CASE Tools

A Framework for Assessment and Evaluation

E. Hochmüller, R.T. Mittermeir, E. Kofler, H. Steinberger

University of Klagenfurt, AUSTRIA

Abstract

Using CASE tools is commonly regarded as necessity to enhance development productivity as well as software quality. However, the variety of commercially available tools demands a carefully planned selection process to identify the tool or toolset which meets best the particular expectations and requirements of an organization. This selection is a major project itself. This paper presents a framework for this process.

Based on a case study comparing four different tools in a sizable software project, we collected a set of criteria that ought to be helpful during case tool evaluation. These criteria define a framework of various dimensions. Organizations using it would have to weigh them according to their particular requirements.

1. Motivation

CASE tools are commonly regarded as efficient means to achieve productivity improvements in software development to raise the quality of the software product developed. Also for organizational reasons, modern software development is not conceivable without CASE [Flec93]. Thus, any software developer will sooner or later have to decide for a CASE tool which would fit best to his proper requirements. But as these tools are very complex and their choice has a long lasting effect on the future development situation this decision should be taken with care. The abundance of tools that are on first sight similar and the fact that the investment is costly and hard to reverse even aggravates the problem.

Sound evaluation includes the elaboration of a tool assessment strategy together with well-considered evaluation criteria, which - if carried out with reasonable care - will amount to a substantial project on its own right. Furthermore, a sound selection cannot be based on considering just some demo/toy-projects. Certain crucial aspects figure only after one has worked with the tool for a while. Alternatively, one can directly address some key features, if one has already prior knowledge from working with a similar tool. This asks for substantial experimentation in preparing the choice. A requirement most enterprises cannot afford. University environments are rather prepared for such experiments. Our labor resources and the support of tool vendors allowed for such a project. The experience we gained was not confined to become just acquainted with different tools. It enabled us to compare the tools on the basis of a specially developed framework for assessment and evaluation. This framework was first charted before the experiment, but it got sharpened and extended during the project.

In the sequel, we outline the purpose of our framework. Then we discuss it by explaining the rationale behind the individual criteria that structure its various dimensions. Afterwards, we will discuss the evolution of the framework and compare it to further approaches.

2. Purpose of the Framework

The primary purpose of the framework is to assist in CASE tool evaluations and in the subsequent tool selection. However, software developers differ in their static organization structures, in the development processes and strategies they apply, as well as in the methodologies, methods and technologies they use. Thus, one tool which fits best to the actual needs of one software developer can be completely inadequate for other companies. In order to be able to cope with these incompatibilities, we decided to establish a reference framework for tool assessment and evaluation. This reference framework remains as generic as necessary while becoming as detailed as possible. Hence, each software developing company should be able to instantiate its own demand-specific framework containing those criteria which are considered most important in its particular situation. Additionally, the framework allows the possibility to further specialize high level requirements as well as to include supplementary demands. In relating the reference framework to the proper needs (concerning process model, development methods, and further organization-specific demands) each CASE tool selector can set up his own priorities. Finally, the framework proposed can also be regarded as a general guided tour through various kinds of possible requirements providing starting points for criteria some of which otherwise would easily remain unnoticed.

3. Dimensions of the Evaluation Framework

As stated above, the evaluation framework should be regarded as a reference model which needs to be instantiated according to organization-specific requirements. The evaluation criteria are contextually assigned to ten semantically coherent dimensions each of which is hierarchically structured giving varied depths. The complete framework is given in the annex of this paper.

1. General Software Evaluation Criteria

This evaluation dimension taps aspects which are important with any software purchase. The range of these aspects comprises technical questions, organizational issues and general economic considerations.

On the technical side, the compatibility with the current hardware and software environment, and additional requirements for the underlying technology are to be assessed. Organizationally, one has to consider the current and future organization of development teams. Project management considerations such as multi-user support (How many users can simultaneously use the tool? Is it possible for several persons to work on one project simultaneously? ...) and multi-role support (Does the tool support different user roles, e.g. administrator, developer, manager? Which tasks are related to what roles? What about dependencies between roles? Which roles should be defined in the developers organization?) will extend already the set of general software evaluation criteria. Tool documentation and support by the distributor (Is the distributor also producer of the tool? Is there a hotline support? What about the support during the introductory phase? ...) are important issues concerning the as-is-utility of the tool and its efficient usage, while profane questions like international and national distribution are indicators of the longevity of the investment. Last but not least, purchasing conditions (concerning prices, maintenance, upgrades, and error removal) will determine the economical merit. Likewise the installation effort should be considered and budgeted to reach a fair selection.

2. Support for Initial Usage

The make or break of introducing CASE-technology is quite often determined in the introductory phases [Hoch96]. Hence, special emphasis should be laid on the criteria listed under this heading. Our experience is, that notably on-line tutorials provide important help, since they are rather used than written documentation. Certainly, some documentation comes with any tool, but its structure, readability and completeness matter. Likewise, one should see, that the expert user needs different support than the initial user. While for the latter the existence of an on-line tutorial and fully developed example applications are important, the former needs rather a precise, up to the point, well indexed reference, be it available in hard-copy or electronically.

Course material might also be an important criterion. With tools that adhere to a generally accepted standard, well published in the open literature, this might be less so than with those fully based on proprietary concepts. The contents of these courses have to address methodology issues as well as tool specific issues. This might also be the right spot to mention that tool selection should not be the first step in the evaluation project. One rather should decide first on the methodology suitable for the organization in the long run and focus on the tools supporting this methodology only afterwards. To prepare this choice, the generally available literature is of course again a key factor.

3. Extent of Support over Development Process

Today’s state-of-the-practice requires that a CASE tool covers most of the software development process. It is not sufficient that a CASE tool only supports some of the development stages, be it only early (Upper CASE) or just late (Lower CASE) phases. Usually, a CASE tool covering the whole software life cycle (albeit to a different degree) will consist of several tool components each of which supporting one development phase. Sharing a common user interface and data dictionary the tool components can be combined to form an integrated CASE tool.

This integration presupposes that development follows a certain process model. We support this view, since this adherence will provide benefits in its own right [Hump92]. For the purpose of this paper we assume a phase-driven process model and suggest to examine the kind and degree of support concerning the tasks of requirements analysis / early stages, data-oriented and function-oriented analysis and design, structured programming, testing, configuration management, and project management. If other process models are instantiated, the criteria have to be adjusted accordingly. However, irrespective of the process model followed, the prescribed process always is of a forward directed, progressive nature; the real process followed though, unfortunately, is less straight. Hence, even while the questions listed on the fourth and fifth evaluation dimension need to be reviewed and possibly adapted for the particular process one plans to instantiate, the dimensions themselves will remain.

4. Support of Forward Engineering

The straightforward software development following the downward flow of the waterfall model as it is covered in most software engineering textbooks is subject of our so-called forward engineering dimension. These criteria concentrate on the general orientation of the tool (e.g. towards data base design, general software development, or both), the kind of code aimed for (DDL, DML, 3GL, 4GL) and the methods supported by the tool. Out of the variety of existing methods, we selected the three most commonly supported classical methods (ER, SA, SD) and propose method-specific criteria for evaluating the compatibility of the tool with the proper standard notations. Again, if one is using other methods, the list of criteria might serve as input into a brainstorming process for the questions to be asked against such a tool.

Special explanation is warranted for the entry 'point of no return'. It stems from the experience that some tools require early decisions about the programming language that is finally to be used. The benefit is specific language support or use of language specific heuristics. This is certainly beneficial. If however - notably in a heterogeneous language situation - these decisions are forced too early or cannot be revoked, lot of rework has to be done. Similar considerations apply, when models ask very early to define the data types (types of objects) to be manipulated. Quite often, these need to be extended during dynamic modelling. The tool should allow for this.

5. Maintenance, Adaptation, Rework

Tool-supported assistance in forward engineering will not suffice in typical software development projects. They will additionally require support for modifications to already established decisions. Thus, the possibility and effects of modifications, extensions and deletions of supported method elements are to be considered carefully. Notably the handling of ripple effects is important (are they seen at all, is the developer warned about them, are some of them automatically taken care of, has the developer a chance to intervene).

Furthermore, automatic detection and reporting of possible inconsistencies represent valuable features in supporting the developer. These consistency checks are helpful already during forward engineering but they become detrimental during maintenance operations, be it in the maintenance phase or be it during roll-backs of forward development. The handling of inconsistencies is important though. Everybody would agree that none of them should finally remain. However, it would be likewise a burden for the developer, if he cannot sketch and experiment. Good CASE tools will guide and foster creativity, poor ones will turn out to be creativity killing procrustean beds. Good ones allow to consider variants and to move back and forward whenever necessary. Although all these aspects do not seem to be very visible when choosing a tool, ignoring them can substantially handicap developers in their flexible way of work.

6. Functional Support

Especially with integrated CASE tools, the interplay between the single tool components as well as the 'intelligence' of the tool component itself are essential requirements which distinguish proper CASE tools from conventional drawing tools. Formal checks and consistency checks within and between methods and tool components and the support of view integration within a modelling technique are characteristics which are indispensable for any sophisticated CASE tool.

7. Usability

The CASE tool should prove usable in the sense of providing an acceptable degree of performance, a comfortable user-interface and a tool philosophy which is as open as possible to gain flexibility (by exchangability of underlying data base management system, extendibility with third-party modelling or analysis components, customizability with user-specific tool extensions). A comfortable user-interface requires not only essential prerequisites like a common look and feel of all integrated tool components, an easy to learn approach in working with the tool is as important.

Besides these issues that are somehow obvious, there are also other usability concerns which figure only after gaining experience with the tools. Foremost amongst them a seemingly trivial one: Does the tool provide an undo-function and how does it work (toggle or stack). Lack of it is a severe nuisance. Other aspects on the usability dimension relate to the support of teamwork. Even when a tool provides multi-user- and multi-role-support (1.2, 1.3) that does not say that working in teams is easy. How much structure for the team work is assumed? Is it possible that one team-member can continue the work where an other has terminated. To which degree is concurrent work supported (locking strategies)? Another nuisance, which certainly remains a problem invariable of the particular tool is how to obtain hard-copies of complex diagrams. We might assume, that the support of interactive work (browsing, zooming, setting user-defined presentation defaults) helps to cope with the problem of sizable diagrams. If one can work in several windows concurrently (having several levels of abstraction 'on screen') will be adequate for the interactively working developer. But how can he bring the result of his work to paper in such a way that it can be thoroughly reviewed by a third person or by a review team? All these usability issues seem to be less important at first glance, but prove essential during daily work with the tool.

8. Code Generation

Some interesting questions concerning code generation are: What documents are prerequisites for code generation? Which adaptations on what documents have to be made in order to be able to generate code? What about tool-supported modifyability of generated code? Is it possible to include external libraries? How maintainable is the generated code? What about the quality of the generated code in comparison to manually written code (naming conventions, structuring, ...)? Is the generated code independently executable or is a runtime version of the CASE environment necessary? Is prototyping supported by the possibility of testing and executing parts of the generated code within the tool environment?

9. Additional Features

An open-ended list of additional features include management support (concerning planning, control, estimation, ...), support of versioning, report generation, import as well as export interfaces, and quality assurance, process guidance and process tracking.

These items are certainly subject to discussion, since they tap a diverse set of issues, most of which will not apply for a given tool. Whether management support (9.1) applies at all for the tool depends on the extent it covers the system life cycle. If none is present, one might at least question to which extent a dedicated project management tool can be connected in such a way that no (or little) duplicated data entries are needed.

10. Further Support by the System

Problem solving support for developers as well as support in tool administration are two families of criteria which are worth mentioning separately as an independent item in our check list. Support in problem solving would comprise the existence of understandable, complete and helpful manuals, plausible and well-described default strategies, competent hotline support, and reliable tool architecture (stable in case of hardware as well as user failures). Support for the tool administrator could concern installation management (first installation, customization, upgrades), model management as well as reuse management (in making available certified classes/objects to the public, providing extendable libraries, ...).

4. Evolution of the Framework

The basic ideas to establish an evaluation framework for CASE tools originated from a student project for comparing four different CASE tools that support a classical software development process. These four tools were selected according to their share of the middle-european market as well as to their level of maturity in supporting conventional software development methods (ER, SA/SD). Four teams of three to four students - each of them already well-educated in software analysis & design as well as in database design - had to work with one particular CASE tool each. After an introductory phase getting acquainted with the tool itself, they had to use their tool in a software development project dealing with a common problem domain (conference management system). The project went beyond a typical student project as all team members were not only already familiar with the supported methods but also most problems of novice users could be identified previously during a small-sized 'toy'-project carried out prior to the proper development project.

The primary goal of the project was the assessment of the four tools according to given evaluation criteria. These criteria were elaborated independently of the examined tools and can be regarded as a-priori criteria. During the intensive contact with the tools, additional criteria - which previously were regarded as less important - were identified. Most of these new criteria proved to be requirements which were not satisfied by one or the other tool investigated (e.g. missing undo-function, problems with hard-copy documentation of designs, ...). Both a-priori as well as a-posteriori criteria were collected, refined, completed and assigned to categories resulting in the evaluation framework presented in the annex. Some qualitative results of the original student projects are published in [Mitt93].

5. Related Work

The evaluation method in [duPl93] emphasizes also on the assessment of classical CASE tools. It particularly investigates CASE support of life cycle phases (in some more detail than in our 3rd dimension) and, furthermore, discusses a special evaluation procedure. Further evaluation criteria sets include management, configuration, technical and usage criteria all of them organized in a rather flat hierarchy of only two levels.

Motivated by our framework for the assessment of classical CASE tools, [Kasc95] propose criteria to evaluate object-oriented analysis tools as well as the underlying methods. In [Kasc96] that framework is applied to a tool (OMTool) supporting OMT [Rumb91].

The evaluation presented in [Chur95] focuses on object-oriented CASE tools supporting the development of embedded real-time systems in the domain of telecommunications. The primary evaluation criteria was the degree of automatic code generation. They planned to develop a sample product with the tool selected, but no tool (out of the evaluated 14 OOCASE tools) could pass the evaluation procedure successfully. Hence, the sample project had to be canceled.

From all the attempts in evaluating object-oriented CASE tools and from our own experience in the area of object-oriented software engineering, we conclude that at present the underlying object-oriented methods have not yet reached the level of maturity as did classical methods (entity relationship modelling, structured analysis and design techniques). As object-oriented approaches are still that young and inhomogeneous, a method for evaluating OOCASE tools cannot seriously be suggested. However, once decided about the proper object-oriented method to apply, our evaluation framework (particularly 4th dimension) can easily be extended with essential characteristics of the relevant method.

6. Concluding Remarks

Selecting a CASE tool and having it become operational in software development is a major and almost irrevocable decision. If possible, it should be based on actual experience. If this is unfeasible, checklists can provide a help for identifying at least the most crucial aspects; they are to be customized to the particular circumstances of the developer. Here, we motivated and explained one such checklist, that resulted from an in depth comparison of four powerful integrated CASE tools.

As cautionary remark we would add though, that this (and similar) checklist(s) is tool related. However, the key decision about how one wants to organize software development and which development strategies and development methods one plans to use have to be made prior to the tool decision. These more fundamental decisions ought to be reviewed during the tool selection process though.

References

[Chur95] T. Church, P. Matthews: An Evaluation of Object Oriented CASE Tools: The Newbridge Experience, Proc. Seventh International Workshop on Computer Aided

Software Engineering (CASE 95), Toronto, 1995, pp. 4-9

[duPl93] A.L. duPlessis: A method for CASE tool evaluation, Information & Management, Vol. 25, No. 2, Aug. 1993, pp. 93-102

[Flec93] T. Flecher, J. Hunt: Software Engineering and CASE - Bridging the Culture Gap, McGraw Hill, 1993

[Hoch96] E. Hochmüller, C. Kohl, H.C. Mayr, R. Mittermeir: CASE-Tools - Perspektivenwandel am Weg vom Anfänger zum Experten, Informatik/Informatique, Schweizer Informatiker Gesellschaft, Heft 3, Juni 1996

[Hump92] W.S. Humphrey: Managing the Software Process, Addison Wesley, 1992.

[Kasc95] R. Kaschek, H.C. Mayr, C. Kohl, C. Kop: Characteristics of OOA Methods and Tools - A First Step to Method Selection in Practice, Technical Report 95/1, University of Klagenfurt, 1995

[Kasc96] R. Kaschek, H.C. Mayr: A Characterization of OOA Tools, Proc. 4th International Symposium on Assessment of Software Tools, Toronto, May 1996

[Mitt93] R. Mittermeir, E. Hochmüller, K. Kienzl, E. Kofler, H. Steinberger: CASE-Tool Evaluation in einem Multi-Development Projekt, Proc. ADV-Tagung CASE 93, Wien, Okt. 1993, pp. 6-26

[Rumb91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorensen: Object-oriented Modelling and Design, Prentice Hall, 1991

Acknowledgement

We are grateful to the tool vendors/distributors of IEW, Innovator, Oracle*CASE and ProMod-PLUS for their cooperation and support during the evaluation project. The authors also want to express their acknowledgement to Klaus Kienzl for his contributions to the evaluation project.

ANNEX

Evaluation Criteria

1. General software evaluation criteria

- 1.1. required hardware and software configurations and platforms
- 1.2. multi-user support
- 1.3. multi-role support (administrator, manager, user, ...)
- 1.4. interfaces to other tools and programming languages
- 1.5. market penetration / share of market
- 1.6. tool documentation
  - 1.6.1. overview (see also 2.1.)
  - 1.6.2. reference manual (see also 10.1.1.)
- 1.7. support by distributor
- 1.8. conditions
  - 1.8.1. prices
  - 1.8.2. maintenance
  - 1.8.3. upgrades
  - 1.8.4. error removal
- 1.9. installation effort

2. Support for initial usage

- 2.1. tutorial - manual
  - 2.1.1. structure
  - 2.1.2. where to start
  - 2.1.3. readability
  - 2.1.4. completeness
- 2.2. tutorial - demo examples
  - 2.2.1. existence
  - 2.2.2. connection to manuals
- 2.3. courses
  - 2.3.1. on method
  - 2.3.2. on tool
- 2.4. generally available literature
  - 2.4.1. method
  - 2.4.2. notation
  - 2.4.3. tool
- 2.5. self-explaining tool
- 2.6. adequate help function for beginners

3. Extent of support over development process

- 3.1. early phases / requirements analysis
- 3.2. analysis: data-oriented and function-oriented
- 3.3. design: data-oriented and function-oriented
- 3.4. structured programming
- 3.5. test
- 3.6. configuration management
- 3.7. project management

4. Support of forward engineering

- 4.1. tool aspects
  - 4.1.1. application emphasis
    - 4.1.1.1. just data base design
    - 4.1.1.2. just software engineering
    - 4.1.1.3. both
  - 4.1.2. possibility of free comments (annotations)
  - 4.1.3. point of no return
    - 4.1.3.1. for decision about programming language
    - 4.1.3.2. for decision about data types
  - 4.1.4. support of model variants
- 4.2. method aspects
  - 4.2.1. kind of methods supported
    - 4.2.1.1. static modelling
    - 4.2.1.2. dynamic modelling of objects/functions
  - 4.2.2. existing methods (exemplary)
    - 4.2.2.1. entity relationship modelling
      - 4.2.2.1.1. completeness
      - 4.2.2.1.2. cardinalities
      - 4.2.2.1.3. integrity constraints
      - 4.2.2.1.4. association types (n-ary associations, is-a, aggregations, n:m associations)
      - 4.2.2.1.5. expressing association attributes
      - 4.2.2.1.6. expressing multi-attribute keys
      - 4.2.2.1.7. expressing candidate keys
      - 4.2.2.1.8. relationship between primary key and candidate keys
    - 4.2.2.2. view concept
    - 4.2.2.3. representation of structural hierarchies
      - 4.2.2.3.1. high-level representation (process/function hierarchies)
      - 4.2.2.3.2. low-level representation (module hierarchies)
      - 4.2.2.3.3. very-low-level representation (semi-formal code)
    - 4.2.2.4. data flow modelling (structured analysis)
      - 4.2.2.4.1. access to attributes of entity relationship diagram
      - 4.2.2.4.2. definition of derived attributes
      - 4.2.2.4.3. temporary definition of auxiliary variables
    - 4.2.2.5. structure charts (structured design)
      - 4.2.2.5.1. control flow
      - 4.2.2.5.2. data flow
      - 4.2.2.5.3. predefined elements
      - 4.2.2.5.4. connectors (currently invisible components)
      - 4.2.2.5.5. integration of one process in different charts
      - 4.2.2.5.6. factoring
      - 4.2.2.5.7. rebalancing
      - 4.2.2.5.8. aggregation possibility for data flows
  - 4.2.3. differences/extensions to published methods and notations
  - 4.2.4. integration aspects
    - 4.2.4.1. common data pool
    - 4.2.4.2. consistency
    - 4.2.4.3. feed forward
      - 4.2.4.3.1. from early phases to static model
      - 4.2.4.3.1. from static model to dynamic model
    - 4.2.4.4. automatic generation of representations from existing data
  - 4.2.5. view integration (see also 6.6)
- 4.3. kind of target code

    4.3.1. kinds of programming languages and/or 4GL
    4.3.2. development from scratch - generation of code in
        4.3.2.1. data definition language (DDL)
        4.3.2.2. data manipulation language (DML)
        4.3.2.3. programming language (3GL)
        4.3.2.4. end-user language (4GL)

5. Maintenance, adaptation, rework

- 5.1. feed backward of modifications
    - 5.1.1. within one method
    - 5.1.2. across methods
- 5.2. feed forward of modifications
    - 5.2.1. within one method
    - 5.2.2. across methods
- 5.3. kind and extent of warnings about inconsistencies/invalidities because of maintenance
- 5.4. kind and extent of warnings about distant effects because of maintenance activities
- 5.5. kind of failure avoidance
- 5.6. to which extent are milestones also 'points of no return'
- 5.7. changeability of
    - 5.7.1. association matrices
    - 5.7.2. process trees
    - 5.7.3. entity names
    - 5.7.4. attribute names
    - 5.7.5. relationship names
    - 5.7.6. data flow names
    - 5.7.7. layout
    - 5.7.9. other ...
- 5.8. extensibility of
    - 5.8.1. ER diagram
        - 5.8.1.1. additional entities
        - 5.8.1.2. relationships between already existing entities
        - 5.8.1.3. additional attributes
        - 5.8.1.4. additional key attributes
    - 5.8.2. data flow diagram
        - 5.8.2.1. new processes
        - 5.8.2.2. new arrows
        - 5.8.2.3. reversing of arrows
- 5.9. deletion of
    - 5.9.1. processes
    - 5.9.2. functions
    - 5.9.3. entities
    - 5.9.4. attributes
    - 5.9.5. relationships
    - 5.9.6. data flows
    - 5.9.7. modules
    - 5.9.8. formal parameters
    - 5.9.9. other ...
- 5.10. manipulations in data dictionary and their effect on other areas
- 5.11. reusability

5.11.1. inter-project reuse of results
5.11.2. intra-project reuse of results

6. Functional support

6.1. formal checks within a method / tool component
6.2. formal checks between methods / tool components
6.3. assurance of formal correctness by development primitives
6.3.1. within one level
6.3.2. between different levels
6.4. effects of (6.4.x) within actual / on previous / for following modelling phases
6.4.1. modifications
6.4.2. extensions
6.4.3. deletions
6.5. consistency checks
6.5.1. on demand
6.5.2. automatically invoked
6.6. support of view integration
6.6.1. support of bottom-up data modelling
6.6.1.1. contextual and graphical integration of views
6.6.1.2. support on resolving partial overlaps of attributes
6.6.1.3. support in dealing with homonyms/synonyms
6.6.2. support of bottom-up functional modelling
6.6.2.1. generalization support
6.6.2.2. support in resolving naming conflicts
6.6.3. primitive integration support of independently developed models
6.6.3.1. extensibility of models by (rather closed) partial models
6.6.3.2. extensibility/adaptation of graphics after integration of partial models

7. Usability

7.1. performance aspects
7.1.1. speed of tool
7.1.1.1. when (re-)booting
7.1.1.2. with selected operations
7.1.2. minimal storage space per application
7.2. interactive support
7.2.1. homogeneity (common user interface of all tool components)
7.2.2. learnability
7.2.2.1. transparency (development of 'where is what' feeling)
7.2.2.2. orthogonality (no partial function overlapping)
7.2.2.3. completeness of functional possibilities in different usage modes
7.2.3. existence and clearly defined scope of undo-function
7.2.4. lucidity of display
7.2.5. customizable display (redefinability of default options)
7.2.6. representation aspects
7.2.6.1. layouting
7.2.6.2. binding of texts and graphics
7.2.7. reports
7.2.7.1. simplicity
7.2.7.2. true to scale in comparison to display layout
7.2.8. zooming/fisheye

7.2.9. support of modularity
7.2.10 support for switching between different tasks
7.2.10.1. between different levels
7.2.10.2. between different scopes of one level
7.2.11. user guidance - user freedom / method guidance - method compulsion (forced trail, guided trail, recommended trail, no trail)
7.2.12. return possibility
7.2.13. support of teamwork
7.3. openness
7.3.1. open interface to data storage component
7.3.2. combinability with third-party modelling or analysis components
7.3.3. definability of user-specific tool extensions
7.3.3.1. functional customization
7.3.3.2. macros

8. Code generation
8.1. support of prototyping
8.2. ease of code generation
8.3. prerequisites for code generation
8.4. tool-supported modifyability of generated code
8.5. incorporation of external libraries
8.6. quality of generated code
8.7. portability of generated application to other platforms

9. Additional features
9.1. management support
9.1.1. progress control
9.1.2. progress planning
9.1.3. cost estimation for future phases
9.1.4. other ...
9.2. version management
9.3. report generator
9.3.1. quantity versus quality of reports
9.3.2. import of externally generated representations (graphics, texts, binaries)
9.4. import/export interfaces
9.4.1. from/to other systems
9.4.2. from/to other projects
9.5. quality assurance ISO 9001, ISO 9000-3
9.6. process documentation
9.7. other ...

10. Further support by the system
10.1. problem solving support for developer
10.1.1. manuals
10.1.1.1. understandability
10.1.1.2. completeness
10.1.1.3. quality of reference manual (hit-ratio on specific search)
10.1.2. plausible default strategies
10.1.3. hotline

    - 10.1.4. protection on failure
        - 10.1.4.1. of developer
        - 10.1.4.2. of hardware
- 10.2. support for tool administrator
    - 10.2.1. installation management
    - 10.2.2. model management
    - 10.2.3. reuse management