exported to several development tools). With the growth of OO- and client/server applications (cf. infra), object libraries conforming to language-independent standards (like CORBA, DCE) are becoming important when constructing applications in an object-oriented manner. As a result, tools can upgrade their applicability when they provide interfaces to existing routines, modules or standard object libraries.*(Koelmer, R. 1995, p. 246)* Since most development tools lack CASE- features to support the inception and analysis phase in a phased project, it might be important that a tool provides an interface to existing CASE-tools or techniques.

the presence of standard-functions (mathematical and statistical functions, high-level statements to improve structured programming ... (Benjamin, R.I. and Blunt, J., 1993, p. 12)

The presence of these functions can be seen as an enrichment of the programming environment. It is however useful to check whether such functions are general applicable, or limited to a certain range of applications.

other low-level functionality provided by the base language

With low-level functionality, we mean those functions that do not contribute to the semantic logic of an application, but are necessary to make the application run on the hardware or operating system. Such functions include the creation of autonomous executable-files, the definition of own (error) messages, the possibility to manipulate dynamic memory allocation and the access to or creation of platform-dependent APIs (e.g. DLLs in a Windows environment).

All these criteria are the basis of the following classification :

specific-purpose tools : - database tools - internet tools - scientific visualisation languages - GUI-builders ...	non-procedural (mostly full-function 4GLs)	procedural (full-function 4GL and OO-languages)	**general-purpose tools** procedural with extended low-level functionalities (4GL or OO-languages)

Examples of general purpose tools include C++-tools, Smalltalk tools and tools like Forté, Delphi, Visualworks, etc. Tools such as Powerbuilder 4.0, Visual Basic 3.0 are illustrations of general purpose tools that lack low level features like creating platform dependent API's or memory allocation and cannot create objects or libraries that can be used by different languages or tools... In the category of database-tools, Access (Basic) (a "subset" of Visual Basic), Visual Dbase, Visual Foxpro and Developer 2000 are representative examples. Most of these tools lack the possibility to create autonomous executable files and/or possess detailed functionalities that are intended to access data(bases). Some examples of widely used tools intended to construct internet applications are Perl and Java (a modified "subset" of C++). An example of a specific User Interface Builder is TAE-Plus (Szczur, M.R., Sheppard, S.B. Jan. 1993, pp. 76-101). This development tool allows the user to prototype GUI's and rehearse them, which helps the user to check and feel the look of various designs. Other examples of GUI-builders are Serpent or Teleuse (Szczur, M.R., Sheppard, S.B. Jan. 1993, pp. 76-101). It is important to state that most modern general purpose environments also possess the necessary features to build user-friendly interfaces and access databases. These tools mostly provide GUI subtools and/or database subtools that allows the construction of user interfaces and interfaces to databases using visible components. The disadvantage of the general-purpose tools is that they are more complex and hence demand more training time (cf. infra).

When applying this scheme on a larger scale, it becomes clear that a dichotomy exists that distincts the specific-purpose tools (left column of the classification) from the general-purpose tools (right three columns) (Collins, D. 95, p. 430). The differences between specific and general-purpose tools are multiple and restricting factors can in most cases easily be recognised (although it is difficult to construct metrics that quantitatively measure certain features). The distinction between the two right categories though are more subtle and not always clear. The possibility to produce standard libraries, the ability to develop and manipulate low level functionality and the nature of the base-language (4GL, OO) can be considered as being decisive.

As tools become more specialised and assume more about the nature of the resulting application, selecting the right tool to fit the application is of primordial importance. The difficulty in this approach is that each application domain has unique requirements and even

if a set of domain-specific criteria were developed, not all of the criteria would apply to the same degree for each problem within the domain. (Howatt, J. 95 p. 38)

Although few research has been conducted to cover this category of criteria, some researchers defined criteria based on the intended use of the language. (Alghamdi and Urban 93; Shaw, M. et al. 81). Other references of more general criteria can be found in (Klerer, M. 1991; Watt, D.A. 1990; Verhoef, D., 1995)

More recently, an extensive survey of the use of development tools in the Netherlands (Verhoef, D., 1995) revealed several deterministic project-based factors to the choice of a tool in practice :

- size of the project : (number of man-years, number of developers)
- complexity (function point analyses, number of nested iterations,...)
- projectype : maintenance; created from scratch (tailor-made) or package-implementation
- design method : waterfall (phased) <> iterative (prototyping)
- nature of the application (real-time, business, scientific)

As an illustration, some result are presented in the following table[1]:

% of all projects of the survey	small	medi-um	large	newly built	mainte nance	packages	pha-sed	itera-tive
SDW	7.9	14.7	14.9	8.8	21.2	13.7	15.9	5.4
MS Access	7.3	4.7	2.1	4.7	4.5	9.8	3.9	8.1
Oracle/CASE	5.3	4.7	2.1	5.2	1.5	2.0	3.9	6.7
Oracle Forms	6.0	5.3	0.0	4.7	6.0	2.0	3.4	7.4
Powerbuilder	4.0	4.7	4.3	6.7	1.5	2.0	3.9	5.2
Uniface	4.0	3.3	8.5	4.7	0.0	5.9	5.3	3.0
IEF	2.0	4.0	8.5	3.6	3.0	5.9	3.9	3.0
Visual Basic	3.3	2.0	4.3	3.1	1.5	2.0	1.9	3.7
CA-Clipper	4.6	1.3	0.0	4.1	0.0	0.0	3.4	1.5
rest	56.3	55.3	53.2	54.4	60.6	56.9	54.6	57.8

source : Verhoef, D., 95, p. 94

The table indicates the number of projects (in %) that use a certain tool. When interpreting this table, one should be aware that it does not claim to give any causal relationships. The reason why this tools are used cannot be deducted (from the data above) and the percentages only indicate and illustrate the actual use of tools in practical software-projects. The tools that fit our definition are : Powerbuilder, MS Access, Uniface, Visual Basic and CA-Clipper.

[1] In this survey, no distinction is made between visual development tools (as defined in this text) and other tools like CASE-tools, DBMSystems, etc. .

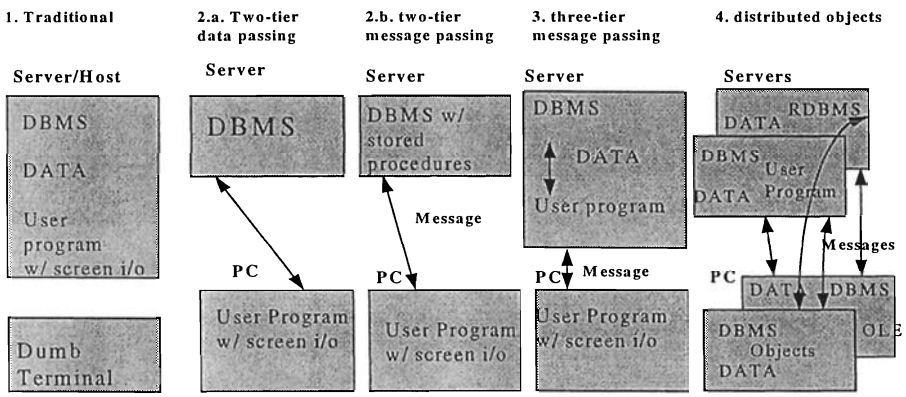
One can state that Powerbuilder and Visual Basic (general-purpose tools) are evenly used in all kinds of projects. Both tools are use in 2 to 5% of almost each type of project.). MS-access (a database tool) and Uniface (general-purpose) are unequally distributed over several kinds of projects. Access is relatively more used in small projects and package-implementation, whereas Uniface is primarily employed in large and waterfall projects. Clipper (a database tool) is primarily involved in small, newly-built and phased projects.

Further refined research is required to confirm the relationships as suggested in the Dutch request and to obtain some causal correlations...

2.3. client/server and application partitioning

2.3.1. client/server considerations

With the growth of client/server architectures and applications, it is useful to consider the way development tools allow the construction of C/S applications and provide access to databases, information files and object-servers. Although numerous definitions and descriptions of client/server applications exist, the five-part model as proposed by the Gartner Group is the most widely used basis for describing an enterprisewide client/server application. With the rise of multiple databases, improved LAN-performance and distributed object standards, the main criticism to the model is that the mainstream C/S systems decide the partition of the applications on their hardware architecture, - a mainframe connected to clients or clients and servers connected to a LAN - while it should be a business-function or application-logic driven decision. (Semich, W. June 95, p. 41; Gartner Group). As a result, the Gartner group has proposed a new multi-tier, object-based model that focuses on the development of distributed applications who are not dependent on the underlying physical location of data or application-logic.



source : Gartner Group

According to the models (the previous and the modified model) of the Gartner Group, a C/S application should meet two conditions : 1) the end-user should have a transparent access to processing algorithms (applications, programs or processes) and datasources (databases and files) (Low, G.C. , Henderson-Sellers, B. and Han, D., 1995, p. 328.) and 2) there should be a split between client-functions and server-functions that both belong to the same application logic (Kerkhof, G. 23 mart 1990, pp. 37-39).[1]

In this view, the development environment should provide applications with sufficient communications protocols to approach the servers by means of messages or RPC mechanisms that can be controlled using middleware techniques. The increasing employment of separate reusable modules of applications, goes beyond the three-tiered architecture - a split between data, application logic and presentation logic so that existing or new application can be provided with adapted and several user interfaces - and allows to develop a distributed and multi-tiered architecture (Semich, W. June 15, 95, p. 41). In this context, client applications can interface to a remote and reusable function or objectclass, bind to an instance of it and issue remote object invocations using message passing or RPC. (Maffeis, S. June 95, p. 135). As a result, time- and cost-savings are achieved not by programming faster but by consulting existing and reusable object-libraries (Harmon, P. and Taylor, D.A. 94, p. 11).

In this way, libraries or containers of classes or functions representing interactive and application-logic components can be divided in two categories : platform-dependent and platform-independent objects (Collins, D. 95, p. 441). Tools that use platform-native classes are limited to creating applications with the same look, feel and functionality as the platform they depend upon. The achievement of portability when integrating platform-independent libraries is then opposed to the loss of all the functionality and interface details of each platform's style, but accommodates heterogeneity and autonomy.

Tool-builders and vendors have recognised this trend and increasingly focus in their technological development on the usage of standard objects. A number of them have tried to construct a standard binary object and architecture independent of the used scripting code, and residing on servers that provide client access by inter-process communication (IPC) mechanisms. IBM for instance based its object-development on the System Object Model (SOM) and the distributed version DSOM. These models are based on binary objects that are language-independent and can be imported in various development tools like C++ and Smalltalk (Linthicum, October 95, p 52). In 1989, the OMG-group began with the development of the CORBA standard that should improve the multi-platform and multi-tool reusability of objects.(Benjamin, R.I. and Blunt, J., winter 93, p. 19; Harmon, P, Nov. 95 p. 85; Maffeis S. 95 p. 135). CORBA (Object management group : *The common object request*

[1] Co-operative processing is made possible by letting different systems communicate via a sequence of bits (low-level) or procedure calls (high level) in a connect-oriented or connectionless mode. (Lobelle, M. *The structure of client/server systems*; Buug, *Client/server event,* 25 sept 1991, pp. 6-12; Tanenbaum ,a.s., *Computer networks*, 1989, p. 434)

broker : Architecture and specification, 1995) specifies a standard that allows different Object Request Broker implementations to communicate over a network. The client holds an object reference that points to the objects that resides on the server-side.

Also Microsoft is promoting the OLE-model (Microsoft corporation. *OLE2 programmer's reference,* Microsoft press, 1994) which is based on OLE-objects that do not support inheritance, but are usable in numerous Windows development environments and end-user applications and seem to be gaining in importance. (Linthicum, Oct. 95, p. 52 ; Verhoef, D. 95, p. 61). A severe comparison of existing standards provides useful insights but is beyond the scope of this paper.

The modularization in different layers stimulates multi-platform development and simplifies maintenance and scalability on the condition that the distributed system can cope with fundamental problems which occur in real-life, namely partial failures, consistent ordering of events and asynchronous communication (Maffeis, S., June 95, p. 135). Development tools that allow the construction of multi-tiered systems can be evaluated according to the degree to which they can create, manipulate and/or access independent server functions or objects and prevent the above mentioned problems. More specifically, (R. van der Lans, 1996, Software Automation) compared tools that focus on application partitioning and the following table resulted :

	Composer	Forté	NatSTar	NewEra	Developer 2000	Unify
server modules outside DBMS	x	x	x	x	-	x
server modules call server modules on other servers	x	x	x	x	x	x
server modules call modules on client	?	x	x	Partially	-	-
moving server modules dynamically	x	x	x	-	-	x
call to server module does not include a location	x-	x	x	x	x	x
synchronous calls	x	x	x	x	x	x
asynchronous calls	Partially	x	?	x	-	x

Source : modification of R. van der Lans, education seminar, Software Automation 1996.

The construction of enterprise-wide applications can only be accomplished in a shared development environment, where a team of developers are able to access concurrently but transparently repositories from heterogeneous servers. Distributed development includes problems such as locking mechanisms for objects, keeping track of different versions, making sure that only compatible versions are linked together, etc. (Taylor, D.A., 92, p. 240). Although many of these functions are delegated to the servers or the underlying architecture

of available object models (such as CORBA), a client/server development tool should be considered on its ability to give an overview of the shared development in progress, and provide features that manage a project, compose the application, control version-management and (if necessary) delegate responsibilities. These particular tasks, combined with the possibility to seamlessly co-operate with distributed object-repositories will be referred to as *team development* in the rest of this paper and will be a decisive feature in the client/server dimension (cf. infra).

2.3.2. specific C/S features

In the above section, general requirements to implement application partitioning were discussed. When evaluating C/S tools, two practical issues should be considerably focused : data-access and portability.

2.3.2.1. Data-access

Although the modified Gartner model focuses on the distribution of object architectures and middleware technologies, regardless of the server-function, the interaction with databases dispersed across several servers still has a central meaning in a client/server environment. Not only is it important that a client is able to access heterogeneous databases, but it is also necessary that data can be transparently retrieved from different heterogeneous databasemodels, in such a way that the user regards the databases as one logical databasemodel. When SQL is used as a general database-request language, multiple joins and the ability to construct complex queries are the most important practical issues. (Vogt, C., June 95, pp. 217-223)

In pure OO development tools, database information has to be accessed through the use of objects, with no regard to the underlying databasemodel. If e.g. data is stored in a relational database, the relational databasemodel has to be translated in an object-model that can be implemented by a tool. Some tools possess a mapping subtool that converts relational data-items in objects (e.g. the Data Modeller in Visualworks). Development environments that are not purely OO, can access databases by way of native database API-calls or ODBC APIs that give access to the relational model (tables, rows or columns) or the file I/O system (e.g. ISAM-files and a number of desktop databases).

When evaluating development tools, they cannot be put in general categories on the basis of their database-drivers and differ individually when SQL-links and native APIs are considered.

2.3.2.2. portability

server portability

In designing a client/server architecture, a trade-off has to be considered between the degree of portability (enhanced by the use of for instance ODBC APIs or RDA (see Geiger, K. 1995, p. 66)), and the degree to which an application makes use of special server-features (like stored procedures, triggers, a particular SQL-syntax or business-logic wrapped in object-servers). Generally speaking, an increased employment of server-possibilities increases performance but also enlarges the dependency on the server. (Borland International, 1995, p. 128).

In order to obtain the appropriate proportion of portability when designing an application, it is unavoidable to control for each potential tool the access-possibilities and the way it is related to a certain type of server.

client portability

Client portability points to the degree to which a tool can produce applications that can be distributed on several different platforms. As with server portability, there is an inherent trade-off between the range of an application (number of platforms that support the program) and the potential to use particular characteristics of a certain platform.(Darling, C.B. august 95, p. 66).
There are a number of significant differences between platforms that can be rarely be surmounted or fully utilised. A first feature that can not easily be resolved is the fact that an application uses programming interfaces to operating systems that are platform-dependent (e.g. Windows-APIs). Another significant trade-off in using a cross-platform tool comes from the different interface-conventions generated in different platforms. The most evident example is the fact that several GUI standards are used in platforms like UNIX, Windows or OS/2. Finally, the communications modes managing the exchange of data between platforms is another hindering issue to portability. In a Windows-environment for instance, OLE-files, C++-calls and DLL-files are used, while a public-and-subscribe system is applied in a Macintosh-environment. (Darling, C. B., Aug. 95, p. 66) Consequently, every additional platform support means additional compromises that had to be foreseen when the tool was being constructed.

Generally speaking, tools can be divided in the following clusters :

- tools that support only one platform ;
- tools that support several platforms of the same vendor (e.g. Visual Basic can be executed on Windows 3.1(1), Windows 95 and Windows NT);

- tools that possess several versions (and compilers) to be installed on several platforms (e.g. Delphi-compilers for OS/2 and Windows);
- tools that support several platform of different vendors (e.g. Visualworks supports Windows, OS/2, Macintosh, Open Look or OSF Motif)

2.3.3. client/server dimension

The following classification results from the discussion above :

simple C/S tools	**limited C/S tools using repositories that are integrated later**	**C/S tools that are repository-driven**	**distributed C/S tools**
- native and ODBC drivers -no team development features - single-vendor platform	- native and ODBC drivers -access to and/or generation of certain platform dependent repositories (e.g. automated OLE, ,...) -limited distr. development management - cross-platform facilities	- native and ODBC drivers - access and addition to a platform dependent shared repository out of which the tool is built (e.g. Uniface's repository) - management of distributed development - cross-platform facilities	- native and ODBC drivers - generation of or access to cross-platform distributed object standards like CORBA or DSOM - management of distributed development - extended cross-platform features

Not many tools can be found yet in the category of distributed C/S tools. Forté is the most representative example, since it is compliant with CORBA standards, and provides excellent functionalities to apply application partitioning on a large scale. In the class of repository-driven tools, recently build tools like NewEra 2.0, Unify vision, Natstar and Composer are based on own platform dependent repositories and possess extended feature to distribute client and server functions. Tools like Delphi II, Visual Basic 4 or Powerbuilder 5.0 can be placed in the second column, because they possess team development features and object repositories, that are attached on or integrated in previous versions. Earlier versions of these tools should be categorised as simple C/S tools since they were not suitable for large team-development and did not allow concurrent or multiple access to central repositories.

In general, it can be stated that the environments in the left column lack the possibility to construct multi-tiered systems. Basically, these tools are not suited to build independent server functions and they do not possess a central repository that can be accessed by multi-users. They are often referred to as "first-generation C/S tools". The two columns to the right refer to tools that are suitable to program the client and server side of an application that is build with a team of developers. The tools are called "second-generation environments". The second column represents tools that used to be "first-generation" but are upgraded by adding repositories and team development features and/or linking them to existing object-libraries.

As already discussed when commenting the OO classification, this subdivision is also based on certain assumptions - the possession of a multi-user repository and team development

features are being considered decisive - and other classifications can exist when different hypotheses and other accents are put forward. Besides, some characteristics (e.g. cross-platform facilities) may not be an advantage in certain projects (e.g. more cross-platform possibilities means that less platform-specific features can be utilised).

2.4. productivity

The criteria that were treated above especially focus on the functional possibilities and limitations of a tool, they do however not express the efficiency with which an environment can execute an assignment or task. When measuring productivity of a tool, several issues and attributes can be taken into account. Productivity is a general concept that can relate to the design or coding process or the software in combination with the hardware. Improving or testing the productivity during coding or designing is in fact also testing the productivity of the personnel executing the task. (Fenton, N. E. 91 p. 262) Because this issue relates to several factors outside the development tool (such as training, intellectual capabilities, project-management, etc.), it will not be discussed in this section.

Only productivity measures concerning the tool in combination with hardware and software attributes (the client/server architecture, the performance of the server, the platform used, etc.) are regarded in this context. A critical issue in performance comparing lies in the fact that performance and efficiency analysis of software is dependent on several factors outside the programming environment - the speed and capacity of the servers, the network, the kind of transactions, etc. - that cannot easily be controlled but considerably influence the performance of the tool. Therefore, it is necessary to test performance in different modules or surroundings with particular hypotheses which are equal for different tools, so that it becomes clear which tool is most suitable in which environment.

As already stated when discussing the purpose of a tool, it is desirable to have a broad classification of projecttypes and their requirements so that different performance measures can be put forward in each project domain. Generally speaking, applications can be subdivided according to their nature : scientific, business, real-time (Verhoef, D. 95). Although each class can be further refined in subclasses (e.g. business-projects can be categorised as transaction processing systems, office automation systems, management information systems, decision support and executive information systems; real-time systems can be divided in hard and soft or static and dynamic real-time systems (Bacon, J. 93 p. 2), etc.), some frequently-used productivity measures can be applied in the three different categories.[1]

Frequently used measures include the standard benchmarks (SPEC, TPC, ...). By running a benchmark and comparing the results against a known configuration, one can potentially

[1] It is not the intention of the authors to give an exhaustive list of every possible performance measure. Only some general and frequently used measures will be shortly discussed...

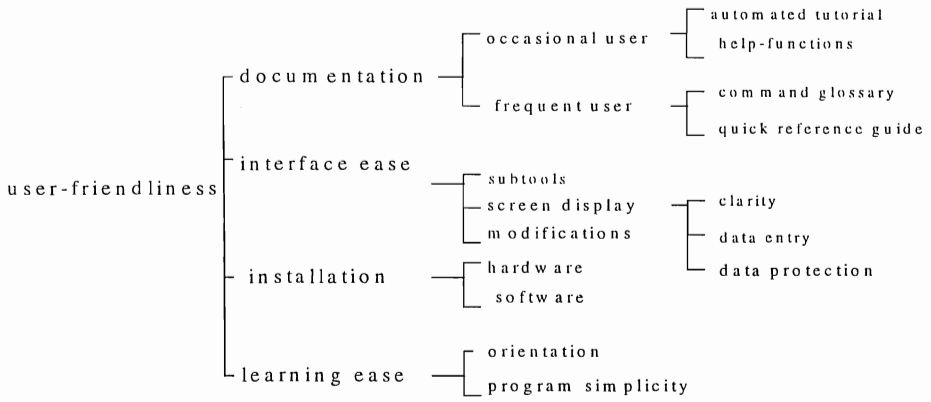
pinpoint the cause of poor performance.(Geiger, K., 95) The benchmark measure is closely related to and based on response time (waiting time and processing time per service unit) and throughput time (the number of service units per time unit). It is necessary to check which measure is dominant in a target application. If for instance a database is measured that is updated in batch format, throughput is dominant over response time, whereas response time is clearly dominant in a trading system at the stock market where timing is more decisive than the number of transactions. It is obvious that throughput and response times are not only deterministic factors in database performance. They can also be applied to measure the way a tool behaves in a more general multi-tiered distributed **C/S** environment. Since these measures largely depend on the server and network capabilities, tools should only be compared regarding the hardware capacities at hand in a concrete project...

A major element in measuring and comparing the performance of tools depends on whether the source code is compiled or interpreted. The difference in performance between a compiled and an interpreted application can be seen on two levels. Since a compiled program offers a better run-time performance but compiling is more time-intensive than interpreting, it is important to know in which development phase or in which environment a tool is to be employed. If a distinction is made between prototyping and implementation during project-development and prototyping is used frequently on a small basis, it may be desirable to use an interpreting tool (e.g. Visual basic, Smalltalk-tools,...) in prototyping and other tools in the final production phase. When compilers are tested, one should be aware that different compilers can exist for the same source-code. Different platforms can be used for the same source-code on the same or different platforms with a different productivity.

3. Human factors criteria

These criteria are used to assess the usability and ease-of-learning of a development environment. Usability focuses on the efficiency, effectiveness and user-friendliness of the interface. It helps answer questions such as : "To what degree does the environment allow a competent developer to code algorithms, easily and correctly, so they can be understood and easily adapted by other developers ?" A second question to be answered is : "Can the environment be used by non-experienced developers and is it suitable for end-user computing ?" (Howatt, J., 95 p. 38)

(Buede, M. 1992) used several evaluation criteria to test (amongst other things) the usability and performance of decision analytical software. A modification of the ideas of Buede, leads to the following scheme :



Definition of criteria :

Documentation

Occasional User :

automated tutorial : is the user taught how to use the software on-line ?

help : are there context sensitive "help" screens throughout the software ?

Is the developer provided with sufficient unambiguous error messages ?

Frequent User :

Is there a quick reference summary or a detailed glossary of commands, their formats and implications ?

Interface ease :

Subtools :are there subtools that can help the developer in particular application domains (e.g. query-builders)

Screen Display :

clarity : does the screen display promote understanding ?

data entry : is data input enhanced and controlled by the screen display ?

data protection : can the user protect his data from other users ? Is data in database or files protected ?

Modifications : can the interface be adapted to the needs of a particular developer ?

Installation :

hardware : are there any special devices required ? What are the RAM requirements of the tool, etc. ?

software : is the installation procedure automated and without errors ?

Learning Ease :

Orientation : how easy is the graphical interface to learn ?

Program Simplicity : is the software designed so that most operations (of the interface) are obvious ?

Are their powerful tools to help the structuring of programming (e.g. browsers, debuggers, ...)?

Are the error messages unambiguous and domain-relevant ?

Although it is evident to state that not only the way of coding (visually, textually (Petre, M. 95), object-oriented or not) but also informative error messages, help functions, debug- and browse tools and consistent interfaces are means to achieve usability, it is not easy to know the explicit relationships between these internal attributes - attributes that can be measured in terms of the product or resource itself (Fenton, N.E. 91 p. 42) - and the external concept of usability - attributes that can only be measured with respect to their environment (Fenton, N.E., 91 p. 43 and 249). Following the ideas of (Fenton, N.E. 1991), and (Gilb, T. 1987), the

actual measure has to be decided by a particular 'user' according to the particular type of product and leads to the following measurable attributes :

Entry level : years of experience with similar class of applications
Learnability : speed of learning, e.g. hours of learning before independent use is possible
Handling ability : e.g. speed of working when trained and/or errors made when working at normal speed

The usability dimension should be interpreted as a ranking that represents the external concept of usability. As a result, the answers to questions about usability should be stated in an open way and cannot be limited to a priori defined possibilities...

4. Miscellaneous

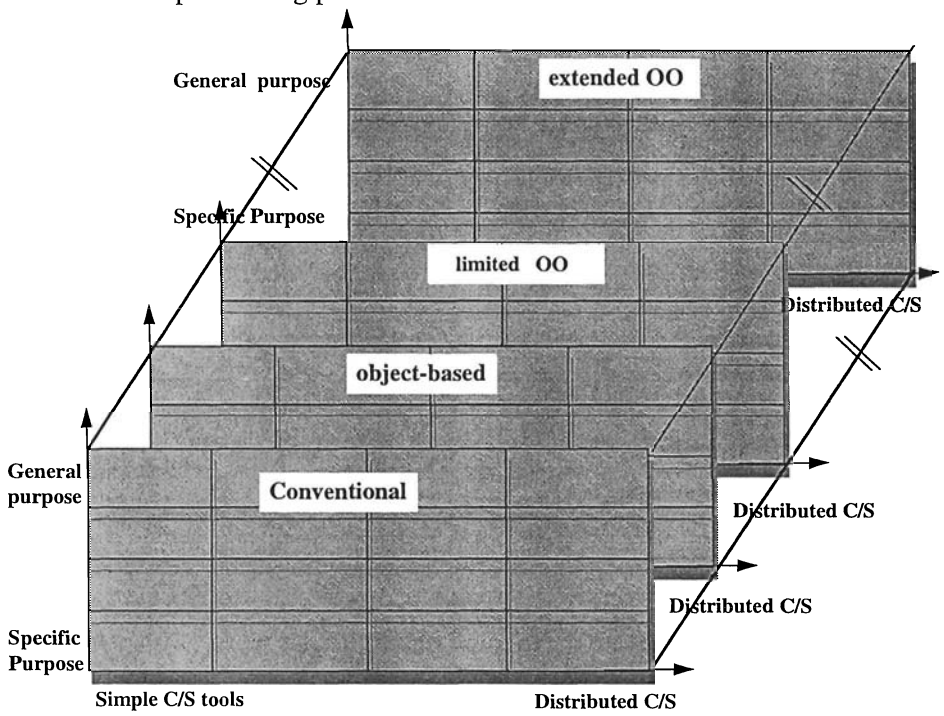
According to (Howatt J. 95), software engineers often focus on the following features when selecting tools :
- they choose the tool they have used in the past (or an upgraded version) and of which a huge quantity of code is already available;
- they select the product that is wide-spread and possesses a large market-share (a lot of experience is available and the tool can be considered as mature);
- Considerations of contractual nature can be decisive.

It is obvious that producers with a reliable and continuous service should be preferred when critical or complex applications are to be built. In (Attachmate Corporation, April 95, p. s-23), the latter issue is confirmed and it is argued that service- and consulting costs are the biggest hidden costs in the management of a **C/S** project. Also in the Dutch survey (Verhoef, D., 95 pp. 68, 77 and 82), criteria such as the continuity, the growth potential of the tool and installation and maintenance support of the vendor are indicated as criteria that can be used to investigate the satisfaction of the interviewee. Among these, the continuity of the supplier seemed to be allocated a lot of importance according to the interviewees. Consequently, it is important to select a reliable producer that can support their clients in a technical, economically and organisational way.

Other important aspects to consider relate to costs of purchase, installation, maintenance, training and the effect the tool has on the organisational structure.

5. A general framework

In the following figure, several dimensions discussed above are brought together so that a general basic framework results, in which developments tools can be positioned. In the scheme below, the ordinal factors (C/S and purpose) are represented in two-dimensional planes, and the nominal factor (object orientedness) is shown in depth. Usability, vendor support and training (ordinal factors) are left out, because they depend on the users and supplier and cannot be measured on a general reliable scale. Productivity measures can be included when positioning particular tools.



It is important to remark that the three global dimensions are independent variables which are not per definition related to each other. In this respect it can e.g. not be stated that a tool is well suited in a C/S environment, because it is object-oriented or general purpose or vice versa. Another example relates to the usability of tools. Tools can offer general functionalities and scalability opportunities, but demand extensive learning. Criteria may have different importance to different projects, and aggregation of the dimensions should therefore be done with caution.

Nevertheless, some general remarks and correlations can be put forward when applying the model on a global basis. In general, it can be stated that tools with a general purpose allow re-use for additional projects, but demand more training. A very specific tool can save initial investments but decreases re-use opportunities. The decision maker has to decide whether there are trade-offs to be made. Moreover, it can be concluded that the characteristics of the object-oriented paradigm lend themselves to the purpose and requirements of application

partitioning and are also suited to model complex enterprise-wide applications in a diversity of environments. It is then not surprising to note that second generation C/S tools, which are the cornerstones of a growing market segment are more likely to possess object-oriented features. Compared to first generation C/S tools, these environments can tackle a large range of enterprise-wide applications, but the complexity and thus the training cost and entry level can also be expected to grow. Re-use of component and class repositories, the integration of graphical (specific-purpose) subtools, visual coding and standard user interfaces are currently the solution to counterbalance the growing demands of complexity that constructors of tools are faced with.

By way of illustration, three development tools will be roughly positioned in the framework above : Visualage Team 2.0 for Smalltalk, Visual Basic 4.0 and Delphi II.
Starting with the object-oriented dimension, it can be concluded that Delphi and Visualage are the only true OO-tools. Visualage for Smalltalk supports dynamically-typed dynamic binding, whereas Delphi provides statically-typed dynamic binding. Both tools are provided with single inheritance but no generic classes. Visual basic is clearly object based (it lacks inheritance and polymorphism).

All three tools are general purpose, but Visual Basic is less generally applicable. Delphi is fully OO and offers possibilities to create DLL-files and extended memory allocation functions and Smalltalk (also fully OO) offers extensions to COBOL and C and can (amongst other reasons) therefore be considered more general-purpose than Visual Basic.

Contrary to Visual Basic and Delphi, Visualage (Team) is a truly distributed C/S tool that supports concurrent team programming based on a LAN repository and allows version management, source and object code tracking and configuration management. Visualage 2.0 for Smalltalk is not (yet) CORBA compliant but supports the DSOM architecture. Delphi II and Visual Basic 4.0 should be placed in the second column of the C/S dimension (cf. supra) since they support the creation of and access to automated OLE objects (limited to Windows platforms) and have added team development features. (Both issues were not present in their previous versions). Visual Basic is only portable in Windows platforms, whereas Delphi possesses compilers to port code to Windows and OS/2 systems. Visualage is more portable and can be ported to OS/2, Windows and Motif platforms. The three tools possess several relational database drivers. Delphi possesses native drivers to access Paradox and Dbase databases and it can access SQL server databases (like Oracle, Informix,...), whereas Visual Basic offers native drivers to Microsoft databases (especially MS-Access). Both tools provide access to other databases using ODBC drivers. Visualage can interface to DB/2, Sybase SQL servers, Microsoft SQL servers and Oracle databases.

As far as productivity is concerned, Delphi is the only tool that provides a compiler and has therefore relatively more performance at run-time. Considering the learning curve, Object Pascal (Delphi) and Smalltalk (Visualage) are truly object-oriented and demand in general a larger learning period than Visual Basic. One should always keep in mind though that the

learning curve is dependent on the experience of the target developers-group. In the three tools, graphical subtools (GUI-builders, browsers, debuggers and database tools) are integrated with the underlying scripting language. Visualage has adopted techniques from visual programming and allows developers to construct application logic by wiring icons and other graphical representations. Furthermore, the three tools are supported by three large and reliable constructors - IBM, Microsoft and Borland - and experience by other user-groups is present. (Although Visual Basic seems to be the most used tool).

The table below can be used as a general guide to examine certain tools in more detail. Depending on the target application, some features will have more or less weight, and additional requirements may have to be added. After the following requirements have been pin-pointed though, the tools can be positioned in the dimensions presented in this paper and more detailed requirements can be put forward...

OBJECT-ORIENTEDNESS	
Is abstraction possible in the language ?	instance variables instance methods class variables class methods
Is encapsulation possible ?	of variables(public/private) of methods(public/private)
Is a hierarchy of classes possible ?	inheritance(single/multiple) metaclasses generic units
Is it a typed language ?	strongly typed (yes/no) binding (static/dynamic) polymorphism(single/multiple)
concurrency	multitasking (yes, no, indirectly)
persistence	persistent objects
PURPOSE	
What is the scripting language of the tool ?	specific-purpose 4GL non-procedural 4GL procedural 4GL or 3GL
To what degree is visual programming possible ?	is it optional ? is it constraining the developer to certain areas ?
Can standard-code be generated that can be used by other tools ?	exportable libraries standard-objects links to libraries of other languages links to CASE tools or methodologies ?
Are subtools available that are necessary for the target project ?	database-tools report-generators expertsystems browsers debuggers ...
Are their basic functions that can be used in the application logic ? Are these functions limited to a certain application-domain ?	mathematical functions statistical functions functions to improve structured programming (e.g. case-structures) ...
Can functions be generated that do not contribute to the application logic, but make the application run on the hardware ?	autonomous executable-files the definition of own messages creation of platform-dependent APIs dynamic memory manipulation
CLIENT/SERVER	
data-access	
databases drivers ?	native APIs ODBC drivers other SQL-links
Is it possible to connect to middleware tools	embedded middleware links to autonomous middleware packages
application partitioning	
can server functions be generated or manipulated ?	stored procedures or triggers can be manipulated server modules outside DBMS are possible server modules call server modules on other servers server modules call modules on client moving server modules dynamically call to server module does not include a location synchronous calls and/or asynchronous calls

team-development	
remotely accessible object repositories ?	platform-dependent platform-independent
concurrency control ?	two-phase locking other mechanisms
version-control ?	
Is there any co-operation between different development sites (of one project) and common repositories ?	
project-management ?	e.g. a tool that indicates which sub-application is responsible for integrating the whole ; a tool that compares differences in time-budgets, etc.
portability	
standalone or server-based ?	
cross-platform ?	the same vendor compilers of different vendors portable to different vendors (by using adapted tools and compilers)
PERFORMANCE	
Does the tool meet benchmarks for the target C/S configuration ?	e.g. the TPC-B benchmark, the SPEC benchmarks
Is response time or throughput time dominant in the target project ?	
Is the tool interpreted compiled or are both possibilities optional ?	
Is an optimised compiler used ?	compile time compile and link time execution time object code size execution size
HUMAN FACTORS CRITERIA	
Are the subtools user-friendly and usable in a consistent and graphical way ?	handling ability , reliability and entry level for browsers debuggers text editors ....
Is there sufficient context-sensitive help ?	
Can errors be easily detected ?	
What is the learning period for the tool ?	entry level learnability
MISCELLANEOUS	
Is there any experience present in the organisation ? Do bodies of users exist (elsewhere) ?	
Is the supplier reliable ?	Can the supplier guaranty continuity in service and material ? Is the constructor's organisation stable ? installation support ? implementation support ?
Is the product manageable ?	is the software readily available ? growth potentials ? scaleable ? Is there a possibility to obtain training and education ?
various costs	software and additional hardware maintenance education and training consultancy support organisational modifications

Conclusion

The proliferation of client/server architectures, object-oriented system development and visual, user-friendly programming, has resulted in an increasing involvement of PC's and workstations in traditional data processing and automation areas. Therefore, more and more professional development products appear on the market to develop application that can make optimal use of the graphical user interface and access remote databases or object repositories via a network. The differences and potentials of this new generation of products is unclear and changing rapidly. Therefore, it is necessary to present a general framework to make the comparison of existing tools and even the construction of future tools a more informed decision.

Since numerous criteria exist that can be taken into account, the criteria have a divergent nature (technical, as well as human and economic factors should be considered), and the importance of the criteria is dependent on the application at hand, it is a complex problem to find generally applicable dimensions that allow to set up an all-embracing classification. Nevertheless, we have given a scheme that offers a usable and clear positioning-framework that integrates ordinal with nominal measures in a multi-dimensional whole, on the basis of which development environments can be selected. Moreover, it is a solid basis that can be refined and completed through further research.

Different sets of criteria were developed. They relate to software-engineering, human factors and cost and training issues. When discussing software-engineering factors, the degree of object-orientedness, the purpose, the client/server capabilities and the performance of tools were put forward. Human factors criteria are important elements that can give an answer to questions like "How user-friendly is the tool" and "How easily can it be learned ?". Mostly, these issues depend on the training and motivation of the developers using the tool and can only be answered by particular usergroups using a particular product. Criteria that focus on the consultancy, training and installation costs of a tool are to a strong degree dependent on the supplier, who should be chosen with care so that a continuous and stable support is assured.

However, the selection of an environment should also be accompanied by an accurate problem description from which the required characteristics of a tool can be derived. By choosing general purpose tools with many possibilities, a re-use of the tool for additional development projects is possible, but software training is more demanding. By choosing a tool which is very specific for the target application, initial investments can be saved, but it is not sure that the tool will be re-usable. A detailed survey of different tools using the proposed criteria and the requirements of the application to be developed will finally lead to the best choice.

References

Agha, G.A. and Wegner, P., *Research directions in object-oriented programming.* MIT Press Cambridge, Mass., 1993, 532 pp.

Alghandi, J. and Urban, J., Comparing and assessing programming languages : Basis for a qualitative methodology. In *Proc. 1993 Software Applications Conference,* ACM Inc. 1993 , pp.1222-*229.*

Attachmate Corporation, How to revitalize host systems for client/server computing today & tomorrow, in *Datamation*, April 1, 1995, pp. s-2- s-24.

Beek v.d. G. , Het gebruik van een vierde generatietaal, in *Informatie*, jg 29 nr. 10, pp. 861-956.

Benjamin, R.I. and Blunt, J., Critical IT issues : The next then Years, In *Sloan Management Review*, winter 1993 , pp.7-19.

Bodker S., *Through the interface : A human activity approach to user interface design,* Lawrence Erlbaum Associates, Inc., 1991.

Booch, G., *Object-oriented analysis and design.* The Benjamin/coummings publishing company, Inc., California, Massachusetts,..., 1994.

Borland International, Inc., *Databases Application Developer's Guide*, Borland International, Inc., 1995, 200 pp.

Burnett, M. , *Visual object-oriented programming : concepts and environments*, Prentice-hall Englewood Cliffs, 1995, 274 pp.

Cardelli, L. and Wegner, P. On understanding Types, Data Abstraction and Polymorphism. in *ACM Computing Surveys* vol. 174(4), 1985.

Collins, D., *Designing of object-oriented user interfaces.* The benjamin/cummings company, INc.,New York/Amsterdam/Bonn/..., 1995.

Daring, C.B., Immortalize your apps, in *Datamation*, august 1, 1995, pp. 65-74.

Garfinkel, S.L. and Mahoney, M.K., *Nextstep programming. Step one : object-oriented applications.*, Springer New York, New York, 1993, 631 pp.

Geiger, K., *Inside ODBC,* Microsoft Press, Washington, 1995, 482 pp.

Gilb, T., *Principles of Software Engineering Management*, Addison Wesley, 1987.

Glinert, P.E., *Visual Programming Environments,* IEEE Computer Society Los Alamitas, California, 1990.

Harmon, P., Object-oriented AI : A commercial Perspective, in *Communications of the ACM*, 1995, vol. 38 no. 11, pp. 80-86.

Harmon, P., and Taylor, D.A., *Objects in action,* Commercial applications of object-oriented technologies. Addison-Wesley publishing company, Massachusetts, California,..., 1993.

Hopkins, T. and Horan, B., *Smallltalk : an introduction to application development using visualworks,* Prentice-Hall Englewood Cliff, 1995, 408 pp.

Howatt, J., A project-based approach to programming language evaluation. in *ACM SIGPLAn Notices, 30(nor. 7), 1995,* pp.37-40.

Janse, H. J. and Achterberg, J., Invloed van vierde generatie software op ontwikkelingsfuncties, In *Informatie,* jg 29 nor 11, pp. 957 -1044.

Kerkhof, G., Van master/slave naar client/server model, In *Computable*, 23 mart 1990, 23 (12), pp. 37-39.

Khoshafian, S., e.al., *Intelligent offices : object-oriented multi-media information management in client/server architectures*, John Wiley & Sons, Inc., New York/Chichester, 1992.

Klerer, M., *Design of Very High-level computer languages : A User-Oriented Approach*, McGrw-Hill, 2nd edition, 1991.

Linthicum, D.S., The Object Revolution, in *DBMS*, Oct. 1995, pp. 46-52.

Low, G.C., Henderson-Sellers, B. and Han, D., Comparison of object-oriented and traditional systems development issues in distributed environments, in *Information & management,* Elsevier Science B.B., 28(1995), pp. 327-340.

Maffeis, S., Adding Group Communication and Fault-Tolerance to CORBA, in *Coots*, Monterey, California, June 26-29, 1995, pp.135-146.

Szczur, M.R. and Sheppard, S.B., TAE-Plus: Trasportable Applications Environment Plus : A User Inerface Development Environment, in *Acm Transcations on Information Systems*

Shaw, M., Almes G.T., Newcomer, J.M., Reid, B.K. and Wulf, W.A., *A comparison of programming languages for software engineering,* Software-Practice and experience, 11, 1981, pp. 1-52..

Shu, P.C., *Visual Programming.* Van Nostrand Reinhold, New York, 1988, 315 pp.

Semich, J.W., Client/server unchained : Finally, Hardware Independence, in *Datamation*, june 15, 1995, pp. 40-45.

Stroustrup, B. *The C++ programming language*. Addison-Wesley Reading, s.p., 1991, 680 pp.

Taylor, D.A., *Object-Oriented information systems.* Planning and implementation, John Wiley and Sohns, Inc., New York/Chichester/Brisbane, 1992, 357 pp.

Templ, J., A systematic approach to muliple Inheritance Implementation, In *ACM SIGPLAN Notices, april 4, 1993, pp. 61-66.*

Urlock, Z., *An overview of the Delphi 2.0 Optimizing Native Code Compiler for Windows 95 and NT,* Borland International, S.L., 1996, 28 pp.

Verhoef,D., *Toolvision: Nationaal onderzoek naar het g*

Vogt, C., Simple SQL, in *Byte,* June 1995, pp. 217-223.

Watt, D.A., *Programming Language Concepts and Paradigms*, Prentice-hall Inernational, 1990.

Welke, R.J., The shifting software development paradigm. In *Database,* vol. 25 nor. 4, 1994.

Winblad, A.L., Samuel D. Edwards and King, D.R., *object-oriented software*. Addison Wesley publishing company, inc., Massachusetts, California, new York, 1990.