BSSC(94)1
May 1994

Guide to the selection and use of CASE tools

Prepared by:
ESA Board for Software
Standardisation and Control
(BSSC)

european space agency / agence spatiale européenne
8-10, rue Mario-Nikis, 75738 PARIS CEDEX, France

DOCUMENT STATUS SHEET

DOCUMENT STATUS SHEET			
1. DOCUMENT TITLE: BSSC(94)1			
2. ISSUE	3. REVISION	4. DATE	5. REASON FOR CHANGE
1 1	0 1	17/05/1994 31/10/1994	First issue Editorial changes

Approved, 17th May 1994
Board for Software Standardisation and Control
C.Mazza, Chairman

TABLE OF CONTENTS

PREFACE

In 1992, the Director General of ESA requested the Board for Software Standardisation and Control (BSSC) to report on the use of Computer Aided Software Engineering (CASE) tools by the Agency. The BSSC issued a report recommending that:

- ESA should not impose a single set of CASE tools for all its software projects;
- commercially available CASE tools should be used;
- every proposal for the use of CASE tools should include a training plan.

The Director General endorsed these recommendations.

The fact that no single CASE tool or toolset can be recommended for all ESA software projects implies that projects must define their own requirements for CASE support and obtain the tools that meet their needs. The BSSC have produced this guide to help projects do this by discussing:

- CASE concepts, benefits and strategy for use;
- where CASE tools can be applied in the software life cycle;
- how CASE tools can be integrated into a Software Engineering Environment;
- how to evaluate and select CASE tools.

The following past and present BSSC members have contributed to the production of this document: Carlo Mazza (chairman), Gianfranco Alvisi, Michael Jones, Bryan Melton, Daniel de Pablo, Adriaan Scheffer and Richard Stevens. The BSSC wishes to thank Jon Fairclough and Colin Rolls for their assistance in the development of this document.

The BSSC intends to issue this guide as a PSS document after a period of trial use and feedback. Requests for clarifications, change proposals or any other comment concerning this guide should be addressed to:

BSSC/ESOC Secretariat
Attention of Mr C Mazza
ESOC
Robert Bosch Strasse 5
D-64293 Darmstadt
Germany

BSSC/ESTEC Secretariat
Attention of Mr B Melton
ESTEC
Postbus 299
NL-2200 AG Noordwijk
The Netherlands

This page is intentionally left blank

CHAPTER 1
INTRODUCTION

1.1 PURPOSE

ESA PSS-05-0 describes the software engineering standards to be applied for all deliverable software implemented for the European Space Agency (ESA) [Ref 1].

Guidelines on the application of the standards are provided in the series of guides described in ESA PSS-05-01, "Guide to the Software Engineering Standards" [Ref 2 to 9]. This document has been produced to provide organisations and software project managers with guidelines for the use of Computer Aided Software Engineering (CASE) tools.

Information about tools present in the ESA PSS-05 guides has been collected into this document to provide a self-contained, integrated view of CASE. The concepts and terms provided in this guide provide a consistent and logical framework for using CASE tools.

This guide does not identify specific CASE tools or make recommendations about which tools to use. Organisations and project managers should evaluate and select tools with the aid of this guide and decide what is best for them.

1.2 OVERVIEW

Chapter 2 defines computer aided software engineering and discusses the benefits and the strategy of its use. Chapter 3 identifies the types of CASE tools that can be used to support the life cycle tasks defined in ESA PSS-05-0. Chapter 4 discusses Software Engineering Environments. Chapter 5 provides guidelines for the evaluation and selection of CASE tools. Appendix A contains an extensive glossary of CASE terms. Appendix B contains a list of references and Appendix C an index.

This page is intentionally left blank

CHAPTER 2
CONCEPTS, BENEFITS AND STRATEGY

2.1 INTRODUCTION

Software tools have existed for as long as software itself. The concept of "Computer Aided Software Engineering" (CASE) was formulated in the early 1980's to describe the tools used for editing data flow diagrams and entity relationship diagrams. These "graphical editors" enforce the rules of the methods that they support. This enables software engineers to construct models that are consistent and syntactically correct. The graphical editors are sometimes combined with a database, usually called a "repository", that stores the products of the tools.

The meaning of the term "CASE tool" has now broadened to include any tool that supports software development. A CASE tool may apply to one or more activities of the software life cycle. CASE tools may be integrated to make a "Software Engineering Environment" (SEE) that ideally provides full support for all software engineering activities. SEEs are discussed in Chapter 4.

Section 2.2 defines what a CASE tool is. Section 2.3 discusses the benefits of using CASE tools. Section 2.4 outlines the strategy for the successful use of CASE tools.

2.2 CONCEPTS

A Computer Aided Software Engineering (CASE) tool is "a software tool that aids in software engineering activities, including but not limited to requirements analysis and tracing, software design, code production, testing, document generation, quality assurance, configuration management and project management" [Ref 10].

All CASE tools should:

- support the application of a software method, programming language or procedure;
- check that the rules of the method, programming language or procedure are obeyed and inform the tool user when they are not;
- store the software products generated by the application of the method, programming language or procedure.

2.3 BENEFITS

The justification for using CASE tools is to improve quality and productivity. Quality improvements are measured by reductions in the number of faults and increased conformance to user requirements. Productivity improvements are measured by reductions in the effort required to develop and maintain the software.

Examples of quality improvements are:

- fewer errors in software models when the rules of the method are enforced by modelling tools;
- fewer errors in source code when static analysis tools are used to enforce coding standards;
- fewer configuration management errors because the tools ensure that the correct procedures are followed every time;
- more errors being captured by the verification and validation process, because tracing tools ensure that all requirements are implemented and because test tools help ensure that the software is thoroughly tested.

Examples of productivity improvements are:

- software models can be created and modified more quickly;
- problems in code can be more quickly diagnosed with debuggers;
- performance problems can be quickly pinpointed with performance analysers;
- regression testing can be much faster with tools that run the tests, retrieve expected output data for comparison, compare actual output data with expected, and then report the result.

In summary, there are many mechanical tasks in software engineering that are simple in concept but complex in detail, because of the need to be consistent, complete and accurate in activities that have many small steps. Product quality is significantly reduced if they are not done well. CASE tools improve quality and productivity by automating these tasks.

An initial drop in productivity should be expected when software engineers start to use CASE tools, particularly when there is no experience in the method that the tools support. The loss of output can be minimised with proper training and will be more than recovered later. However, without training, productivity can fall and never recover!

2.4 STRATEGY

To realise the benefits of CASE, organisations should adopt the following strategy :

1. define the software engineering process, then;
2. choose methods suitable for each activity in the process, then;
3. evaluate and select the tools to support the methods, then;
4. train software engineers to use the methods and tools.

The ESA PSS-05-0 standard defines ESA's software engineering process. At the top level this process consists of a life cycle containing the following phases:

- user requirements definition phase;
- software requirements definition phase;
- architectural design phase;
- detailed design and production phase;
- transfer phase;
- operations and maintenance phase.

Procedural activities in each phase of the life cycle are:

- software project management,
- software configuration management,
- software verification and validation,
- software quality assurance.

Projects should define a process model that details the inputs, outputs, methods, procedures, roles and responsibilities associated with these phases and activities. Figures 2.4A, B and C are example process models showing the inputs and outputs of each activity in the SR, AD and DD phases of a project.

Figure 2.4A: SR phase process model

Figure 2.4B: AD phase process model

Figure 2.4C: DD phase process model

A method defines the rules and procedures associated with an activity. For example Structured Analysis is a method for constructing a logical model and Performance, Evaluation and Review Technique (PERT) is a method for project planning. The choice of method depends upon the application domain (e.g. different methods are normally used for designing real time systems and non-realtime systems). There are many methods in the software engineering literature. Projects should use methods that are well documented and proven through industrial experience, and not invent new methods unless there is no alternative. New methods lack tool support and are more likely to contain errors. The ESA PSS-05 series of guides discusses the methods for supporting the activities defined in the ESA Software Engineering Standards.

The ESA PSS-05 guides discuss the general capabilities of tools. Specific tools are not discussed as ESA has a policy of not identifying commercial products. ESA PSS-05-0 does not state that specific methods or tools must be used, but it strongly recommends that projects use CASE tools for:

- constructing the logical model in the software requirements definition phase (Part 1, Section 3.3.1)
- constructing the physical model in the architectural design phase (Part 1, Section 4.3.1)
- configuration management (Part 2, Section 3.2.2).

There are many other activities in the software life cycle that may be supported by tools. Word processors, compilers and linkers are obvious necessities that need no specific recommendation.

The evaluation and selection of tools can be a complex process because of the wide range of tools now available. A systematic process of evaluation and selection, such as the process described in Chapter 5, is strongly recommended.

New methods and tools require time and training to assimilate. Organisations adopting new methods and tools should train their staff. Project managers should assess training requirements when planning the

project and set aside resources for it if necessary. Lack of training in methods and tools is a risk.

CHAPTER 3
CASE TOOLS IN THE SOFTWARE LIFE CYCLE

3.1 INTRODUCTION

This chapter defines the functions of the different types of CASE tools. Figure 3.1 identifies the tools and shows where they may be used in the life cycle. Software production tools support the production of the URD, SRD, ADD, DDD, code and SUM. The other four categories of tools support the procedural activities common to each phase.

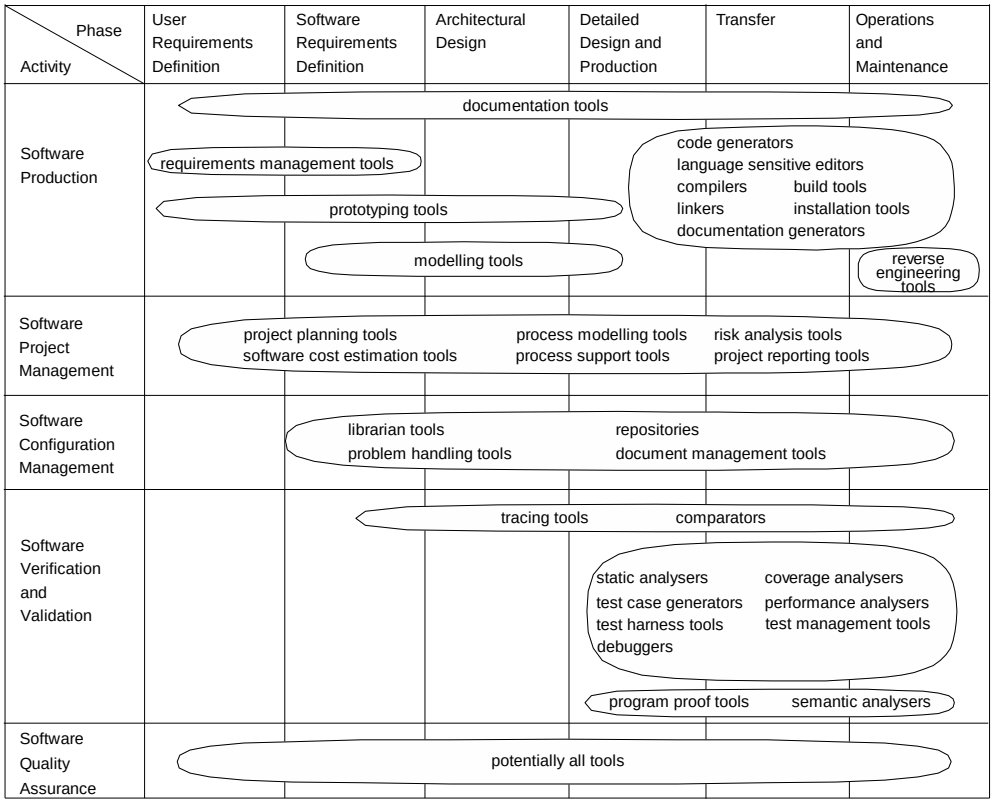

Figure 3.1: Tool use in the software life cycle

The ESA PSS-05 series of guides to software engineering [Ref 2 to 9] discuss tools in detail. This chapter summarises those guidelines. Section 3.2 describes tools for supporting software production, phase-by-phase. Section 3.3 describes tools for supporting the procedural activities.

3.2 TOOLS FOR SOFTWARE PRODUCTION ACTIVITIES

The tools for software production are introduced in order of the stage in the life cycle where they are first used. A tool is reintroduced if additional capabilities are required in the phase after it is first used.

3.2.1 User Requirements Definition Phase

The purpose of the user requirements definition phase is to refine an idea of a task to be performed, using computer equipment, into a definition of what is expected from the computer system [Ref 1]. Tools are useful in the user requirements definition phase for:

- documentation;
- managing user requirements;
- constructing prototypes.

3.2.1.1 Documentation Tools

Documentation tools include:

- text editors;
- word processors;
- desktop-publishing systems.

A project should standardise on a single documentation tool for producing documents and plans. The documentation tools should allow the creation of structured documents and be able to interface with as many of the other tools as possible. The documentation tools should be able to import and export text in a standard format (e.g. ASCII).

Graphics tools may be part of the documentation tool or separate from it. These should be able to import and export graphic data in a standard format (e.g. CGM). Projects should standardise on a single graphics tool for producing diagrams for documentation.

3.2.1.2 Requirements Management Tools

Requirements management tools should support the:

- entry, inspection, update and deletion of requirements;
- attachment of attributes to requirements;
- control of access to requirements;

- tracking of changes to requirements;
- search and retrieval on requirement text or attribute value;
- definition of dependencies for the purposes of structuring and tracing.

Requirement management tools are sometimes called "requirements engineering" tools. For medium to large projects, a tool to manage requirements is invaluable. A thousand or more requirements may be defined in a large project, at which point manual requirements management becomes both error-prone and time-consuming.

3.2.1.3 Prototyping Tools

Tools that help construct prototypes can be useful for exploring the user requirements. Prototype menus, forms and reports can help define the user requirements for human computer interaction for example. Minimal programming should be required to create an effective prototype.

Simple prototyping tools can produce "static" prototypes that mimic the look and feel of the real software. More advanced prototyping tools will contain commercial software packages and code generators to implement software functions, such as data retrieval. The prototyping tools should not be expected to produce software that has the required performance.

3.2.2 Software Requirements Definition Phase

The purpose of the software requirements definition phase is to analyse the statement of user requirements and produce a set of software requirements as complete, consistent and correct as possible [Ref 1]. Tools are useful in the software requirements definition phase for:

- constructing the logical model;
- constructing prototypes;
- managing software requirements.

3.2.2.1 Modelling tools

The logical model defines what the software must do. Methods and tools should be used to construct it. The methods identified in ESA PSS-05-03 Guide to the Software Requirements Definition Phase [Ref 4] are:

- functional decomposition;
- structured analysis;

- object-oriented analysis;
- formal methods;
- Jackson System Development [Ref 24].

Some of these methods are really a class of methods. Structured analysis includes the methods of De Marco [Ref 21], Ward-Mellor [Ref 23], SSADM [Ref 22] and SADT [Ref 25]. Object-oriented analysis includes the methods of Coad-Yourdon [Ref 26], Rumbaugh [Ref 27], Shlaer-Mellor [Ref 29, 30] and Booch [Ref 28].

Each method can include several techniques. For example the Yourdon method includes data flow diagrams and data dictionary. Ideally the CASE tool should support all the techniques of the chosen method. The tool should enforce the rules of the method.

When two or more tools are required to support a method, the tools should be well integrated and able to share data. Poorly integrated tools make a consistent logical model difficult to construct. For example, a tool to support data flow diagrams will normally have an underlying data dictionary. If a separate tool is used for entity-relationship diagrams, the data dictionary should nevertheless be common. Without this, inconsistencies will appear in the logical model, even though the tools make their own parts of the model consistent.

Modelling tools should be able to interface to the documentation tool used for preparing the software requirements document. Export or linking of diagrams produced by the tool into the SRD should be possible.

Some modelling tools support "animation", i.e. simulation of behaviour. This can be a very effective method for verifying the logical model. Animation requires that the tool supports the construction of a dynamic model. State transition diagrams are the most common technique for dynamic modelling.

Formal methods for constructing a logical model in mathematical terms are available. Examples of formal methods are LOTOS, Z and VDM [Ref 32]. Some of these methods, such as LOTOS, are supported by tools. The tools need to support the construction of mathematical expressions and to be able to check the syntax of an expression.

3.2.2.2 Requirements Management Tools

Requirements management tools, described in Section 3.2.1., can also be useful for handling software requirements. Ideally the tool should interface to the modelling tool so that the functional requirements and interface requirements can be derived directly from the model. It should be possible to extract the software requirements from the tool for insertion into the software requirements document.

A Requirements Management tool should help trace software requirements to user requirements and vice-versa, thus providing the compliance table required in the Software Requirements Document.

3.3.3.13.2.3 Architectural Design Phase

The purpose of the Architectural Design Phase is to define a collection of software components and their interfaces to establish a framework for developing the software [Ref 1] Tools are useful in the architectural design phase for constructing the physical model and prototyping.

3.2.3.1 Modelling Tools

The physical model is the core of the architectural design. Methods and tools should be used to construct it. The methods identified in ESA PSS-05-04 Guide to the Software Architectural Design Phase [Ref 5] are:

- structured design;
- object-oriented design;
- formal methods;
- Jackson System Development [Ref 24].

The first three of these methods are classes of methods. Structured design includes the methods of Yourdon [Ref 33], Ward-Mellor [Ref 23], SSADM [Ref 22] and SADT [Ref 25]. Object-oriented design includes the methods of Booch [Ref 28], HOOD [Ref 31], Coad-Yourdon [Ref 26], Rumbaugh [Ref 27] and Shlaer-Mellor [Ref 29, 30].

Each method can include several techniques. For example the HOOD method includes design process tree diagrams and object definition skeletons. Ideally the modelling tool should support all the techniques of the chosen method. The tool should enforce the rules of the method selected.

If two or more tools are required to support all the techniques of a method, then it is important that the tools be well integrated and able share data. Poorly integrated tools make a consistent physical model difficult to construct.

Modelling tools should be able to interface to the word processor used for preparing the architectural design document. Export or linking of diagrams produced by the tool into the ADD should be possible.

Modelling tools need to be able to interface with the tools used for constructing the logical model. Some methods, such as structured design, provide techniques for transforming a logical model into a physical model, for example. Even if direct transformation is not possible, the capability to link components of the physical model to functions, data and objects in the logical model should be provided.

Some modelling tools support "animation", i.e. simulation of behaviour (see Section 3.2.2.1). This can be a very effective method for verifying the physical model.

3.2.3.2 Prototyping Tools

Tools can also be used for constructing experimental prototypes to test the feasibility and performance of the design. Two designs may be equally feasible on paper and prototyping may be needed to decide which is the better.

3.2.4 Detailed Design and Production Phase

The purpose of the Detailed Design and Production Phase is to detail the design outlined in the architectural design document and to code, document and test it [Ref 1].

Tools are useful in the detailed design and production phase for:

- detailed design;
- code generation;
- editing code;
- compiling;
- linking;
- building;
- documentation generation.

Testing is a major activity in this phase. Testing tools are discussed in Section 3.3.3.

3.2.4.1 Detailed Design Tools

The methods and modelling tools used for architectural design should be used in the DD phase for the detailed design work [Ref 1]. Supplementary tools may be needed to support use of the Program Design Language (PDL).

3.2.4.2 Code Generators

There are two types of code generator:

- modelling tools that can produce skeleton code or "templates" containing module headers and data declarations;
- full code generators that produce all the necessary code from a model.

Full code generators are helpful for producing repetitive code. Database management and human-computer interaction are typical applications of full code generators.

Code produced by full code generators is unlikely to comply with project coding standards and be commented. This may be a problem if software engineers have to examine the generated code to diagnose the cause of a software fault. The relationship between the code and the model may not be obvious. Debuggers, used for examining running code, normally operate at the level of the generated code and not at level of the modelling tool. Problems in understanding the generated code can also arise when software engineers have to write code that interfaces with the generated code.

3.2.4.3 Language-Sensitive Editors

Language-sensitive editors provide statement templates for the programmer to complete, and check that each statement entered is syntactically correct.

Some editors are integrated with librarian tools to allow programmers to "browse" module libraries. A template procedure call for a library module can be copied into the module being coded for completion by the programmer.

Editors may be integrated with the compiler and linker to allow incremental compilation and linking of the code. Not only is the syntax of each line of code checked as it is entered, but also the code is ready to execute almost immediately.

3.2.4.4 Compilers

The choice of compiler on a given platform or operating system may be limited. If there is a choice, aspects to be considered are:

- compliance with language standards;
- efficiency of compiling;
- efficiency of code generated by the compiler;
- integration with other tools (e.g. debuggers and performance analysers).

A precompiler generates code from PDL statements. However it is difficult to trace faults in the source code to PDL statements because debuggers are usually not available for precompilers. This problem is shared with Code Generators (see Section 3.2.4.2).

3.2.4.5 Linkers

As with compilers, the choice of linker on a given platform or operating system may be very limited. The linker should be capable of making standalone or shared executables, and minimise the size of the program executable for efficient loading. Link commands can be very complex, and the linker should accept its parameters from control files as well as the command line.

3.2.4.6 Build tools

Build tools vary from simple procedures that compile and link programs to ‘make ” tools that use information about the history and dependencies of the modules to minimise the build time. For example in most builds it is only necessary to compile the source modules that have changed or are affected by a change (e.g. an include file change may affect many modules). Efficient build tools are essential for reducing the mean time to repair of large systems.

3.2.4.7 Documentation Generators

Documentation generators may produce:

- source module header information from the Detailed Design Document and vice versa;
- help documentation from the Software User Manual and vice versa.

Documentation generators help maintain consistency between code and documentation, and make the process of documentation truly concurrent with the coding.

Code generators (see Section 3.2.4.2) may include tools for automatically generating documentation about the screens, windows and reports that the programmer creates.

3.2.5 Transfer Phase

The purpose of the transfer phase is to install the software in the operational environment and show to the users that it meets their requirements [Ref 1]. Besides tools used in the previous phases, tools are useful in the transfer phase for:

- installing the software;
- acceptance testing the software.

When problems are found, tools from the previous phases will be required to re-specify, re-design, re-code, re-build and re-test the software. Tools for testing are discussed in Section 3.3.3.

3.2.5.1 Installation Tools

Installation tools transfer software from distribution media into the target environment and configure it for the new software. Minimal intervention should be required from the user. The installation tools should not make system changes without permission.

3.2.6 Operations and Maintenance Phase

The purpose of software maintenance is to ensure that the software continues to meet the needs of the end users. All the CASE tools used in the previous phases will be needed. The software engineering environment of the development phase should therefore be retained, sized adequately for

the maintenance team. Consideration should be given to improving the software engineering environment as new tools become available.

The need may arise during the operations and maintenance to "reengineer" the code. Reverse engineering tools may be used to support this activity.

3.2.6.1 Reverse Engineering Tools

Reverse engineering tools reconstruct the software design from the source code. They are useful for:

- understanding poorly documented software;
- keeping documentation up to date with code changes;
- redesigning poor quality software.

Reverse engineering tools may support efficient "navigation" through the software. This helps to identify what parts of the software will be affected by a change.

3.3 TOOLS FOR SOFTWARE PROCEDURAL ACTIVITIES

This section discusses CASE tool support for the activities of:

- software project management;
- software configuration management;
- software verification and validation;
- software quality assurance.

3.3.1 Software Project Management

Software project management is the process of planning, organising, staffing, monitoring, controlling and leading a software project [Ref 12]. Software project management may be supported by:

- project planning tools;
- software cost estimation tools;
- risk analysis tools;
- process modelling tools;
- process support tools;
- project reporting tools.

3.3.1.1 Project Planning Tools

Project planning tools should support the definition of the project activities, the resources they need, and the relationships between them. The tools should use this information to display:

- activity schedules (e.g. Gantt charts, PERT networks, or equivalent)
- resource utilisation in resource profiles and summary tables.

In practice, capturing all the dependencies between software project activities can be very difficult, limiting the usefulness of the tools. However planning tools can be very useful for displaying the schedule, calculating resource usage and checking resource conflicts. The value of project planning tools is also apparent when updating plans or assessing new scenarios (i.e. "what-if" analysis).

3.3.1.2 Software Cost Estimating Tools

Software cost estimating tools can be used to predict labour costs and project duration. Typical inputs are specifications of functions and numbers of lines of code. Accurate software cost estimation is very difficult and tools are only of limited benefit.

Some methods, such as Function Point Analysis (FPA) and Boehm's Constructive Cost Model (COCOMO) use empirical formulae to derive the prediction. Spreadsheets can be used to implement these methods. The tools should be calibrated for the organisation using them, otherwise results are unlikely to be accurate.

Other methods use historical comparisons to arrive at a result. Database management tools and knowledge-based systems can be useful for supporting these methods.

3.3.1.3 Risk Analysis Tools

Risk analysis tools have been applied in other fields, but difficulties in accurately quantifying software project risks limits their usefulness.

One approach to risk analysis is to use heuristics or "rules-of-thumb", derived from experience. Tools are available that have been programmed with the rules related to common risks to a software project. The tools ask a series of questions about the project and then report the most likely risks. Again, these kinds of tools have been rarely used in software development.

3.3.1.4 Process Modelling Tools

A software process model defines the roles, responsibilities, methods, procedures, inputs and outputs associated with software life cycle activities. As explained in Section 2.4, projects must base their software process model upon the ESA PSS-05-0 life cycle model. Projects must define the process in more detail when they make a project plan.

Process modelling methods are relatively new, and the tools that support them are consequently immature. Process modelling tools should:

- support the definition of procedures;
- contain a library of "process templates" that can be tailored to each project;
- make the process model available to a process support tool (see Section 3.3.1.5).

There are few dedicated software process modelling tools. Analysis tools that support the structured analysis techniques of data flow diagrams and entity relationship diagrams can be effective substitutes.

3.3.1.5 Process Support Tools

Process support tools help guide, control and automate project activities. They require that a process model has been formally defined, and made available in a suitable format for input to the process support tools.

Some configuration management tools provide process support functions by automating the exchange of information about software problems and change requests. Groupware products can provide process support functions as they supply information exchange facilities for the members of a workgroup. Project management is one area that could substantially benefit because progress could be controlled and monitored be means of electronic messaging.

3.3.1.6 Project Reporting Tools

Progress tables and charts are used for reporting work package expenditure and presenting current estimates of the resources required for the project. Spreadsheets are useful for constructing them.

Project planning tools (see Section 3.3.1.1) that are capable of recording the actual resources used, and the dates that events occurred, can be useful for project reporting. Such tools should use the information to:

- mark up the Gantt chart to indicate schedule progress;
- constrain replanning.

3.3.2 Software Configuration Management

Software configuration management [Ref 1] is the activity of:

- identifying and defining the configuration items in a system;
- controlling the release and change of these items throughout the system life cycle;
- recording and reporting the status of configuration items and change requests;
- verifying the completeness and correctness of configuration items.

Configuration management may be supported by a single tool or a set of tools such as:

- librarian tool;
- repository tool;
- document management tool;
- problem handling tool.

A toolset needs to be well-integrated because the configuration management functions are highly interdependent.

Configuration management tools should support the security policy of a project by allowing changes to configuration items to be made only by authorised personnel.

3.3.2.1 Librarian Tools

Basic librarian tools store configuration items of the same type in the same library (e.g. source files in text libraries, object files in object libraries), and coordination of the libraries must be done manually.

Basic librarian tools do not allow more than one version of a configuration item to be stored. More advanced librarian tools store multiple versions by keeping a complete copy of one version and the changes

needed to generate all the other versions from it. This minimises storage space requirements

3.3.2.2 Repository Tools

Repository tools should be capable of storing all the versions of every configuration item in the system. They store the history of configuration items and the relationships between them. Repository tools use this information to make rollback and update of software easy and efficient.

3.3.2.3 Document Management Tools

Document management tools should be capable of storing all the versions of every document configuration item in the system. They should be able to manage document images as well as computer files.

3.3.2.4 Problem Handling Tools

Problem handling tools should support the online entry of data describing a problem and the processing of that data by the software development and maintenance teams. They should maintain information about all the problems that have ever been reported. They should provide a variety of ways of viewing the problem information, such as listings of all problems that are open, and configuration items that have open problem reports against them.

3.3.3 Software Verification and Validation

Software verification is ‘the act of reviewing, inspecting, testing, checking, auditing, or otherwise establishing and documenting whether or not items, processes, services or documents conform to specified requirements ’ [Ref 1]. Software validation is ‘the evaluation of software at the end of the software development process to ensure compliance with user requirements ’ [Ref 1].

Software verification and validation may be supported by tools such as:

- tracing tools;
- comparators;
- static analysers;
- test case generators;
- test harnesses;

- debuggers;
- performance analysers;
- coverage analysers;
- test management tools;
- program proof tools;
- semantic analysers.

Tool support for test case generation is currently weak. Tool support is strongest in the areas of comparison, static analysis, running tests, debugging, analysing the results and managing test software. Tools should do most of the mechanical or repetitive work involved in software verification and validation.

3.3.3.1 Tracing Tools

Tracing tools should allow easy and efficient navigation through the documentation and code. The arrows in Figure 3.3.3.1 show the relationships that need to be traced.

Tracing tools, normally applications based upon commercial database management systems, are used to build relationship databases. In addition, the database may form part of, or be integrated with, a repository.

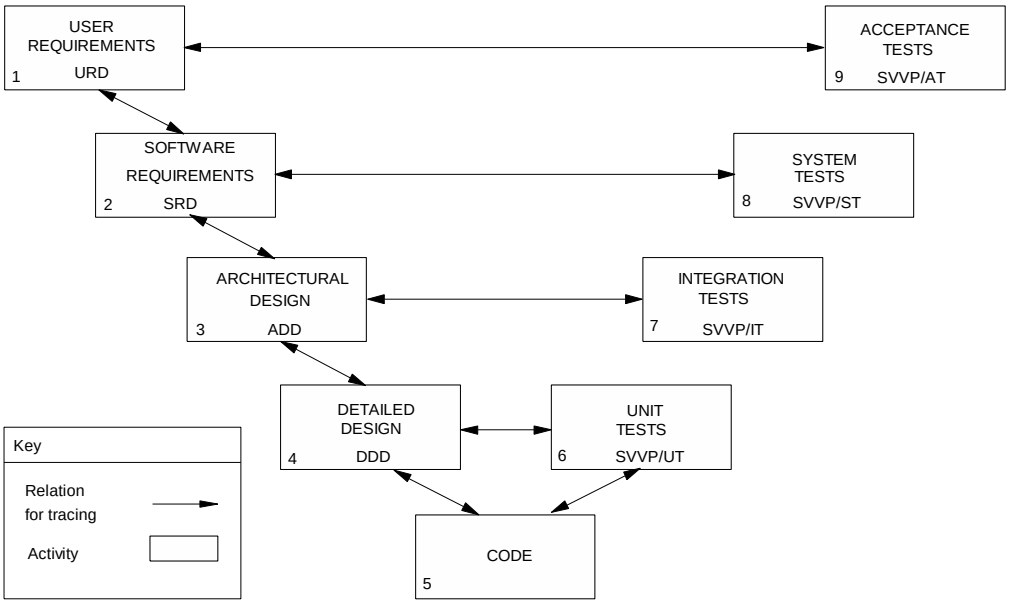

Figure 3.3.3.1: Relationships for tracing

A simpler alternative to using a database management system is to employ the indexing and cross-referencing facilities of a word processor. However with this technique the relationship database (i.e. the cross-reference matrices) becomes fragmented amongst different documents, and maintaining consistency is a problem.

Requirements management tools (see Section 3.2.1.) should support the tracing of software requirements to user requirements and vice-versa. The requirements management tool may also permit tracing requirements to design components, i.e. the requirements management tool also functions as a tracing tool. A single tool that manages the relationships between requirements, software components, test cases, test procedures and test results is preferable.

3.3.3.2 Comparators

A comparator is a software tool used to compare two documents, source code modules, files, or sets of data, to identify commonalities and differences.

Comparators are needed for marking changes to documents, easing verification of the change. A document comparator may be part of the documentation tool.

Comparators are needed in testing to compare actual test output data with expected test output data. They are essential for efficient regression testing.

3.3.3.3 Static Analysers

Static analysis tools measure software complexity (e.g. cyclomatic complexity) and check the code against language standards and coding standards. They should be used to support reviews of code and McCabe's Structured Testing method [Ref 35, 36]. Static analysis tools may provide the control graph for coverage analysis tools (see Section 3.3.3.7).

3.3.3.4 Test Case Generators

Test case generators produce test cases and test data by processing source files. They should support the equivalence partitioning and boundary value analysis methods used in unit testing [Ref 8]. Automated test case generation at integration, system and acceptance test level is not usually possible.

3.3.3.5 Test Harnesses

Test harness tools control the software under test, providing test inputs and reporting test results. They should provide a language for programming test procedures (i.e. a script language).

3.3.3.6 Debuggers

Debuggers can be used for controlling and monitoring the execution of the software. A good screen-based debugger should be used in all software projects for white-box testing and problem diagnosis.

3.3.3.7 Coverage Analysers

Coverage analysers "instrument" the code so that information is collected on the parts of it that are executed when it is run. After the test run, the coverage analyser is used to see what parts of the software under test have been executed. Coverage analysers are essential for verifying statement and branch coverage. They are used mainly in unit testing. Some coverage analysers mark up the control graph produced by the static analyser, providing a graphic display of branch coverage.

3.3.3.8 Performance Analysers

Performance analysers instrument the code so that information can be collected about resources used when it is run. After the test run, the performance analyser is used to analyse the data collected, and to evaluate resource utilisation. Performance analysers are often integrated with coverage analysers to make "dynamic analysers"

3.3.3.9 Test Management Tools

Test management tools provide configuration management functions for the test data and scripts. They enable tests to be setup and run with the minimum of steps and automatically manage the storage of outputs and results.

Significant software costs accrue during the operations and maintenance phase of a project, and much of this is often due to the need to test for regression after each change. Test management tools can be used to reduce the cost of regression testing.

3.3.3.10 Program Proof Tools

Program proof tools may be used to prove the correctness of programs written in languages that can be formally defined. Subsets of Pascal and Ada are examples. There are very few commercial tools.

3.3.3.11 Semantic Analysers

Semantic analysis is the symbolic execution of a program using algebraic symbols instead of test input data. Semantic analysers use a source code interpreter to substitute algebraic symbols into the program variables and present the results as algebraic formulae. Semantic analysers may be useful for the verification of small well-structured programs.

3.3.4 Software Quality Assurance

Software Quality Assurance (SQA) is "a planned and systematic pattern of all actions necessary to provide adequate confidence that the item or product conforms to specified requirements" [Ref 13]. SQA activities evaluate the process by which products are developed [Ref 14].

All the software tools used for other activities in the life cycle can be used to support SQA activities. For example:

- software configuration management tools may be used to obtain a change history (i.e. "audit trail");
- coverage analysers and test management tools may be used to obtain access to the test coverage and test results;
- static analysers may be used to compare code with language and coding standards.

SQA staff may choose to employ tools different to those used by the developers, in order to gain increased independence and objectivity. For example they might choose to use a static analysis tool different to that of the developers because it provides them with the measurements they want, or because they are more familiar with its operation.

CHAPTER 4
SOFTWARE ENGINEERING ENVIRONMENTS

4.1 INTRODUCTION

A Software Engineering Environment (SEE) is "a system that provides automated support of the engineering of software systems and the management of the software process" [Ref 10]. Another name often used for an SEE is "Software Development Environment" (SDE). No distinction is made between the terms SEE and SDE in this guide.

An SEE is an integrated set of CASE tools that supports most, if not all, life cycle activities. The functions and interfaces of an SEE are defined by the process model (see Chapter 2).

Section 4.2 defines SEE design concepts. Section 4.3 introduces SEE Reference Models. Section 4.4 discusses CASE tool integration. Section 4.5 describes the European Space Software Development Environment (ESSDE), a practical example of an SEE implemented by ESA.

4.2 SEE DESIGN CONCEPTS

A Software Engineering Environment normally consists of:

- a framework;
- CASE tools.

A framework provides a set of (relatively) fixed infrastructure capabilities such as user interface, database management and storage. This simplifies tool design and improves usability [Ref 10]. Frameworks normally use the repository tool for database management and storage (see Section 3.3.2.2). They often designate a user interface standard for all tools, such as X Windows, instead of providing user interface capabilities directly.

CASE tools are a necessary part of any SEE. A framework is optional. Tools may exchange information directly without the mediation of a framework, for example. Each tool may have its own user interface and database.

The designer of an SEE must consider [Ref 15]:

- data integration;
- control integration;
- presentation integration.

The degree of data integration measures the ability of the tools to exchange information and use the information exchanged, i.e. it measures interoperability. Data integration requires not only that tools can export and import data to and from other tools, it also requires that the tools understand what the data means, and can use the data correctly. Data integration may be implemented via database linkages, data exchange or a common repository (i.e. shared database).

The degree of control integration measures the ability to combine the functionalities offered by the tools. For example, when a new source module is submitted for inclusion in a master library, the software configuration management tool may use the compiler and the linker tools to generate the corresponding object module and executable program. This implies that the configuration management tool is able to control the compiler and linker.

The degree of presentation integration measures the ability of the user to interact with the tools in similar modes, and is critical to its usability. Presentation integration requires the adoption of a user interface standard, such as "X Windows", to provide a consistent "look and feel".

4.3 REFERENCE MODELS

The design of an SEE should be based upon a standard reference model to make it an "open" system. The SEE Reference Model developed by the European Computer Manufacturers Association and the US. Department of Commerce is shown in Figure 4.3 [Ref 11]. The model defines a framework of common basic services. Tools, such as compilers and linkers, use the services of the framework. They are pictured as fitting into tool "slots", giving this model the nickname of "the toaster model".

The interactions between users and the tools are mediated by the user interface services and coordinated by the process management services. The object management services provide a common repository for sharing tool information. Communication services support the message passing between the objects in each service group. Policy enforcement,

administration and configuration services, not shown, provide access control and system management services. The services are summarised in Table 4.3.

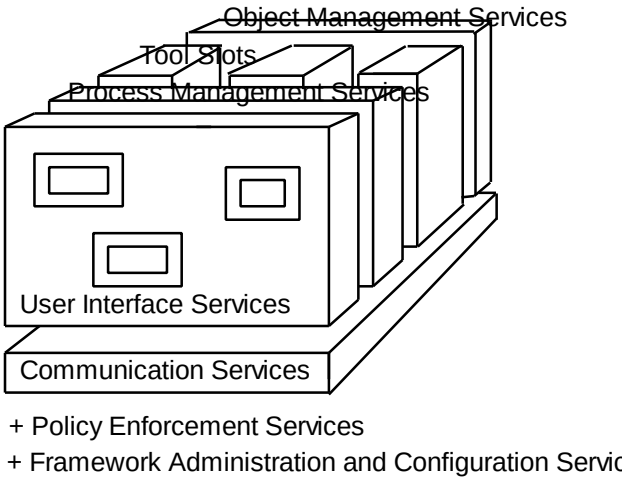

Figure 4.3: ECMA SEE Reference Model

Service	Purpose
Object Management	definition, storage, maintenance, management and access of object entities and the relationships among them
Process Management	definition and performance of software engineering activities across the software life cycle
Communication	provision of a standard inter-tool and inter-service communication mechanism
User Interface	provision of the interface between users and the different components of the environment
[CASE] Tool	support a particular application
Policy Enforcement	enforcement of security and integrity
Administration and Configuration	registration and collection of administrative data

Table 4.3: ECMA Model Services

4.4 CASE TOOL INTEGRATION STANDARDS

Commonly used standards for CASE tool integration are:

- CASE Data Interchange Format (CDIF), a standard for exchanging data between modelling tools;
- Portable Common Tools Environment (PCTE), a standard for integrating tools into an SEE.

4.4.1 CASE Data Interchange Format

The CASE Data Interchange Format (CDIF) is an industry standard for exchanging data between CASE tools [Ref 16]. CDIF permits data integration. The CDIF standard defines the format for:

- data flow diagrams;
- entity relationship diagrams;
- data dictionaries;
- process specifications;
- structure charts;
- state transition diagrams;
- cross reference matrices;
- Ada structure graphs;
- pictures;
- notes.

The objects listed above are produced by structured analysis and design methods. Modelling tools supporting structured analysis and design methods should be able to import and export information in CDIF.

4.4.2 Portable Common Tools Environment

The Portable Common Tools Environment (PCTE) is the European computer manufacturers standard for the integration of CASE tools. PCTE is now gaining wider international recognition.

PCTE is "an interface to a set of facilities that forms the basis for constructing environments". The facilities are designed to provide an infrastructure for tools. PCTE therefore defines the interface between an SEE framework and the CASE tools in the SEE.

Facilities are defined generically in ECMA-149 "PCTE Abstract Specification" [Ref 17] and specifically for the C and Ada programming languages in ECMA-158 and ECMA-162 [Ref 18 and 19]. Key components of PCTE are the:

- Object Base, which is the repository of the data used by the tools;
- Object Management System, which provides the functions used to access the object base.

The ECMA Reference Model (see Section 4.3) defines the requirements for an SEE. The PCTE specifications provide the detailed design.

The benefits of PCTE for software engineers are:

- good control and data integration between tools;
- good presentation integration because of the use of the X standard;
- easy reuse of designs, code and documentation.

Tools that comply with the PCTE standard should be preferred. Although currently few tools and frameworks comply with PCTE standard, the trend seems to be increasing adoption of it.

4.5 THE ESSDE

In 1987, as a result of initiatives by the ESA Technical Directorate and the Columbus and Hermes programmes, and recommendations by the ESA Board for Software Standardisation and Control (BSSC), the concept of a "European Space Software Development Environment" (ESSDE) was formulated. The ESSDE was entrusted to a working group consisting of representatives from the Columbus and Hermes programmes, the European Space Operations Centre, and the ESA Technical Directorate.

The working group produced a URD and an SRD for the ESSDE and investigated areas of the software life cycle which required more detailed definition. A series of reports covering some areas of the software life cycle were produced and a detailed software process model was defined. A detailed evaluation exercise of five systems concluded that current SEE technology, although usable and reliable, was:

- not capable of being easily adapted to provide advanced features;
- difficult to integrate because it did not support standards.

The working group proposed that an ESSDE Reference Facility should be created to provide a baseline for ESA projects. This is composed of two commercial SDE's, ('Concerto" and 'Rational"), supplemented by other commercial tools. Some custom software is included to ensure compatibility with the ESA Software Engineering Standards and to improve integration between the Concerto and Rational platforms.

The ESSDE Reference Facility can be used to:

- demonstrate SEE capabilities;
- demonstrate recommended software methods and procedures;
- produce software according to ESA PSS-05-0 standards;
- provide training;
- evaluate new tools, methods and procedures.

Five important lessons can be learnt from the ESSDE project:

- current SDE technology, although mature, has little growth potential and is difficult and costly to integrate;
- a detailed process model specifying the inputs, outputs, methods and roles associated with every process is essential for the definition of an effective SEE;
- an organisation can benefit from centrally negotiated SEE procurement, installation and maintenance arrangements;
- training in the use of an SEE is essential;
- external support is required for the commercial software.

CHAPTER 5
EVALUATION AND SELECTION OF CASE TOOLS

5.1 INTRODUCTION

A systematic approach to the evaluation and selection of CASE tools should be adopted by projects. Normally the software project manager is responsible for the evaluation and selection of CASE tools. However organisations running multiple projects developing similar software products can achieve economies of scale by evaluating and selecting CASE tools centrally.

The evaluation and selection process is illustrated in Figure 5.1. The methods to be used for software projects, such as object-oriented analysis, are the primary input. These define the capabilities required of the tools. The local environment imposes constraints on aspects such as cost, platform, user interface and ability to integrate with other tools. The evaluation stage combines the capabilities and constraints into a set of "evaluation criteria", and then measures the tools against them. The output of this stage is an evaluation report describing how the CASE tools measure up to the evaluation criteria.

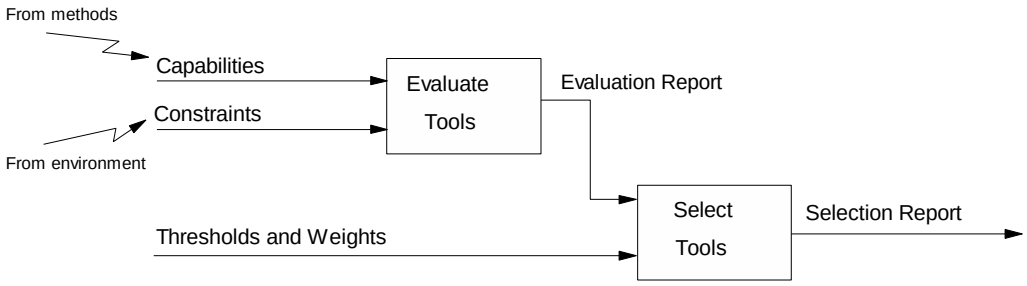

Figure 5.1: Evaluation and Selection Process

In the selection stage, thresholds and weights are applied to the measurements to arrive at a decision as to which CASE tools (if any) will be selected.

An efficient way to evaluate and select tools is often to define first the key criteria and survey a wide range of tools. The tools meeting the key criteria are then shortlisted for a second round of evaluation and selection,

which evaluates the shortlisted tools against all the criteria. The second stage of evaluation may include demonstrations and tests.

Sections The guidelines should be tailored according to cost and complexity of the tools required.

5.2 and 5.3 discuss evaluation and selection in more detail. Much of the discussion is based upon the IEEE Recommended Practice for the Evaluation and Selection of CASE tools [Ref 10]. The IEEE Standard conceives of both a standalone evaluation process and a standalone selection process, as well as a compound process. The description of evaluation and selection below reflects this. The guidelines should be tailored according to cost and complexity of the tools required.

5.2 EVALUATION OF CASE TOOLS

Evaluation is a measurement process. The steps of the process are:

- plan the evaluation;
- define the evaluation criteria;
- identify candidate tools;
- collect data about CASE tool products;
- compare tool data with the evaluation criteria;
- produce the evaluation results.

The plan for the evaluation should define the purpose and scope of the evaluation, identify assumptions and constraints, and list evaluation activities. Thorough evaluation can require a large amount of effort and careful planning and control is needed.

The process model defines the process to be supported by the CASE tools. Evaluation criteria can be formulated in the same style as user requirements [Ref 3]. The criteria should be based upon:

- capabilities required by the process model (e.g. activity and method support);
- additional capabilities not directly implied by the process model (e.g. performance);
- constraints forced by the local environment (e.g. platform, cost);
- additional constraints (e.g. user interface, documentation).

Examples of evaluation criteria are:

- supports activity "A" (e.g. regression testing);
- supports method "B" (e.g. Ward-Mellor structured analysis and design);
- performs all updates of a model within "C" seconds (performance capability);
- runs on platform "D" (platform constraint);
- provides online help (documentation constraint);
- technical support is available (constraint);
- training courses are available (constraint);
- has at least "E" existing users (constraint).

A checklist should be made from the evaluation criteria.

The next steps are to identify tools that might support the requirements and to collect data about them. This information should be stored in a database. Data collection may involve:

- examining CASE tool literature;
- interviewing users of the tools;
- viewing demonstrations of the tools;
- testing the tools.

The tool data is then used to complete an evaluation checklist for each tool. The degree of conformance of a tool to each criterion should be measured numerically. A matrix may be assembled from the checklists to ease comparison. The matrix should have one row for each criterion and one column for each tool.

The next stage is to evaluate how the tools will work together as a system. This stage uses the raw data from the preceding comparison stage and the software engineering knowledge of the evaluators. Several possible systems may be assembled from the tools. Some points to consider are tool-to-tool:

- compatibility (e.g. can exchange and use each other's data);
- consistency (e.g. common user interface).

Individual tools may comply with the requirements but not be compatible or consistent with any of the other tools in the SEE. Any

inconsistencies and incompatibilities should be highlighted in the final stage of evaluation. The evaluators should contribute their own knowledge and experience of the CASE tools to draw attention to their strengths and weaknesses.

The last stage of the evaluation is to cost the possible toolsets.

The output of the evaluation is a report that should contain:

- a summary of the evaluation activities;
- summary tool data (name, version, vendor, functions, platforms, cost elements, standards supported);
- completed checklists or compliance matrices;
- summary and analysis of each system option;
- costs of each system option.

5.3 SELECTION OF CASE TOOLS

The selection process considers the evaluation report and defines which CASE tools will be used. The separate measurements made in the evaluation stage are filtered, weighted and combined to produce recommendations.

The steps of the selection process are:

- plan the selection;
- identify selection criteria;
- apply selection algorithm;
- assess selection;
- make recommendations.

The selection process for a small project may be performed entirely by the project manager. The selection process for the software engineering environment of a large project may require formal reviews involving several people. This may require planning.

The selection criteria come from:

- evaluation criteria;
- analysis of system options.

Evaluation criteria may be ignored in the selection process because they are not important (e.g. colour displays are not necessary) or because they do not discriminate between tools (e.g. as when all the tools under consideration run on the same platform).

The analysis of system options may result in additional selection criteria (e.g. compatibility).

The selection algorithm is then applied. This may use one or more of the following selection algorithms [Ref 10]:

- scale-based algorithm;
- rank-based algorithm;
- cost-based algorithm.

A scale-based algorithm calculates a single value for each CASE tool by multiplying the weight given each criterion by its score (on a scale) and adding all such products.

A rank-based algorithm orders the CASE tools according to their scores and selects them according to their place in the order.

A cost-based algorithm identifies the minimum level of capability acceptable and then ranks tools above that level in order of cost.

The results of the selection algorithm are then assessed. Scores may be recalculated using different selection criteria and different weights to check the sensitivity of the result to a particular criterion.

Final selection of all the tools early in the project may not be possible, or even desirable. However the need to integrate tools and make a cost estimate for the project forces the options to be evaluated and a provisional selection to be made at the start. Ideally, the provisional selection should be reviewed and a final selection made just before the start of the phase in which the tool is first needed.

The option of not having tool support for a task should always be considered. The cost of a tool may outweigh its benefits.

The output of the selection process is a report that:

- defines the selection criteria;
- explains the selection algorithm;
- assesses the selection;
- recommends what CASE tools, if any, should be selected.

APPENDIX A
GLOSSARY

A.1 LIST OF TERMS

The usage of all terms in this document is consistent with the other ESA PSS-05 documents [Ref 1 to 9] and ANSI/IEEE Std 610.12-1990 [Ref 14]. Definitions of terms used in this document and not listed here may be found in those references. Cross references between terms in the list are identified by *italics*.

build tool

A software tool that compiles and links a software system.

code generator

A software tool that accepts as input the requirements or design for a computer program and produces source code that implements the requirements or design [Ref 14].

comparator

A software tool that compares two computer programs, files or sets of data to identify commonalities or differences [Ref 14].

compiler

A computer program that translates programs expressed in a high order language into their machine language equivalents [Ref 14].

computer aided software engineering tool

A software tool that aids in software engineering activities, including but not limited to requirements analysis and tracing, software design, code production, testing, document generation, quality assurance, configuration management and project management [Ref 9].

coverage analyser

A software tool that supports the tracking of the path taken by the control flow during execution with the object of determining what parts of the code have been executed.

cross-compiler

A compiler that executes on one computer but generates machine code for a different computer [Ref 14].

debugger

A software tool to control and monitor the execution of a program with the objective of detecting and locating faults in the program.

documentation generator

A software tool that supports the generation of documentation from models and source code.

document management tool

A software tool that supports the storage and change control of documents, and, optionally, handles document images as well as text and graphics.

dynamic analyser

A software tool that supports the process of evaluating a system or component based upon its behaviour during execution [Ref 14]. A combination of a *coverage analyser* and a *performance analyser*.

installation tool

A software tool that copies software from its distribution medium into the target environment and configures the target environment for the new software.

integrated project support environment

The support environment for projects developing systems of both software and hardware. An IPSE for a software-only project is the same as a *Software Engineering Environment*.

language sensitive editor

A software tool that allows the entry, alteration and viewing of source code, checks the syntax of the source code, and provides template source code statements for the programmer to complete.

librarian

A software tool that supports the creation and maintenance of libraries of software modules.

linker

A computer program that creates a single load module from two or more independently translated object modules or load modules ... [Ref 14].

modelling tool

A software tool that supports the construction of a simplified description of one or more of the functions, entities, data, states, components, behaviour, control flow and data flow in the system.

method

The rules and procedures associated with an activity.

performance analyser

A software tool that supports the measurement of the computer resources used by each part of the code during execution. Same as a *profiler.*

problem handling tool

A software tool that supports the online entry, storage and retrieval of problem reports, change requests and modification reports

procedure

A course of action to be taken to perform a given task [Ref 14].

process

A sequence of steps performed for a given purpose [Ref 14].

process model

The procedures, inputs, outputs, roles and responsibilities, associated with each process in a project.

process modelling tool

A software tool that supports the definition of procedures, provides a library of templates for process design, and produces the framework for defining the work packages and workflow in the project.

process support tool

A software tool that helps guide, control and automate project activities.

profiler

See *performance analyser*.

program proving tool

A software tool that formally proves that code is logically correct.

project planning tool

A software tool that supports the definition of activities, resource requirements, activity dependencies, activity start and end times, and uses this information to construct activity networks and schedules.

prototyping tool

A software tool that supports the rapid development of software used for exploring requirements or experimenting about technical feasibility.

repository

A software tool that stores every version of every configuration item in the system, maintains a change history of each configuration item, and maintains a database of the relationships between configuration items.

requirements management tool

A software tool that supports the entry, inspection, update and deletion of requirements; attachment of attributes to requirements; control of access to requirements; tracking of changes to requirements; search and retrieval on requirement text or attribute value; definition of dependencies for the purposes of structuring and tracing.

reverse engineering tool

A software tool that accepts source code as input and outputs restructured source code, or reformatted source code, or software design information.

risk analysis tool

A software tool that supports the identification and evaluation of the factors that threaten the success of a project.

semantic analyser

A software tool that substitutes algebraic symbols into the program variables and presents the output of the program as algebraic formulae.

software cost estimation tool

A software tool that supports the prediction of labour costs and project duration.

software development environment

Same as *software engineering environment*

software engineering environment

A system that provides automated support of the engineering of the software systems and the management of the software process [Ref 10]. An SEE should support most, if not all, life cycle activities.

static analyser

A software tool that supports the process of evaluating a system or component based upon its form, structure, content or documentation [Ref 14].

test case generator

A software tool that accepts as input source code, test criteria, specifications, or data structure definitions, uses these inputs to generate test input data, and, sometimes, determines expected results [Ref 14].

test harness

A software module used to invoke a module under test and, often, provide test inputs, control and monitor execution, and report test results [Ref 14].

test management tool

A software tool that supports the storage and change control of test data and scripts, the setup and execution of tests, and the storage of outputs.

tracing tool

A software tool that establishes a relationship between two or more products of the development process [Ref 14].

workbench

An integrated set of tools that supports some closely related activities. For example a "programming workbench" might contain a *language sensitive editor*, a *compiler* and a *debugger*.

A.2 LIST OF ACRONYMS

AD	Architectural Design
ADD	Architectural Design Document
ANSI	American National Standards Institute
ASCII	American Standard Code for Information Interchange
BSSC	Board for Software Standardisation and Control
CASE	Computer Aided Software Engineering
CDIF	CASE Data Interchange Format
CGM	Computer Graphics Metafile
DD	Detailed Design
DDD	Detailed Design Document
ECMA	European Computer Manufacturers Association
HOOD	Hierarchical Object-Oriented Design
IPSE	Integrated Project Support Environment
LOTOS	Language Of Temporal Ordering Specification
OM	Operations and Maintenance
PCTE	Portable Common Tools Environment
PDL	Program Design Language
SDE	Software Development Environment
SEE	Software Engineering Environment
SADT	Structured Analysis and Design Technique
SCM	Software Configuration Management
SCMP	Software Configuration Management Plan
SPM	Software Project Management
SPMP	Software Project Management Plan
SQA	Software Quality Assurance
SQAP	Software Quality Assurance Plan
SR	Software Requirements
SRD	Software Requirements Document
SSADM	Structured Systems Analysis and Design Methodology
SUM	Software User Manual
SVV	Software Verification and Validation
SVVP	Software Verification and Validation Plan
TR	Transfer
UR	User Requirements
URD	User Requirements Document
VDM	Vienna Development Method

This page is intentionally left blank

APPENDIX B
REFERENCES

1. ESA Software Engineering Standards, ESA PSS-05-0 Issue 2 February 1991.

2. Guide to the ESA Software Engineering Standards, ESA PSS-05-01 Issue 1 October 1991.

3. Guide to the User Requirements Definition Phase, ESA PSS-05-02 Issue 1 October 1991.

4. Guide to the Software Requirements Definition Phase, ESA PSS-05-03 Issue 1 October 1991.

5. Guide to the Software Architectural Design Phase, ESA PSS-05-04 Issue 1 January 1992.

6. Guide to the Software Detailed Design and Production Phase, ESA PSS-05-05 Issue 1 May 1992.

7. Guide to Software Configuration Management, ESA PSS-05-09 Issue 1 November 1992.

8. Guide to Software Verification and Validation, ESA PSS-05-10 Issue 1 February 1994.

9. Guide to Software Quality Assurance, ESA PSS-05-11 Issue 1 July 1993.

10. Recommended Practice for the Evaluation and Selection of CASE Tools, IEEE Std 1209-1992

11. Reference Model for Frameworks of Software Engineering Environments, ECMA TR/55 NIST Special Publication 500-201, December 1991

12. IEEE Standard for Software Project Management Plans, ANSI/IEEE Std 1058.1-1987

13. IEEE Standard for Software Quality Assurance Plans, ANSI/IEEE Std 730-1989.

14. IEEE Standard Glossary of Software Engineering Terminology, ANSI/IEEE Std 610.12-1990

15. Real-Time Case: the Integration Battle, C. Chappell, V. Downes and C. Tully, Ovum Ltd, 1989.

16. CASE Data Interchange Format (CDIF) Syntax Definition, Cadre Technologies Inc, December 1989.

17. Portable Common Tool Environment (PCTE) Abstract Specification, Standard ECMA-149, 2nd Edition, June 1993.

18. Portable Common Tool Environment (PCTE), C Programming Language Binding, ECMA-158, June 1991.

19. Portable Common Tool Environment (PCTE), Ada Programming Language Binding, ECMA-162, December 1991.

20. Trial Use Standard Reference Model for Computing Tool Interconnections, IEEE Std 1175-1992

21. Structured Analysis and System Specification, T. DeMarco, Yourdon Press, 1980.

22. SSADM Version 4, NCC Blackwell Publications, 1991.

23. Structured Development for Real-Time Systems, P.T.Ward and S.J.Mellor, Yourdon Press, 1985.

24. System Development, M.Jackson, Prentice-Hall, 1983.

25. Structured Analysis (SA): A Language for Communicating Ideas, D.T.Ross, IEEE Transactions on Software Engineering, Vol SE-3, No 1, January 1977,

26. Object-Oriented Analysis, P.Coad and E.Yourdon, Second Edition, Yourdon Press, 1991.

27. Object-Oriented Modelling and Design, J.Rumbaugh, M.Blaha, W. Premerlani, F. Eddy and W.Lorensen, Prentice-Hall, 1991.

28. Object-Oriented Design with Applications, G. Booch, Cummings, 1991.

29. Object-Oriented Systems Analysis - Modelling the World in Data, S.Shlaer and S.J.Mellor, Yourdon Press, 1988.

30. Object Lifecycles - Modelling the World in States, S.Shlaer and S.J.Mellor, Yourdon Press, 1992

31. HOOD Reference Manual, Issue 3, Draft C, Reference WME/89-173/JB HOOD Working Group, ESTEC, 1989

32. Structured Software Development Using VDM, C.B.Jones, Prentice-Hall 1986.

33. Structured Design: Fundamentals of a Discipline of Computer Program Specification and Systems Design, E.Yourdon and L.Constantine, Yourdon Press, 1978.

34. Object-Oriented Design, P.Coad and E.Yourdon, Prentice-Hall, 1991.

35. Structured Testing: A Software Testing Methodology Using the Cyclomatic Complexity Metric, T.J.McCabe, National Bureau of Standards Special Publication 500-99, 1982.

36. Design Complexity Measurement and Testing, T.J.McCabe and C.W.Butler, Communications of the ACM, Vol 32, No 12, December 1989.

APPENDIX C
INDEX