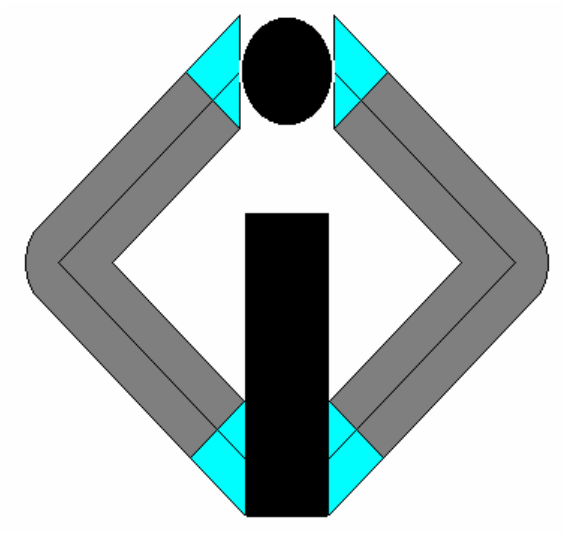

By Andrei Kapustin,
Cybernetic Intelligence GmbH,
Luzern, Switzerland

CASE system

Overview

P-CSO-1999-02-1-0

Feb 4, 1999

Contents

The problem

The number of CASE tools, supposed to help people in software engineering, is vast. However, all of these I have seen or heard of so far have several common characteristics:

- They all support only a small part of the total software lifetime.
- They all have closed or nearly-closed architecture.
- They always try to impose their view on what a "correct" methodology is on the user.
- They all delegate version control and distributed development features to users. At best, users may be able to use third-party products for both.
- Their access control never goes beyond login authorization.
- Most of the tools fall into two categories: either too simple to be of any use, or too complex to use at all.

There are, to be certain, more drawbacks common to all or most CASE environments, (e.g. each CASE tool is either heavily oriented to a single programming language, or have no language support at all, etc.).

Some of common characteristics of today's CASE environments are analyzed in more details in following sections.

Methodology

There is no doubt that having a good software methodology may, when used correctly, result in better software. There exists, however, a trouble of the same sort as for programming languages, if only to a lesser degree: since more than one choice is available, one that best suits the project at hand must be chosen. This would be simple indeed, if only people did not have a tendency towards ideological wars at their worst.

The "which programming language is better" wars are with us for over 30 years, and, it seems, will never cease. The "which methodology is better" wars are not so old, mainly because software engineering methodologies were developed much later, but, to a degree, they are much stronger. People tend to back up their opinions with personal likes and dislikes at best, and nothing at all at worst.

Moreover, the methodological wars often concentrate on things which, in the big picture, mean nothing whatsoever (like "I like methodology A most, because I like its graphical notation, because I like classes to be drawn as ellipses").

On top of that, there is, for some reason unknown, a deep distinction made between "object" and "non-object" approaches, as if they were so different as to be exclusive. In most cases, the people who shout "let's rid of all the non-object-oriented staff" loudest, don't have an idea about what an "object-oriented" really means, anyway. Personally, I don't see why "object" and "non-object" techniques can't coexist in the same project - there's nothing wrong with DFD where live objects, not dead data structures, are passed between processes.

Still, as CASE tools for software analysis/design are written by people, each one heavily promotes one methodology with all it has - notation, steps to take, etc. What is, somehow, forgotten is that no existing methodology is even close to being perfect - otherwise there will be no software disasters. Still, all present CASE tools have methodology built-in to the degree of a straightjacket.

Architecture

This is the second worst feature of modern CASE tools. Each CASE tool has its own view on what a software project consists of, such views being incompatible for different CASE tools from day 1. It would not be so bad, it these views were not completely closed - e.g. a CASE tool may support a view that "a software project is a set of interconnected class, module and deployment diagrams" - indeed, most CASE tools don't go beyond that.

This, of course, means that in a real project such CASE tool will be hard to use at the best. Real projects never fall into nice categories implemented by a CASE tool - a real project is a set of diagrams **and** dictionary **and** planning information **and** ten different versions of requirements **and** much more. This is where existing CASE tools give - none supports the needs of such project even remotely.

While it is clearly not possible to create, once and forever, a CASE tool which will support everything a software engineer may possibly need, we can view a problem from a different angle: for most SE activities there are some standalone applications, which, more or less, can do the work. There is MS Project for planning, MS Word for text documents, etc. So, while CASE tool can't be a Cartesian product of all their functionality, it is clearly possible to equip it with some basic features **plus** an ability to integrate third-party products in.

Long ago, the same approach was used for files. Older operating systems tended to impose their views on what a file should be (like "a file is a sequence of fixed-length records", etc.), but modern operating systems use the open approach instead - the file system provides a generic naming & access mechanism, and it's entirely up to individual users what the contents of these files should be.

Version Control and Distributed Development

Although these two functions (which, together, will be referred to as VCDD) actually belong to OS level, no OS have them built-in. Instead, the standalone software is built, which, to some degree, implements one or both. Most widely knows examples are UNIX SCCS, OpenVMS CMS and MS SourceSafe.

There is, of course, a whole set of problems with this approach:

First, all such products are incompatible. Not only because of different user interface or different OS they run under - much more important is that they assign different semantics to the same operations. One example is the "put back" operation, which creates a new version in CMS and replaces the last version in SCCS.

Second, VCDD, logically, doesn't really belong where it is - it is much lower-level than what OS gives to users as files.

Third, all current VCDD products behave inconsistently **unless** user spends a considerable effort to ensure consistency. In fact, current VCDD systems require a rather large effort just to get used to - a half-man-year project may well require another man-month **just** for putting it into VCDD system.

The pre-implementation stages of the software lifecycle are where things change the most, and also the largest number of people is involved. None of existing CASE tools supports VCDD to any degree, and some actually make using third-party VCDD software more difficult. In something like SELECT, which is supposed to support DD, one can't just use an external version control system to say "fetch the version 2 of the model" - it has to be "fetch the version **and then** insert it into a CASE repository **and then** make sure it's my local version…" etc.

Access control

Most CASE tools have the simplest access control mechanism there is - none at all. Anyone can log in and thrash the model to pieces just by accident - or even on purpose.

Some CASE tools (far less numerous) do just a bit better - with usernames and login authorization. Still, once you logged in - do whatever is on your mind.

There are, however, situations when a fine-grane access control would be of much benefit. For example, we can imagine a large project supported by a CASE tools, where analysts and designers can modify the model, administrator can modify user lists but **not** the model, and customers can read the model and write their comments but **not** modify the model. There are, of course, many situations when a fine-grane access control would come in very useful, or even absolutely essential, like for the development of banking/military software. The general ideas in the access control area are even worked through in many years - UNIX, although very old, supports the read/write access control for

both persons and groups, and Windows NT, with its access control lists, offers full military-class security.

Simplicity vs. Usability

This is a tricky field - for each user holds his own opinion as to what is “simpler to use”.

Still, the majority of CASE tools is so simple, that in real projects they are virtually unusable (like VisualCASE, with its extremely simple user interface - but all you can do is to draw a class charts). True, the smaller the CASE tool the simpler the user interface can be (although there are some exceptions even to that - most notoriously, the EiffelCase, which is both dumb and has awkward UI). There is, however, the other side - the simple a CASE tool, the less you can do with it.

On the other end of scale are monsters like SELECT. All its vast functionality is inside, but so well hidden from user, that, for example, to completely describe a single method for a single class half-an-hour may well be not enough. Software designers may have to live with that, of course, but no customer will ever want to use something like this to view a model.

In one of his C++ books, Bjarne Stroustrup stated, that “*not only a feature should be available, it should also be affordable to use*”. If we look at the “functionality/difficulty of use” ratio for existing CASE tools, the Rational family comes very close to the top - but still too complex for all save highly trained software designers.

The answer

So, if the “bad” CASE tools are what have been discussed so far, what should a “good” CASE tool look like?

The sensible approximation can be achieved by… simply negating the list of features presented at the beginning of chapter 1. A “good” CASE tool:

- Will support entire project life cycle.
- Have an open architecture.
- Be not bound to any software engineering methodology.
- Implement full version control and distributed development features in its kernel.
- Support fine-grane access control.
- Present a simple user-interface.

We may also add some items just out of common sense:

- Be not tightly bound to any particular programming language.

These points are explained in more details in the following sections.

There are, however, two things too general to be applied to just CASE tools:

- Zero-overhead (“you don’t pay for what you don’t use”). For example, for a small project there will be no need for VCDD - and a CASE tool which still forces user to login, check files in and out, etc. is a bad idea indeed.
- Minimum limitations. Many software engineers are very smart people, and if a CASE tool tells them what to do and, most important, what **not** to do - it’s a bad CASE tool. A CASE tool with limitations imposed “for their own good” (like “you have to think C++, you have to use UML, you have to use only single inheritance, you have to use object-oriented approach, etc.”) will never be a success - people will either find a clumsy way to get around the imposed restrictions, or will drop the CASE tool altogether.

Methodology independence

What **is** a software engineering methodology? Two things, really:

1. The sequence of steps which should be taken to create a software.

2. The representation for results of each step.

The first thing is really easy - let's just give a CASE tool user an ability to perform steps in whatever order he likes. For one project, steps may be "compile data dictionary" followed by "create class list", while for another project steps will be "create DFD" followed by "create ERD". This, in fact, is permitted by most existing CASE tools - there are several types of diagrams (like class diagram, module diagram, collaboration chart, etc.) which can be created in any order.

The representation problem is much worse. Not only because there are many incompatible representations for, say, class diagrams - but because, often, the representation is somehow equated with the methodology. Which, of course, is completely wrong - if we know that class A inherits from class B, what difference does it make whether we draw both as rectangles, ellipses or cloud shapes?

The class model itself represents the semantics of the project and, therefore, is essential. The drawing of a class model is a mere syntactical convention - and, therefore, is not important for the project - only for users, who are to read it. That the same semantic model can have several representations is a very old idea (and, indeed, Rational family of CASE tools does that to some degree, allowing to choose one notation out of three at any point), but the suitable implementation of this idea with regard to CASE tools is yet to arrive.

Ideally, we can imagine some set of "notational conventions" for each notation - one for, say UML, one for classical SSADM, etc. We can even extend this set to "methodology conventions" - the set for SSADM will contain all notations **plus** the fact that DFD should be created before ERD, etc. Having something like that may help designers by providing constraints on the sequence of steps (e.g. CASE tool may say "You're using SSADM, so you first step **must** be DFD") or useful hints (e.g. CASE tool may hint "You're using Coad&Yordon, so, won't you like to start with parsing requirements text for classes?").

Although the latter may well be a bit too much, the complete independence of the semantic model from notations used to draw it is essential for a "good" CASE tool. Once we have that, we can have all kinds of interesting situations: for example, one designer likes OMT, while another is a fan of Booch; each draws class diagrams using his preferred notation; but both develop the same model. The third designer, then, looks at the class model expressed in UML notation, totally unaware that anything else was ever used.

Open architecture

The whole idea of a CASE tool is to have all information related to a project in one place; hopefully, all such information interconnected. This "one place" is traditionally called a "repository".

All existing case tools have repositories with strictly predefined structure, which means you can keep there only things which CASE tool permits - class charts, DFDs, etc. However, making a repository more "file system – like" will surely be a great thing, because it will allow user to keep in one place everything about the project - class charts, requirements documents, project planning information, source code, bug lists, etc.

Integration of the third-party applications becomes important. The CASE tool may be able to keep requirements text in the repository - but it doesn't have to contain a text editor, it will just call something like MS Word to view or edit requirements. There are more benefits, of course: if we implement VCDD at the repository-level (next section), then we will be able to maintain several versions of requirements, several versions of the project plan, etc. - while the applications which are used to actually view/edit their contents don't have any VCDD support of their own.

So, instead of CASE tool being an application assisting in software design, it becomes a framework for software project activities and documents - this is why this document's title is "CASE system", not "CASE tool".

Version Control and Distributed Development

It is clearly unwise for a “good” CASE tool to delegate VCDD features to an external application - VCDD just doesn’t belong to the same level. A CASE tool is a system for creating and maintaining project-related information - and VCDD is needed **in the process** of such creation/maintenance.

The “good” CASE tool, therefore, will have a full set of VCDD features at the lowest level possible - repository level. Everything placed into repository - class models, requirements texts, bug lists, etc. - immediately starts to use VCDD features. Something as impossible with present project development technologies as “get me the 3rd version of requirements” becomes blindingly easy with VCDD-enabled repository, even though MS Word, which is used to edit and view such requirements, has, of itself, no VCDD at all. Many people will be able to work on the same project at the same time, all using the same mechanism to coordinate all efforts of their work (had anybody ever tried to join two class diagrams, knowing they were once the same?).

Access control

The login authorization as the only means of access control is, clearly, unacceptable - because often it is insufficient. A “good” case tool will allow to specify access rights for every separate thing in repository and every user. Something like Windows NT ACLs provides a very good approximation - you have a default, which is suitable for most situations. However, when it is not - you can specify detailed access rights for every user and every object. This becomes essential in DD-enabled CASE tool, as the number of people working with the project over time will be vast. Again, the same principle applies - “no restrictions imposed”. If you don’t want access control - don’t use it. If you want it - it can be tightened up to a military level.

Full support of a project lifecycle

This is a direct consequence of an open architecture and integrated VCDD. During a full life cycle, there will be many releases, requirements will change, code will be ported from one programming language to another, bug lists will be assembled and software modified accordingly, etc. It’s much better to have it all in one place, with centralized access control more important than ever (we don’t want a big boss to see a bug list, do we?).

Programming language independence

This is something existing CASE tools don’t have - and a “good” CASE tool should. Once we have, for example, a declaration of a function with all parameters plus flow-chart for a function body, it’s really not essential to which language we translate it to generate executable. Or, we can go even further - what’s wrong with directly executing programs in flow-chart notation? This, by the way, is around for many years, known as “graphical programming”.

Conclusion

This document, of course, outlines only the most troublesome problems with existing CASE tools, and discusses only the most general guidelines towards a better CASE system. There is no doubts that much more investigation and thinking will be required to come up with something remotely like requirements specification.

However, the current situation in the CASE clearly demands such actions to be taken. Existing CASE tools are based on technologies and ideas almost than two decades old, and, as proven by many projects, in many cases just don’t work. The new technology is clearly there - spread around in bits and pieces, so it’s time to collect all these bits and pieces together and create a more-or-less solid CASE system which will support it.