developer.*

Object Oriented CASE Tools: Lost Opportunities and Future Directions

By Mario Van Damme

Those of you that have been around long enough might remember the glory period of the traditional CASE tools and structured programming. These were the days where a lot of people were experimenting with CASE tools and even went so far that they started to predict work estimations using function point analysis. The main drive was that companies wanted to get their IT projects under control again. In the late 80s, the traditional CASE tools were victorious in a lot of large companies because of this reason. These tools had a decennium to learn from their mistakes and from the mistakes of others.

In the early nineties, with the rise of Object Orientation, the traditional CASE tools were perceived as a failure by many people. The root cause of this bad perception was that the tools a) failed to embrace OO, and b) were used in combination with traditional software development processes. This caused traditional CASE tools to be strongly associated with the waterfall methodology, at a time when iterative development and rapid application development were increasingly gaining momentum. With the rise of the OO CASE tools (Popkin Architect, Rational Rose, Cool:Jexx, etc.) in the mid-nineties, everything was reinvented.

As with most software products, the CASE tool market is highly feature-driven. However, CASE tools have been around for more than 20 years, and one might expect also some mature features. The irony of it all is—as I will illustrate further in this article—that a lot of features with high user value can be implemented; it's not that there are rational reasons to not do it.

This article outlines some important aspects that today's "best of breed" OO CASE tools could learn from the traditional CASE tools. A generally applicable rule—which we forget frequently in the software industry—is that we should not neglect the experiences in a very similar domain just because we have labeled it "old-fashioned."

An Object Oriented CASE History

For the purposes of this discussion, I will identify three “generations” of OO CASE tools:

First Generation OO CASE Tools: Diagramming Tools

Diagramming tools only allow the modeling of ideas—things like concepts, analysis artifacts, etc. There’s no link to code. Everything is maintained manually.

The first OO CASE tools still had the “diagramming dialect” battle to fight, which made comparing the tools quite difficult because people were more comparing the diagramming dialects than the tool’s features themselves. (You may remember the “OO notation wars” before Rumbaugh, Booch, and Jacobson got together on UML and ended the controversy.) After the introduction of UML in the late nineties this was resolved.

Second Generation OO CASE Tools: Code Visualization Tools

A “code visualization” tool is a form of diagramming tools that knows how to “harvest” code and map it onto a model of classes and diagrams (this process is also known as “reverse engineering”).

These tools also allow you to start from a model and to transform that into code. However, in these tools this implies that you are modeling at the lowest level—the level where you have a one-to-one mapping with code.

Typically this is not that productive, as you need quite some training and patience before you are able to get the result that you want in one go. An agile way to approach this is using the tool that suits the purpose: If you want to get an interface down on paper and you know the programming language, then just type it using a code editor. If you want a visual representation of it to be able to highlight the relationship with other classes (e.g. which classes implement this interface), then use the reverse engineering feature.

To recall an anecdote, many years ago in the Microsoft COM era I wanted to model an interface that could be generated in IDL (Interface Definition Language). It took me quite some iterations before I could get it right. Effectively, this was a waste of time because I was IDL literate: I really knew what I wanted it to look like and how to write it.

THIRD GENERATION OO CASE TOOLS: COMPUTER AIDED SOFTWARE MODELING TOOLS

CASM tools assist the user in his/her choices regarding software architecture, data modeling, business modeling, etc. These tools don't solve all the problems, but rather help in coming to a solution. For example, the tool could provide a computer-aided review of your design; or it could include a data normalization/demoralization tool; or it could suggest or highlight potential "deviations," such as the use of a functional database key (e.g. customer number) instead of a technical one. These are the tools of the future. In the rest of the article, I'll outline what features you should expect of such a tool.

Lost Opportunities and Future Directions

In the remainder of the article I will explore a variety of features shared or not shared by traditional/structured CASE tools and today's "third generation" OO CASE tools, including discussion of opportunities for today's OO CASE tools to incorporate some of these features.

Before beginning this discussion, I'd like to make the distinction between a CASE tool as "tool" and as a "framework." The viewpoint of a "tool" encompasses everything I can do without any custom development. The functionality is directly accessible using the user-interface of the CASE tool. The viewpoint of a "framework" encompasses everything that can be done with the tool using custom development (that is, by using the CASE tool's API).

As a brief illustration of the difference, consider a standard feature of a CASE tool user interface: finding all artifacts with a certain name. This is a common feature. In contrast, extracting the dependencies between all classes to a spreadsheet is typically *not* a feature of a CASE tool. To do this, custom development—using a scripting language or a regular software development language—is required using the tool's API. I'll discuss the "framework" viewpoint in a future article, because the concerns are different.

The table below summarizes the list of features we will explore, including an indication of whether and how the features are implemented by the traditional/structured CASE generation of and the more contemporary Object Oriented generation of CASE tools.

Feature	Structured/Traditional CASE Tools	OO CASE Tools
Central repository	X	Decentralized
Repository	Database	Structured file
Multi-User Support	Full support	Limited support
Instance Modeling *(modeling 'snapshots' of live data in an easy way)*	X *(You are guided by filling in tables)*	Limited support (no guidance at all)
Design Critics	X *(Standard feature)*	Only experimentally in Argo-UML
Data Normalization Tools	X	No support
Impact Analysis	X *(At many levels)*	None
Code Generation	X *(Full-blown code)*	X *(Skeleton-code only)*
State Modeling	X *(Simulations possible)*	Only supported in Rational Rose Real-Time
Reverse-Engineering: Visualizing the Architecture	X	Limited or no support at architecture-level

Features shared or not shared by traditional and OO CASE tools.

The Repository

The repository in which all metadata was stored in traditional CASE tools was a central database. Extracting metadata could be done easily by querying the central database using SQL language and its aggregation functions. SQL is universal, easy to understand and an obvious "standard." In this section I will discuss the ideas of storage format and central repository.

Typically and especially with the introduction of XML into the OO CASE tool repository format, storing goes very slow. An example of the 'overhead' of the storage format is the IBM Rational XDE XML format: an empty model already takes about 800 KB!

Other tools use XMI (the XML Model Interchange standard) as an internal storage format. From an architectural point of view, this is a strange decision. XMI is typically something you would expect at the boundaries of the tool, not on the inside. What if the XMI standard evolves into a new version with maybe a new structure (note that XMI 1.1 broke severely with XMI 1.0)? Are they going to change their storage format as well? I agree that semantically you need a good mapping from your internal model to an external model, but personally this external format would not be my first choice. Apart from the 'Select' CASE tool, none of the OO CASE tools I mentioned in the introduction use a database as repository.

And what do they miss out on: fast storage, fast querying capabilities, transactions, etc.

Typically, OO CASE tools have an object API to query models. I'm not saying they should throw these capabilities overboard, but they could merit from relational database technology in the following way: by implementing the object query capabilities internally as database queries and by exposing a „public" database structure (or part of it) to create reports on. The latter would be an alternative to expensive tools such as IBM Rational Soda and the like (this is the best case, in the worst case there's even no custom reporting at all unless you resort to writing code).

For relational databases, a lot of standard tools are available that offer data mining capabilities. If OO CASE tools would adopt relational databases and part of the database schema would be opened-up ('public interface'), standard data mining tools could be used (business objects and other analytical tools) to perform the most complex analytical processing in a high performing way. As a lot of companies are storing strategic data in their business models, this definitely would come in very handy. The problem we are facing now is that big investments are necessary in new technology (or at least new implementations) to get this kind of analytical functionality in the OO CASE tools, while this kind of functionality based on relational databases is already a commodity.

Additionally, the concept implemented in traditional CASE tools was that you had a central store (single repository). OO CASE tools completely abandoned this concept and drive users into the direction of creating multiple model repositories. However, the reason behind this is that many OO CASE tools have problems with big repositories. This means that they mainly deviated from the "central repository" approach because they weren't able to create big repositories built on structured files.

Advantages of a central repository are that you can reuse artifact definitions between projects and/or that you have the possibility to easily check whether certain terminology is already used (and whether the semantics are not overloaded). Think about it as the ability for you to create an entity and link it to an already-defined entity. Good browsing and reporting capabilities would bring the documentation associated to the referenced entity in an easy way. The central repository concept has the promise of bringing modeling to the next level.

Multi-User Support

Modern OO CASE tools typically use the file system as the repository. For multi-user support, these systems use a source control system. Many of them use the XML Metadata Interchange format (XMI) to store the UML models.

In order to modify an artifact, the appropriate file needs to be checked out. After check-in, the artifact is accessible to the other users.

The multi-user aspect is of course less easy to solve with databases, but object versioning can also be accomplished using databases (there have been enough articles published which cover this subject). However, the storage capacity and speed of a database would allow more easily to support a larger amount of users, models and versions.

Modeling Instance Diagrams Easily

A lot of OO CASE tools don't have any good support for modeling instance diagrams (a.k.a. object diagrams, as opposed to class diagrams). An instance diagram shows "live" objects, a view you typically need to help explain a class diagram to business people for example. UML doesn't have a separate diagram for this. An object diagram can be completely modeled using a UML class diagram. Tools typically don't have an instance diagram view, where you can put classes on and just fill in the values of the object together with its identity.

Traditional CASE tools offered the feature to fill in data for a certain structure that you defined within the tool. Typically, this was a table editor where you could fill in column values per row. For OO CASE tools, the visualization would have to be done in the form of an instance diagram. None of today's tools have features to enable the creation of instance diagrams easily. The author of an instance diagram needs to explicitly know how object diagrams need to be created. Consider this list of rules that the author of the diagram needs to enforce without any help of the OO tool:

- Objects don't have operations, just attributes with their values.
- By convention, doubles are represented with a decimal point; strings are put between quotes, etc. There's no tool that enforces these conventions.
- Objects have an identity before the colon, and a class name after the colon.
- The only type of relationship that is drawn between objects is an association relationship.
- When you create an instance of a class that you already have defined, you have to check yourself that the attributes you add on the object indeed exist on the class.

Given the following class diagram, an instance diagram would look like this:

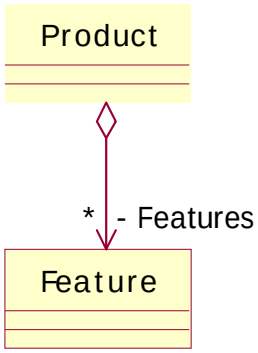

Figure 1: Class diagram

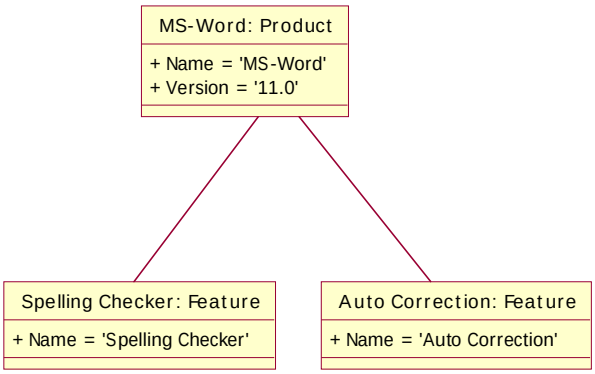

Figure 2: Instance diagram

In general, instance diagrams can facilitate communication with customers a lot, either to help in the explanation of a concept or just to double-check whether you have modeled the customer's business correctly.

Data-to-Entity Class Conversion

XML authoring tools have the capability to generate an XML schema from an XML document, but CASE tools typically don't have this capability. Integrating such a capability in an OO CASE tool would allow for the visualization (in UML) of the structure of the XML document without requiring extra tools. A UML representation of an XML schema can also be very helpful.

People have worked on a UML profile for XML schema modeling, but this has not really been standardized. During the XML hype, some OO CASE tools have added some XML features, but far too limited.

Data Normalization Tools

Traditional CASE tools had support to help you perform data analysis. Although an object is not limited to data but also includes behavior, this does not mean that you don't need any help in defining the correct structure of the data. Some of the traditional CASE tools guided you in going from data to structure using general rules to go up to the fifth normal form.

In an object oriented world this could be translated into something like starting from an object diagram, or from a non-structured table, and the system assists you to normalize the data into a structure.

Another feature is to take a database structure or object diagram and to ask the tool to validate whether it is normalized or not.

Design Critics

Some traditional CASE tools had normalization "critics" built in: These pieces of software informed the user of what was considered as bad design decisions (e.g. a table without a primary key, etc.). Unfortunately, some tools instead of "criticizing" were disallowing the offending option altogether.

As with all tools, the tool should never think it's smarter than the user. Maybe I just want to keep the model this way because I want to store my efforts and go home, or maybe I want to model in an agile way: analyze a bit, data-model a bit, etc. Another reason might be that you know that it is not perfect, but you are constrained to design errors that were made in the past.

Of course, in the traditional CASE tool era the new buzzword 'agile' did not exist yet. However, many OO CASE tools followed the same bad example: Many people have had the experience that when modeling sequence diagrams in Rational Rose, they had to fight against the "supreme modeling" knowledge (!?) of the tool. For example, when you tried to delete a signal which triggers other signals for example, the tool decides for you that you want a "cascading" delete.

Some OO CASE tools, like Argo-UML have experimented with so-called “design critics.” These are rules that are defined and checked on different artifacts (for example a design model, an analysis model, etc). A positive evolution that can be seen in the OO CASE tool scene is that a lot of tools are moving away from enforcing to advising. Typically, you can validate your modeling efforts as an action you need to trigger manually instead of letting the tool performing the validation automatically.

Impact Analysis

Traditional CASE tools allowed you to visualize the effects of certain changes. If we had this feature in an OO CASE tool, you could envision a scenario such as “If I were to change Component X, what would the consequences be? What impact does this have on my deployment architecture? How many components are affected?”

This together with reporting and data mining capabilities would make the modeling tool a key instrument.

Code Generation

In traditional CASE tools, code generation was not limited to skeleton code. Many of the CASE tools supplied a language to express business rules which were translated to the target programming language upon code generation. Some believe that in future, everything will be expressed visually in a UML 2 modeling tool. Others don’t believe in visual modeling at all. The truth will probably lie somewhere in between.

A picture tells more than a thousand words, but one should only model those things that have an added value when modeled visually. For some things, such as business rules, I have the feeling that a business rule language is more readable than a cluttered diagram or a scientific language such as OCL. Having said this, don’t get me wrong: expressing a business rule in a programming language such as Java, C#, or Ruby is equally good for me.

I have the feeling that in CASE tools today we have reached the productivity equivalent to what could be done with a 4 GL language. The next step forward will be either massive code generation or the introduction of a functional language.

State Modeling

Every UML tool supports state diagrams, which is a first-class citizen in the UML diagram family. However, only few tools allow simulating the states and transitions (e.g. Rational Rose Real-time). Also, almost none of the tools support the ability to validate a finite state machine: Is what I've modeled finite or infinite, is it representing exactly what I had in mind? Etc.

In real-life, finite state machines are used to model object state, technical workflows, business collaboration workflows, etc. I can assure you that a complex state diagram with sub-state machines etc. is very hard to validate manually. A tool should be there to help you in these efforts. It could check whether it is indeed finite, it could allow you to simulate the different states and transitions between states, etc.

Visualizing the Architecture

Starting from a set of binaries, a CASE tool could calculate the cumulative dependency of each component. (The cumulative dependency of a component is a figure that gives an indication of the complexity.) The CASE tool could also check for layer violations, depending on the strategy you want to see applied (strict layering versus "loose layering"), and check whether the layering has been violated.

Many tools only allow you to start from the code, while an architect (and also a designer and software developer) would like to have an insight in the big picture. Where does this component fit into the general picture? Is it allowed to access component Y and Z, or would this violate the layering or even introduce circular dependencies?

How do we currently work around these issues? Circular dependencies are detected using a daily build cycle. However, the compiler cannot detect when the layering strategy is violated.

In smaller projects it is possible to manually visualize the actual architecture, but for bigger projects this takes quite some time and is destined become out of sync.

The ability to visualize the component architecture would help tremendously in steering and following-up on the architecture being built (actual architecture) and the planned architecture. When you notice that the actual architecture starts deviating from the planned architecture, you can revise the architecture in an early stage, either by approaching the developers to get the components back in line with the envisioned architecture and/or by revising the envisioned architecture.

Conclusion

I hope that I made you realize—whether you believe in OO CASE tools or not—that there are quite some features that can be implemented *today*. Instead, all CASE tools are currently focusing on design pattern injection, Model-Driven Architecture, etc. The first OO CASE tool vendor that figures out how to break out of this cage and start addressing what is really required to get us to the next level at a decent price will be victorious.

###

About the Author

Mario Van Damme is a software architect, working for quite a number of years in the medical industry, and prior to that in the insurance and banking industry. His main interests lie in software architecture, analysis, UML, database design, software design & patterns, software development processes, business process reengineering, and software development. He can be contacted by e-mail at `mvandamme ~AT~ sopragroup ~DOT~ com` and `mario.vandamme.mv ~AT~ belgacom ~DOT~ net`.