CASE Tools and Software Factories

Erik G. Nilsson

Center for Industrial Research (SI)
Forskningsveien 1, P.O. Box 124, Blindern
N - 0314 Oslo 1
Norway

Abstract

The present paper addresses CASE tools as they are today, and what we believe will be the next generation CASE tools - Software Factories (or IPSEs - Integrated Project Support Environment, or ISDEs - Integrated Software Development Environments).

The paper first gives a definition of CASE tools, *and investigates strong and week sides in todays CASE tools. One of the major drawback of todays CASE tools is the lack of integration between specification tools and construction tools. The paper describes four different techniques for such an integration, one of them being usage of a Software Factory architecture.*

Then the paper gives a definition of Software Factories - *integrated software development environments - , investigates who makes Software Factories, what is achieved by using Software Factories and describes one particular Software Factory - Eureka Software Factory (ESF).*

ESF is a multinational, european research and development project that is developing a framework for integrated software development environments, and special environments for special application areas. ESF has a deep foundation in the industry. The paper describes the ESF project, the technical architecture used, and results so far.

Introduction

CASE tools have gathered much interest recent years. From being quite simple drawing tools, powerful ones with wide functionality have emerged. The interest in this kind of tools (and the tools themselves) come(s) mainly from the industry, not from academic circles. Thus, the topic CASE tools has become an area discussed among practitioners, not by academics. Because of this, there are not many general publications (at conferences or in scientific journals) treating CASE tools as a phenomena [HEDQ88, CASE89, BUBE88, NILS88, ROEV89b], but there exists quite a number of articles in computer magazines

(like BYTE, Datamation, Computerworld, etc.), both on specific tools and on the field in general [BYTE89, COWO87, COWO88a, COWO88b, ELEC87, IEEE88, DAMA88, INWO88]. One of the reasons for the large interest among practitioners, is the huge amounts of available tools.

Within the field *Software Factories,* the situation is the other way around. There has been quite a lot of research done on integrated software development environments, but few (if any at all) available products exist. Because of this, the knowledge of and interest in such environments is not big among practitioners.

CASE Tools

In this section we take a closer look at CASE tools as they are today. We give a definition of our understanding of what a CASE tool is, we investigate week and strong sides, and we describe how an ideal CASE tool should be, as view from the user (i.e. primarily software developers). As the number of commercial available CASE tools are very high (at least three digits), we make this presentation quite general, without any references to specific tools. Because new tools are emerging all the time, making such references complete would be meaningless. Readers who are interested in specific tools should read[ROEV89b, HEDQ88 and COWO88a].

What is a CASE tool?

The term *CASE tool* has become a buzzword the last years. Linguisticly, the term *Computer Aided Software Engineering* could cover almost any tool that supports software engineering (even a compiler), and a lot of vendors indeed use the term in a very wide sense to be able to put the *CASE tool* label on their products.

The term is also used quite inhomogeneously by different people. From the beginning of *CASE tools,* the term denoted tools that support specification work, i.e. systems analysis and to some extent systems design. As *CASE tools* grew more popular, the meaning of the term "expanded" to cover tools for the all aspects of systems design, and also systems construction. This expansion of meaning is consistent with the semantic meaning of the word, but have also made it less precise.

The need for more precise concepts has lead to different prefixes and suffixes to CASE, *upper, middle, lower, toolkit, workbench* and *integrated* being the most commonly used. Some refer to *upper CASE* as tools supporting systems analysis, while *lower CASE* support all activities from systems design through systems construction and even systems installation. Others restrict *upper CASE* to the phases *before* systems analysis, i.e. corporate planning, strategy study, breakdown of goals, objectives, etc., and use *middle CASE* to denote tools supporting analysis and design, while *lower CASE* is tools supporting systems construction. A third usage, is to make the distinction between *graphical (or diagrammatic) tools* and *code generators, 4. gen. systems, and other non-graphical tools.* In this case, *upper CASE* is the diagrammatic tools, and *lower CASE* is the non-diagrammatic tools. Dividing it this way, puts the border between upper and lower CASE somewhere in the systems design phase. In this paper, we use this last definition of *upper* and *lower CASE* (we do not use *middle CASE).*

Some people make a distinction between *CASE toolkits,* supporting one phase of software development, and *CASE workbenches,* giving support across the software development process. This distinction is often quite fuzzy, and we do not use it in this paper.

Integrated CASE, or *I-CASE* is also used differently by different people. Some use it to refer to tools supporting all activities in software engineering in an integrated manner. Others use it to denote the integration mechanism, i.e. either a common data base (repository, data dictionary, encyclopedia, ...) or a tool (e.g. a software backplane). A third usage is tools that support a specific software development methodology (or life-cycle model). The term *I-CASE* is sometimes used synonymously with *IPSE - Integrated Project Support Environment* (by some people also used to denote Integrated Programming and Software Engineering). We will not use the term *I-CASE,* but rather use the term *IPSE, Software Factory* or *Integrated Software Development Environment (ISDE),* which we find more precise.

With this definition, upper CASE tools covers tools that support techniques like conceptual data modelling (e.g. Entity Relationship (ER) models, entity life-cycle history), structured analysis (e.g. data-flow diagrams, control flow diagrams, activity decomposition diagrams, activity dependency diagrams, state transition diagrams and flow-charts), and techniques like action diagrams, minispecs, pseudo code specification, prototyping, etc. Lower CASE include traditional 4. generation tools (i.e. application generators with tools for dialog design, screen painting, report layout specification, automatic data base design, code generation, etc.). Most CASE tools have a built-in data dictionary (or repository). While most CASE tools are specialized for business application (commercial data processing), some CASE tools for other application areas (like real time systems) have emerged the last years.

In this section we consider both upper and lower CASE tools, but we have the emphasis on upper CASE.

What is good about CASE tools?

Present CASE tools have many strong sides. The main benefit in upper CASE tools is the ability to support the human activities in specifying requirement, i.e. they give aids for documenting results from the requirement specification process. The people performing the specification work no longer need to use pen and paper in drawing their diagrams (and *edit* them), nor to develop their own data dictionaries. In that way they can use more time on creative work, not on tedious drawing work. Even more important, as a result of using these tools, the diagrams and the data dictionary are integrated. Many such tools have modules for automatic generation of documentation of the system being developed. In some tools there are also an integration with word processors, which makes documentation even more convenient.

Some tools also support a simple consistency check between different parts of the specification (different diagrams, information in the data dictionaries, etc.), e.g. between a data flow diagram and an ER diagram and their connected data dictionary specifications. Such a consistency check is f.ex. controlling that the content of data flows and data stores in the data flow diagram are specified in the ER diagram.

In addition, most upper CASE tools have facilities for code generation. These are often restricted to the ability of automatic generation of data base schemas (usually expressed in SQL), but some also support limited program code generation. Some upper CASE tools also offers prototyping tools, but it is seldom possible to use (parts of) the prototype (e.g. screen layouts) outside the CASE tool.

An organizational benefit from the use of CASE tools is a standardization of methodologies and techniques used in the organization.

Another feature supported by some upper CASE tools is the ability to do automatic normalization, but we do not consider this facility as being very important. We feel that doing a good job specifying the conceptual data model makes normalization unnecessary. Normalization is in fact only a control. It can never *produce* good data models or data bases.

The lower CASE tools usually have much more powerful code generation facilities than the upper CASE tools. This is partly due to the kind of information put into the tools, especially the level of detail. As more details are fed into lower CASE tools, code generation is more powerful. The lower CASE tools also usually have more powerful tools for prototyping and application development (e.g. report and screen layout specification), and the application development tools produce executable code, i.e. are integrated with the program implementation tools.

What is bad about CASE tools?

Unfortunately, CASE tools also have a number of weak sides. One drawback in most of todays upper CASE tools is that the users are forced to use the techniques / methodologies supported by the tools they use. This applies both to the visual appearance of the diagrams and the rules used in the techniques. This problem is sometimes referred to as *tool imperialism*. We do not claim that it is wrong to use *one* specific technique / methodology in an organization, but the choice should be done by the organization, not by the tool vendor.

Another problem is that some upper CASE tools are cleverly implemented tools based on bad techniques. The people developing the CASE tools sometimes have higher qualifications in implementing graphical user interfaces than they have in requirement specification techniques. Therefore, the choice of the underlying techniques are in some cases rather arbitrary. This is not as big a problem today as it was in the young days of CASE, but it still does occur.

Although most upper CASE tools have quite good user interfaces, especially if one compares them with other tools supporting systems development (e.g. most lower CASE tools), the user interface could be better in many tools. In our opinion, a common weakness in the user interface is too much use of *modes*. This is especially the case in MS-DOS and MS-Windows based tools (which many CASE tools are). One example of unfortunate use of modes is when you have to perform a special command (putting the tool in a special mode) for operations like moving objects, re-sizing objects and deleting objects. Such operations should be performed either by direct manipulation of the objects, or by selecting the objects and then performing an operation (preferably a keyboard shortcut, e.g. the rub-out key for deletion).

Of course, there is a trade-off when to use modes and not. Usage of modes f.ex. has the benefit of *open end* operations. It is also important to have *general* commands, i.e. dividing the operation and the operand. This minimizes the number of commands. In some CASE tools this is not the case, and one may experience long lists of commands like *create entity, create relationship, ..., move entity, move relationship, ...*, etc.

Another aspect of user interface is the choice of how the tools should enforce rules in the techniques. Many tools let the users do almost anything, and performs a check of the results afterwards. Other tools restrict the users while using the tools, making such a check - which may cause a lot of work for the user - superfluous. A good rule when designing user interfaces is that the user shall not be allowed to perform illegal operations.

Yet another aspect of user interface is the integration between diagrams and textual information in the upper CASE tools. This integration is often done by having a possibility to "explode" elements in the diagram, and thereby getting a screen layout for specifying detailed information. This solution is often due to lack of windowing capabilities in the CASE tools, and use of small screens on the monitors of the PCs running the tools. Combining diagrams and detailed information in compound screen layouts often yield a better user interface.

In addition to the bad user interface (usually terminal and character based with no use of graphics and direct manipulation), lower CASE tools also requires quite a lot of information to be supported by the user to be able to generate code successfully. The bad user interface is often due to the fact that many such tools are mainframe or mini-computer based. Therefore, they are often quite expensive, especially compared to upper CASE.

Most CASE tools (both upper and lower) are directed at developing new systems. Few support what most systems engineers spend most of their time doing, namely maintenance.

The major problem, though, is that the CASE tools are "islands", i.e. they fulfill their purpose in *some part* of the systems development process. As mentioned earlier, the upper CASE tools have some code generation facilities, and the lower CASE tools have quite good code generation capabilities. But unfortunately, that is not enough. As the development evolves, the information stored in the CASE tools are not changed to reflect the *actual* specification and implementation. The reason for this is mainly that the upper CASE tools have very little *connection* to the lower CASE tools, and even less to the program implementation tools, i.e. they can produce results for the lower tools, but they can not get changed specification in return. There is also a problem that upper and lower CASE tools sometimes have overlapping domains, so that one may have redundant specifications. This topic will be discussed in more detail below section *Integration between CASE tools and implementation tools.*

How should an ideal CASE tools be?

In our opinion, the ideal CASE tool should have a lot of features. The most important ones - in addition to those listed above under the heading *What is good about CASE tools?* - are listed below:

Upper CASE tools should be independent of techniques, i.e. they should support a number of techniques, also "dialects" of the same main technique. It should also be possible to *tailor*

the tools, at least in four different areas:

(i) It should be possible to tailor the graphical interface for a technique, i.e. to change symbols, colors, the appearance of arrows, etc.

(ii) It should be possible to tailor the *meta model,* i.e. the schema of the data dictionary. If the user wants to document an additional aspect for every entity type, he should be able to add an attribute in the data dictionary to handle that aspect.

(iii) It should be possible to tailor the rules used in the techniques, e.g. to specify whether a relationship could have connected attributes, or whether n-ary relationships are allowed.

(iv) It should be possible to tailor the connection between different models, e.g. to specify which connections that should exist between data flows and data stores on a data flow diagram and entities and attributes in an ER diagram.

All these tailoring facilities should be easy to use, i.e. the upper CASE tools should have special *modules* for tailoring. These tailoring features (at least (i), (ii) and (iii)) are available in some existing CASE tools.

Lower CASE tools should have more graphical user interfaces, and should be based more on manipulation of objects in the systems to be developed. Usage of direct manipulation when f.ex. specifying screen layouts, would make such tools easier to use. This indicates that such tools should run on PCs or work stations, or at least have an interface from such machines. More general, lower CASE tools should run on the same hardware platform as the upper CASE tools, or even more preferable, the CASE tools should be portable.

Both upper and lower CASE tools should support reuse. Not only reuse of code, but also reuse of specifications and design.

All CASE tools should be integrated with implementation tools. (By *implementation tools* we mean tools for either writing code (editors etc.), or manipulation code (compilers, debuggers, test tools, etc.).) It is also important that the integration is done in such a manner that changes in the implementation more or less automatically are reflected in the CASE tool. In an environment with a 4. gen. tool that operates on the "model level", e.g. with concepts from the ER approach and abilities to produce all code automatically (such tools exist), this integration is achievable.

The ultimate goal would be that the distinction between upper CASE, lower CASE and program implementation tools become transparent. The user should operate in an integrated environment, which support all tasks in the software engineering process (also administrative tasks like project management), and the facilities offered by the environment reflect different tasks for the software engineer, not different underlying tools. Such environments are often referred to as IPSEs (Integrated Project Support Environment) or Software Factories. We denote such environment *Integrated Software Development Environments.* But before we take a closer look at such environments, we investigate different models for integrating different CASE tools and implementation tools.

Integration Between Specification Tools and Construction Tools

In this section we take a closer look at four different models for integration between specification tools (mainly upper CASE) and program construction tools (lower CASE and implementation tools). Looking at these four models, it is important to bear in mind the goal of the integration, namely that the user (i.e. the systems developer) should have a work environment where the distinction between the different tools is transparent, that the tools should support the different *tasks* the systems developer performs, and of course that there should be some sort for integration also on the data level [NILS90].

General remarks

One may claim that because specification tools and construction tools cover different phases and activities in a software development effort, there is no need for an integration on the *user* level. We claim that although there are separate phases, an integration on the user level is necessary. The reason for this is that the specification changes in the construction phase, often as a result of trade-offs done to make the implementation effort obtainable. Such changes in the specification should also be reflected in the documentation of the specification, i.e. in the different CASE tools.

The integration between CASE tools and implementation tools must be handled in a special manner in tools for automatic code generation based on specifications on "model level". As long as the "executable" specifications are expressed in terms of the models in the CASE tools, the integration with implementation tools become both a necessity and much easier. A necessity because it is the *specifications* that are altered when a change is issued (not the generated code), and much easier because one only operates on the specification level, and therefore do not need to bother much about the implementation tools.

Independent of which of the models below that are chosen, there is a need for *semantic integration* when different tools support the same problem domain. By semantic integration we mean that the different tools that are integrated have a conformal use of concepts. This is of extreme importance if different tools use the same data store, and is also important if they have different data stores, but common user interface. In the latter case, semantic inconsistencies could be handled by the integration mechanism in the user interface.

Model I: Integration using import and export

Integrating specification tools and construction tools by using import and export is the most common solution today. Most upper CASE tools have some code generation facilities, usually restricted to generation of data base schemas. I.e. this integration is usually a one-way data transfer *from* the CASE tool *to* the implementation tool. (The integration between lower CASE tools and implementation tools is often tighter). In addition to import and export facilities connected to the tools, various users have developed their own import and export program ("bridges"). Some of these have been distributed as freeware or shareware.

Very few (but some) upper CASE tools have the ability to transfer data the other way. (See figure 1 (next page)). The integration the other way could either be done by letting the implementation tool have a facility for producing input to the specification tools (on a

special format) or by letting the specification tool have a feature for importing some data format that is easy to produce from the implementation tool. In the latter case, changes is the data base structure could be expressed in SQL, which the specification tool translated back to a conceptual data model, and merged with its existing conceptual data model. A few existing CASE tools have the possibility to import data from implementation tools (usually from the same vendor).

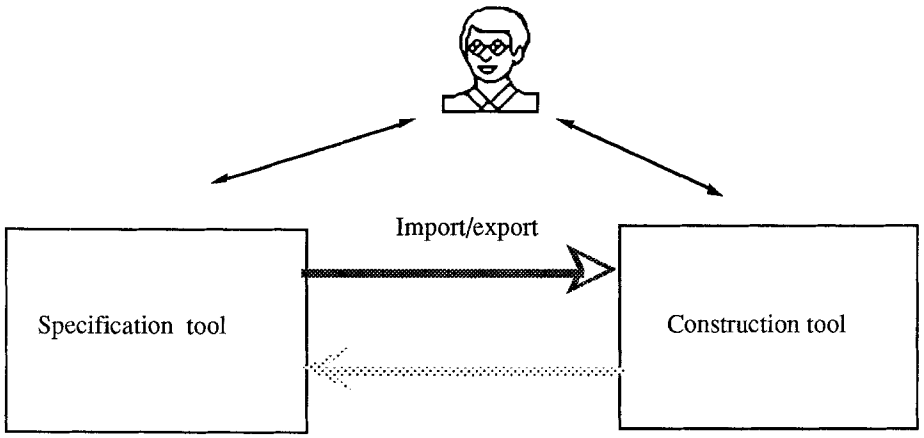

Figure 1 - How integration between specification tools and construction tools usually is done today

The process of translating from implementation level to specification level is sometimes referred to as *reverse engineering* [CASE88, NILS85], and is a more general way of transferring information from implementation tools to CASE tools (both upper and lower).

The advantage of this way of integrating specification tools and implementation tools is that it can be implemented without issuing big changes in the tools. The data transfer and data conversion can be handled by a separate tool (an "agent").

The disadvantages are many. It is often difficult to transfer data *to* the specification tools. This feature is essential if the integration is to be successful. The integration is in many extents "manual", i.e. one has to do special tasks and to run special programs to integrate. Integration should not be a special task that has to be performed every time something changes, integration should be done automatically. And as the integration task has to be manually triggered, there is a fair chance that one forgets to do it, or omit to do it because it is time consuming. The last disadvantage is that there is no user integration.

Model II: Integration using common data base

One way of making the integration between specification tools and implementation tools tighter is by using a common data base where the information is stored. In that way, data from the specification tool can be used by the implementation tool and vice versa (see figure 2 (next page)). This model requires either that one tool is able to read data stored by an other tool, or that one tools is able to write in the data store of an other tool. This kind of

integration can also be used to interchange information between different CASE tools (upper and lower).

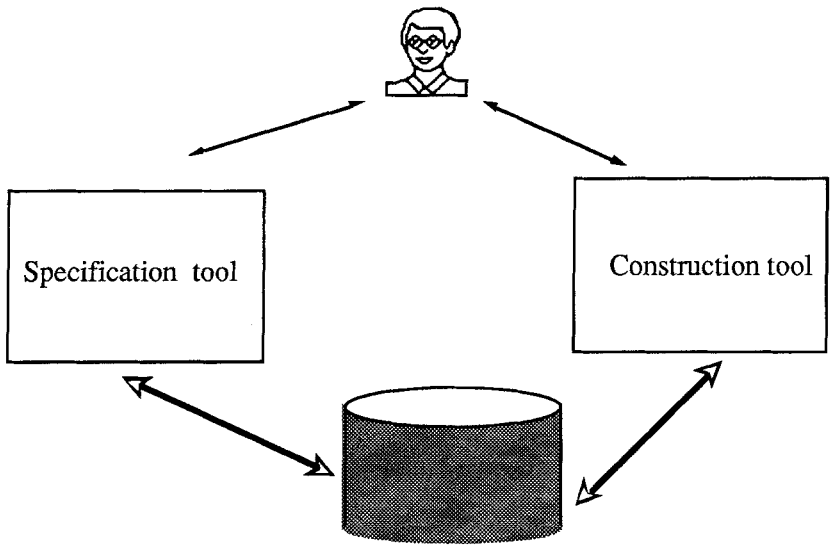

Figure 2 - Integration between specification tools and construction tools using a common data base

This kind of integration is often referred to as *the* future integration between different CASE tools, and between CASE tools and implementation tools. The emergence of different standard repositories (from different vendors and standardization bodies) and interchange formats (like EDIF - Electronic Design Interchange Format) give high expectations to a storage standard.

One advantage of this way of integrating specification tools and construction tools is that the tools are independent of each other as long as they are able to read and write in the agreed formats in the data base. This makes the total system modular and thereby flexible. An other advantage is that the integration could be *instant,* i.e. it is possible to implement the integration so that there is no need to manually trigger the integration.

One disadvantage is that a common data base requires much agreement between the vendors of the different tools and thereby also re-implementation of existing tools. To access a common data base, one has to use a general data base management system having a special subroutine interface. To adapt existing tools to such an interface (not all existing tools uses general data base management systems today) requires a lot of re-implementation. The vendors also have to agree on the data formats that shall be read and written in the data base. There will also be a problem when some tool support more or less information than what is possible to store in the standard repository (semantic integration). One important question is: should the common data base store the union or intersection of all information in all tools that should be integrated?

The key issue to the success of this type of integration is that one standard is commonly *accepted* by the different vendors (and users), and that the vendors conform to it. There are

a number of commercial benefits for the vendors by doing so, but there are also a number of commercial drawbacks.

This solution may also cause integrity problems, especially if one tool is permitted to write in the data store of an other tool. The last disadvantage is that there is still no user integration.

Model 3: Integration using communication mechanisms

In this model, the tools are considered as processes connected to and communicating through a data network. The integration can be performed issuing some network service, e.g. file transfer (FTP) or remote procedure calls (RPC). To achieve a higher level of integration than in the models above, RPC is required. In that case, the integration is implemented by one tool requesting or supporting information from / to an other tool (see figure 3). This is done by initiation actions in the other tool. The tools may use their own data stores or a common one.

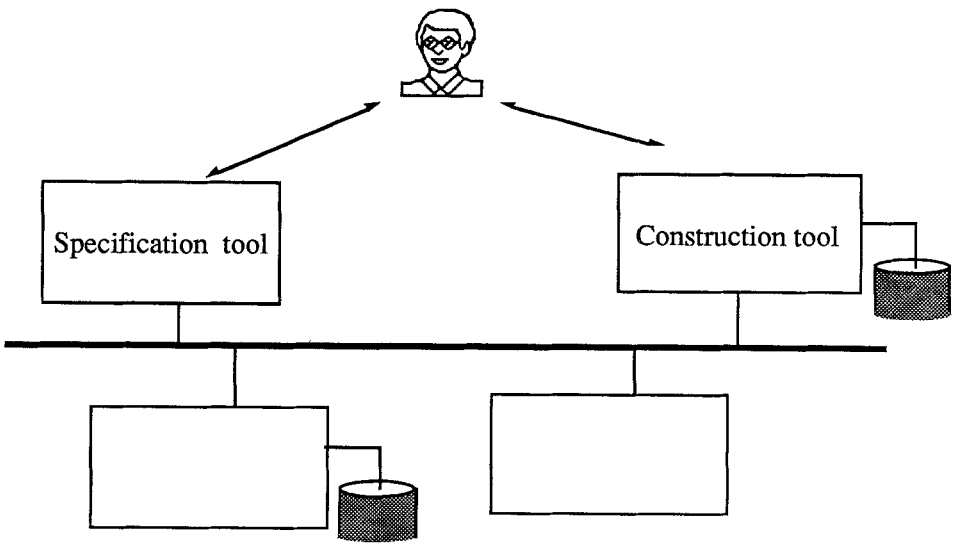

Figure 3 - Integration between specification tools and construction tools using communication mechanisms

This model requires a well defined interface and interaction protocols that the tools agree upon. The proposed EDIF-standard could be one such interface.

The advantage of this way of integrating specification tools and construction tools is that it enables integration in a very flexible manner. One tool may ask for some specific data, or it may start a function in an other tool. One may also use an "agent" to handle the communication. The integration may also be implemented so that it is "instant".

This model gives a form of "loose coupling" between the tools. Ideally, one tool can be exchanged with a new one as long as it offers functions that are equal to or a superset of the functions in the old one. An other advantage is that the responsibility for the data lay on

each tool. To access or alter data in an other tool, a function must be activated, an thus integrity constraints may easier be fulfilled. It is also possible to access *derived* data that is not stored by the tools, but that can be accessed using a function.

One disadvantage is that the vendors of the different tools still have to agree upon quite a number of things (but not necessarily on a common storage format), and this may cause re-implementations in existing tools. An other disadvantage is that there still is no user integration. The CASE tools and implementation tools are still separate user tools.

Model IV: Integration using software factory architecture

This model [OFTE87] is to some extent quite equal to the previous one, but to some extent also quite different. The tools are still processes communicating through a network, but the different tools are now *services* issuing special functions. The user interface is handled by a special *user interaction component,* a tool that support the different tasks in a systems development process (see figure 4).

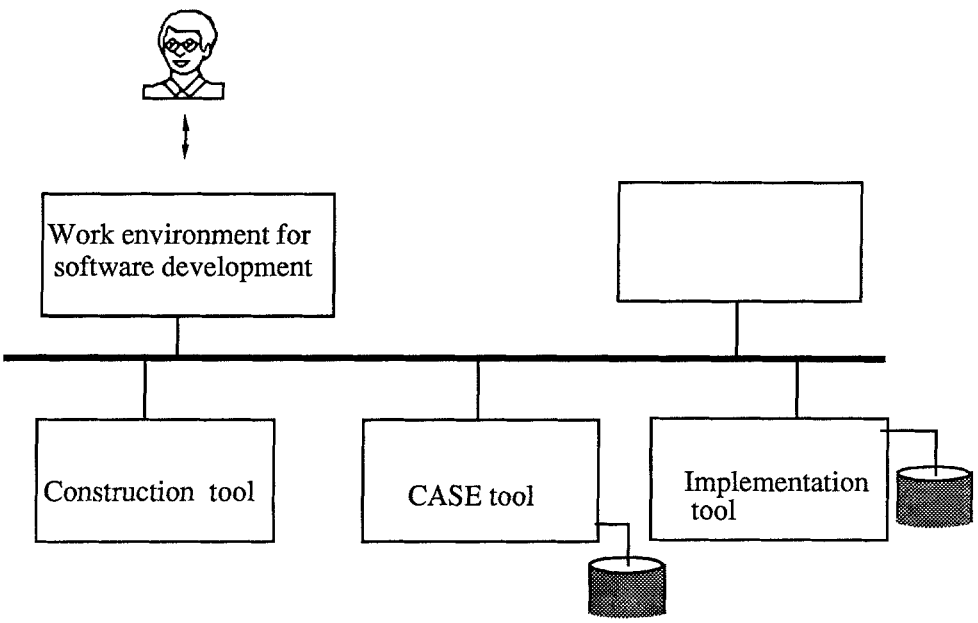

Figure 4 - Integration between specification tools and construction tools using a software factory architecture

The advantages in the previous model also apply on this model, but all the disadvantages do not. The vendors still have to agree on standards and protocols, but on an other level. The tools are now functional components issuing services to other tools, and much of the integration efforts are handled by the user interaction component. Therefore the different tools mainly have to be able to *respond* to certain functions, and only to a small extent themselves *issue* actions in other tools. The integration is no longer done on a tool to tool basis, but more on "global" level in the software factory.

The main advantage, though, is that there in this model is *user integration.* The distinction between the CASE tools and implementation tools (and other tools) are transparent. The user gets tools supporting his work, and the user interaction component handling his work environment issues the necessary functions in the service components. So that when the user perform some changes in the implementation phase, the changes in the CASE tool is performed automatically, i.e. the user has to perform all the changes required by the system, but he does not (and need not) know which changes apply to the implementation tool and which apply to the CASE tool.

The disadvantage of this model is that building the user interaction component is difficult. This kind of architecture requires that a lot of vendors agree on it, and offer their tools as functional components, with a well defined functional interface.

Other aspects of integration

There are a number of other aspects concerning integration between different CASE tools, and between CASE tools and other tools. We mention some of them briefly.

CASE tools should be integrated with tools covering tasks in the whole software engineering process, most important are project management tools and quality assurance. CASE tools may support those tools with valuable information in a more or less automatic manner, e.g. information on the degree of completion of different parts of the system that is developed.

An other aspect that covers the whole software engineering phase is the methodology, or life-cycle model that is used. The software engineering work may be much more structured if the tools support the life-cycle model. Such support is given in some existing CASE tools, i.e. tools from one vendor that support most software engineering activities. This kind of methodology support is sometimes referred to as *process control.* Related aspects are *requirement traceability* and *change control.*

A recent development trend is an integration between tools for software development and tools for hardware development.

An other resent trend is integration of artificial intelligence techniques and tools into the CASE tools. Such techniques and tools can be used both to check rules in the supported techniques, and to help the user in the *creative* part of the development work.

CASE tools (and other software engineering tools) should also cover a quite different aspect of integration, namely integration between different people. This covers both multi-user tools, and support for cooperation between different people cooperating in developing a system. This is sometimes referred to as *groupware.*

Integrated Software Development Environments

In the above section describing CASE tools, we pointed out some characteristics of an ideal CASE tool, which we denoted an Integrated Software Development Environment (ISDE). In this section we investigate ISDEs in more detail. We give a definition showing the main characteristics of ISDEs, we look at who makes such environments, and we point out what

is achieved by using such environments. We use the term ISDE, because we feel that it covers the functionality of such an environment. The term Integrated Project Support Environment (IPSE) and Software Factory are sometimes used to denote the same type of environments.

Characteristics of Integrated Software Development Environments

An integrated software development environment is a collection of tools supporting the different tasks that must be performed while developing software. The tools must be integrated, also on the user level, i.e. there must be user environments for different user groups, consisting of user interaction components tailored to support a special task. Each user interaction component may use different service components (or tools) when operating.

An ISDE must give support for a wide variety of tasks, not only typical systems development tasks like requirement specification, analysis, design, implementation, testing and maintenance, but it must also support related tasks like project management, methodology support (process control), etc.

Furthermore, an ISDE must support all users or user roles in a software development team. Not only data processing roles like programmer, systems analyst, etc., but also more administrative roles like project manager, sales and support people, etc.

Because software development usually is performed in teams, an ISDE must support cooperation and communication among team members. This requires more powerful tools than traditional electronic mail.

Such an environment must be easy to change and to supplement when new and powerful tools emerge. This implies an architecture that supports reuse of old and new components, that the architecture is flexible and configurable, and that the environment operates in a distributed, heterogeneous environment. It must be possible to exchange an existing component with a new one, without having to do major changes in the other components in the environment. These characteristics point towards usage of object orientation [MEYE88].

Who makes Integrated Software Development Environments?

There are quite a number of research and developments efforts around, that aim at building ISDEs.

In the large european research and development programs, like ESPRIT, EUREKA and ALVEY, there are different projects that develop parts of or complete software development environments. One example of such a project is the Eureka Software Factory project. It is one of the largest of the european projects in this area. It will be described in more detail below.

There are also quite a lot of research and development efforts in this area in the US. These activities are performed by various universities, research institutes, governmental bodies (e.g. Department of Defense), and software vendors. Arcadia and CAIS are two examples of such project.

In addition, there are some development efforts conducted by large multi-national companies (like Alcatel) that develop integrated software development environments, and some CASE vendors also develop tools that have some ISDE functionality (e.g. Software BackPlane from Atherton Technology).

What is achieved by using Integrated Software Development Environments?

There are lots of benefits that may be achieved by using ISDEs when developing software. In this section we point of some of them.

The main benefit is *reduced costs and development time* in software development projects. Because the environment is integrated, both on the user level and on the tools level, there will be increased productivity in projects using ISDEs. Such an environment will also support reuse of code when developing new software. Furthermore, the integration issues higher quality in the software developed. All of this elements contribute to reduced costs and development time.

One will also achieve better cooperation between members of the development team, which also gives positive impact on quality, productivity and well-being.

The project manager will be offered better tools for estimating the costs of the project, and for monitoring the progress.

Eureka Software Factory (ESF), an Example of an Integrated Software Development Environment

Center for Industrial Research (SI) is participating in a project that will produce different elements of integrated software development environments - the Eureka Software Factory project (ESF). This project will produce an architecture and a framework for ISDEs, basic building blocks, general components, and a number of environments for different application areas (like business applications, real time applications, telecommunication systems, and embedded systems).

The participants in the project are large european companies, computer manufacturers, software houses, research institutes and universities. Figure 5 (next page) shows the different participants i 1989. (The consortium will change during the project).

The project has a 10 years horizon (1987-1996), and imply approximately 2400 man-years of work. The work is financed 50% by the industry and 50% by national government.

Goals of ESF

The main goals of ESF are *to produce an effective production environment for software* and *to facilitate the production of flexible and integrated applications for the end user.* These goals shall be achieved through a *top-down strategy* and focus on different users needs for *functionality* and *integration.*

ESF is meant to be a european answer to american and asian efforts on software engineering. It shall contribute to making european industry in general, and especially the

software industry, more competitive. ESF shall be driven by the industry and oriented against products in a marketplace. The project shall use results from other research programs like Race, Esprit, Alvey and ESA.

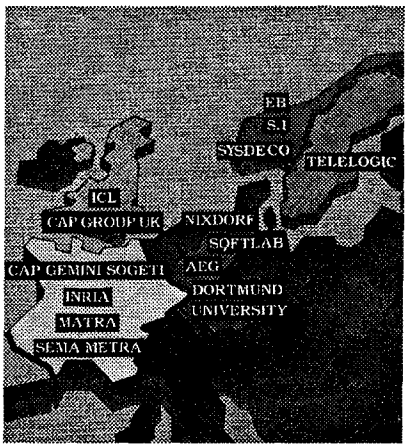

Figure 5 - The participants in ESF

Status in ESF

The project started in 1987 with a requirement specification phase. In 1988, a technical and administrative team was founded in Berlin, and a number of subprojects solving special problems and developing various components, was started. Most of the development work will be performed in subprojects. Also in 1988, a detailed architecture for ESF was developed. The first two milestones in the project were/are:

(i) In may 1989 a number of ESF demonstrators were developed to show different aspects of a software factory. The demonstrators were prototypes showing what can be achieved in the project. One of the ESF demonstrators is the *SI Team Environment (SITE),* which is presented in more detail below.

(ii) In september 1990 the ESF mini is scheduled to be finished. The ESF mini is a first prototype of an integrated software development environment (or software factory), and will consist of components that are commercial products.

Later on (1991 - 1996) the first commercial software factories will be released.

The technological foundation of ESF

On the technological level, ESF shall produce guidelines, standards and products in various areas.

ESF has already defined an architecture for building software factories, which acts as a basic framework. The basic mechanisms and tools in this framework will also be produced. ESF will also develop methods for software design according to this framework.

ESF will develop components and production environments that will be parts of a software factory, and give standards and guidelines for how to produce software components that will conform to the ESF architecture [ESF89].

The goal of the ESF architecture is to be able to "compose" user environments tailored to the tasks that shall be performed by the user, to use common services in different user environments, to be able to change the user interface of the factory without having to change the service components, and to have different components on different computers (e.g. special purpose computers like data base machines).

To facilitate such an architecture, the components must fulfill certain requirements. It must be possible to access operations and data. There must be a well defined interface to the components (preferable a subroutine interface). There must be a possibility to track errors, it is not acceptable that the components present error-messages directly to the screen. There must be a possibility for the components to report changes, so that other components can act according to the changes. Last, but not least, there must be a clear distinction between the user interface and the functionality.

Figure 6 gives an overview of the principles of the ESF architecture.

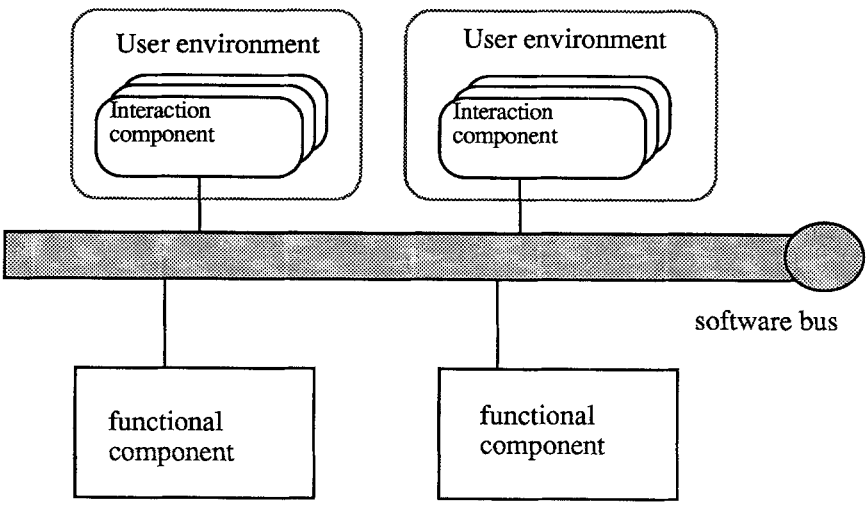

Figure 6 - The ESF architecture

Ideally, each component in this architecture is an implementation of an *abstract data type (ADT)* [GUTT77]. A data type is a set of values - its domain - and a set of operations by which its values can be manipulated. An *abstract data type* is a data type whose domain contain abstract values, i.e. values having an unspecified representation, and whose behavior is completely determined by the effects of the operations associated with the abstract data type. The semantics of each operation may be defined either informally, or by formal mathematical expressions.

The different components communicate through a *software bus,* which handles communication, data conversion, configuration, and a number of other services.

SITE - one of the ESF demonstrators

As our contribution to the ESF demonstrators in may 1989, Center for Industrial Research (SI) developed a prototype, *SITE - SI Team Environment,* which demonstrates a system that supports cooperation between a project manager and a programmer [SITE89]. The prototype covers functions like project planning, job definition, job assignment through electronic mail (with tools and documents integrated in the mail), personal work management, programming and documentation, progress reporting, and progress monitoring.

The prototype is implemented according to the ESF architecture, with different components on different workstations. The implementation is indeed heterogeneous, with components implemented in C, Lisp and Smalltalk. A number of existing components were reused in the implementation. The software bus were implemented using SUN RPC, extended with a specially developed update mechanism (in C).

The environment consist of five user interaction components (process support, job management, extended electronic mail, documentation and programming). These component use functionality in four service components (process model, unix mail, work context storage and document storage).

Expected results from ESF

ESF will produce results on many levels, some being general guidelines and standards, some being public domain software, but the largest quantities of results will be commercially available software.

(i) ESF will produce standards and communication mechanisms which facilitates composing environments based on components from different vendors.

(ii) ESF will produce basic tools (e. g. for user interface specification and data storage) which may be integrated with more specialized tools that fulfill ESF standards.

(iii) ESF will produce a number of separate components (like a project management module tailored for software development projects) that are easily integrated with other components into complete environments.

(iv) ESF will produce complete production environments for software development in special application areas. These environments will be adaptable to needs in the organization using them.

(v) Lastly - but maybe most important - ESF shall become a *market place,* where software developers may offer their tools and compose new tools, and where user organizations may buy just those tools or just the environments they need for their software development activities.

Conclusion

In this paper we have shown that CASE tools have a number of strong sides, but that they must be tighter integrated with each other and with implementation tools to be able to utilize their potential. We have presented different models for such an integration, and have shown that this integration is just the start of what is really needed to give support to the software development work, namely Integrated Software Development Environments (or Integrated Project Support Environments or Software Factories). I.e. environments that support all tasks in the software development - also the early phases where upper CASE tools have their benefits, and related activities like project management - and which integrate different tools, both on the user level and on the data level. The user interface of such environments must reflect the tasks the users perform, not the different tools that are used as service components.

We have pointed out that much work is done to produce such environments, and we have presented in more detail one of these projects, Eureka Software Factory. Hopefully, this project - and the other ones - will produce environments that will contribute to making software development an easier task in the future.

References

[BUBE88] Selecting a strategy for computer-aided software engineering (CASE), SYSLAB, University of Stockholm, June 1988

[BYTE89] CASE, In Depth Section (various authors), BYTE, April 1989

[CASE88] CASE Outlook 1988. CASE tools for Reverse Engineering. CASE Outlook 2, 2, p. 1.

[CASE89] Proceedings of The first Nordic Conference on Advanced Systems Engineering, Bjørn Nilsson (ed.), SISU, Kista, May 1989

[COWO87] Tools of the trade: Is CASE really a cure-all?, Jim Huling, Computerworld, April 20, 1987

[COWO88a] CASE product, Spotlight Section (various authors), Computerworld, June 6, 1988

[COWO88b] What CASE can't do yet, Tony Percy, Computerworld, June 20, 1988

[DAMA88] A Guide To Selecting CASE Tools, Michael L. Gibson, Datamation, July 1, 1988

[DAMA89] Cutting Through the CASE Hype, Kit Grindley, Datamation, April 1, 1989

[ELEC87] Integration is crucial to CASE's future, Tom Manuel, Electronics, September 17, 1987

[ESF89] ESF Technical Reference Guide (version 1.1), Eureka Software Factory, 1989

[GIBS89] The CASE Philosophy, Michael Lucas Gibson, BYTE, April 1989

[GUTT77] Abstract Data Types and the Development of Data Structures, John Guttag, Communications of the ACM, Vol. 20, N. 6, June 1977

[HEDQ88] Datorstödd programutveckling - CASE - i USA, Torbjörn Hedqvist and Jonas Persson, Swedish Attaché for Science and Technology, 1988 (In Swedish)

[IEEE88] Automating software: proceed with caution, John Voelcker, IEEE Spectrum, July 1988

[INWO88] Making a Case for CASE Tools in the Application Development Process, InfoWorld, January 11, 1988

[JONE89] Why Choose CASE?, T. Capers Jones, BYTE, December 1989.

[MCCL89] The CASE Experience, Carma McClure, BYTE, April 1989.

[MEYE88] Object-Oriented Software Construction, Bertrand Meyer, Prentice Hall 1988, ISBN 0-13-629049

[NILS85] The Translation of a Cobol Data Structure to an Entity-Relationship Type Conceptual Schema, Erik G. Nilsson, IEEE Proceedings of the 4th International Conference on Entity-Relationship Approach, Chicago, 1985

[NILS88] CASE Tools are still too young to reach decadence, but we better watch out, Erik G. Nilsson, Position Statement at the 7th International Conference on Entity-Relationship Approach, Rome, 1988

[NILS90] Aspects of Systems Integration, Erik G. Nilsson, Else Nordhagen and Gro Oftedal, Proceedings of the First International Conference, Morristown, April 1990 (In press)

[OFTE87] The Use of Remote Applications from a Smalltalk Workstation, Gro Oftedal, Master Thesis, University of Oslo, 1987

[ROEV89a] Analysis Techniques for CASE: a Detailed Evaluation, Rosemary Rock-Evans and Brigitte Engelien, Ovum Ltd, 1989

[ROEV89b] CASE Analyst Workbenches: a Detailed Product Evaluation, Rosemary Rock-Evans, Ovum Ltd, 1989

[SHIE88] Second Chance for Escape?, Gordon Shields, Computerworld,June 6, 1988

[SITE89] SITE- SI Team Environment, How to Support Cooperation in Teams, Center for Industrial Research, May 1989