CHAPTER 2

SOFTWARE ENGINEERING TOOLS AND TECHNIQUES

2.1 Definition

2.1.1 Software

Software is defined as intellectual creation comprising the program, procedures, rules and any associated documentation pertaining to the operation of data processing system [ISO96]. And, software product is defined as a set of computer programs, procedures and associated documentation and data designated for delivery to a user. Formally software is defined as information that has the following three distinguishing characteristic:

1. Structured with logical and physical properties
2. Created and maintained in various forms and representations during the software system development life cycle
3. Tailored for machine processing in its fully developed state”.

Software is also the specification documentation that leads to computer codes. As a result, the definition of software encompasses both specification documentation and computer codes as shown in Figure 2.1 [SCO97].

Figure 2.1 Software Definition

2.1.2 Software Engineering

Among the many definitions of software engineering proposed since 1970, the most accurate and descriptive one was created by F.L Bauer of the Technical University, Munich, Germany, in 1972. His definition is the established and use of sound engineering principles (methods) in order to obtain economically software that is reliable and work on real machines [NIT93].
The development of consistently high quality software and service support requires that the software process used to develop these products and services be based on sound software engineering principles incorporating current best software engineering practices. We have subdivided software engineering principles into three categories: Quality, Management and Engineering, as shown in Figure 2.2 [JOC94].

![Venn Diagram](image)


Figure 2.2 Software Engineering Principles

### 2.1.3 Software Quality

On the previous section, software is defined not only as computer program codes, but also its related specifications and documentation. The term of software quality has been interpreted in many different ways [DAV90].

Mc Call provides the definition of software quality most widely used within the software engineering world [CAL78].
Cavano and Mc Call, go beyond the notion of "absence of defects" to spell out all relevant factors of software quality in three dimensions of software products (1) product operation, (2) product revision, and (3) product transition. Figure 2.3 shows the three dimensions of software quality [CAL78].

![Diagram of software quality dimensions]

- Maintainability (Can I fix it ?)
- Flexibility (Can I change it ?)
- Testability (Can I test it ?)

Portability (Can I use it on another machine ?)
Reusability (Can I reuse some parts of software ?)
Interoperability (Can I couple it with another system ?)

- Correctness (Does it do what I want ?)
- Reliability (Does it do accurately most of the time ?)
- Efficiency (Will it run well on my hardware)
- Integrity (Is it secure ?)
- Usability (Can I run it ?)


Figure 2.3 Software Quality Attributes

The first dimension, product operation, deals with quality attributes such as correctness, reliability, efficiency, integrity and usability.
The second dimension, product revision, deals with quality attributes, such as maintainability, testability and flexibility. The third dimension product transition, deals with quality attributes such as portability, reusability and interoperability. The meaning of each of these quality attributes is explained in Table 2.1. Mc Call breaks each of these factors down into criteria, each of which has specific metrics associated with it [JCA77].

Table 2.1. Software Quality Attributes

<table>
<thead>
<tr>
<th>Dimension</th>
<th>Quality Attribute</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Product Operation</td>
<td>Correctness</td>
<td>Extent to which the software conforms to its specifications and standards.</td>
</tr>
<tr>
<td></td>
<td>Reliability</td>
<td>Extent to which the software will perform with any failures within a specified time period.</td>
</tr>
<tr>
<td></td>
<td>Efficiency</td>
<td>Relative extent to which a resources is utilized ( eg. Storage space, processing time).</td>
</tr>
<tr>
<td></td>
<td>Integrity</td>
<td>Extent to which software will perform without failures due to unauthorized access to the software or data within a specified time period.</td>
</tr>
<tr>
<td></td>
<td>Usability</td>
<td>Relative effort for training or software operation (eg. Familiarization, input preparation, execution, output interpretation).</td>
</tr>
<tr>
<td>Product Revision</td>
<td>Maintainability</td>
<td>Ease of effort for locating and fixing a software failure within a specified time period.</td>
</tr>
<tr>
<td></td>
<td>Testability</td>
<td>Relative effort to test within the specified time period.</td>
</tr>
<tr>
<td></td>
<td>Flexibility</td>
<td>Ease of effort for changing the software missions, functions, or data to satisfy other requirements.</td>
</tr>
<tr>
<td>Product Transition</td>
<td>Portability</td>
<td>Relative effort to transport the software for use in another environment (hardware configuration and / or software system environment).</td>
</tr>
<tr>
<td></td>
<td>Reusability</td>
<td>Relative effort to convert a software component for use in another application.</td>
</tr>
<tr>
<td></td>
<td>Interoperability</td>
<td>Relative effort to couple the software of one system to the software of another system.</td>
</tr>
</tbody>
</table>

2.2 Software Development Life Cycle

2.2.1 Waterfall Model

The waterfall model, depicted by Figure 2.4. The waterfall model takes its name from figures used to depict the process. This figure shows the process as a chronological sequence of development even flowing from left to right and descending.

Figure 2.4 Software Development Life Cycle

This model classified its life cycle into five phases namely,

1. Requirements Analysis And Definition,

2. System And Software Design,
3. Implementation And Unit Testing,
4. Integration And System Testing,
5. Operation And Maintenance.

The following sections briefly explain the activities involved as well as the accompanying and techniques for five different phases of the software development life cycle. Different activities and technical work can be identified with the five phases [IAN98]:

1. Requirement Analysis And Definition

The software analysis phase is a stage where problems are defined and analyzed. Definition of preliminary goals and requirements for the software project are the purpose of this analysis. These activities are also term software requirement analysis and specifications. The document produced at this phase is called the Software Requirement Specifications (SRS). Other requirement specifications made at this phase include details of the qualities of the system to be developed.

At this phase a top level project plan with milestone, resources, schedules and budget for the software development process is determined so that the system developed is cost effective.
2. System and Software Design

Software design is a crucial process since it translates the requirements into a detailed representation of a software system. The activities involved consist of the architectural and detailed design stages. During the architectural design stage, requirement specifications are decomposed into a system structure which consist of functionally related grouping is known as modules. On the other hand, the detail design is concerned with the transformation of the system structure into procedural descriptions also known as algorithms of a software system which can be given to a programmer.

The document produced at this phase is called the Software Design Specifications (SDS).

3. Coding and Unit Testing

The coding phase is concerned with translating the Software Design Specification (SDS) into programming language. During this stage, the software design is realized as a set of programs or program units. This activity is performed by programmers who define the concrete data structures and write codes for algorithms defined in the detailed design stage. Unit testing involves verifying that each unit meets its specifications. As such the primary goal of software coding is production of simple, well understood and internally source codes documented.
4. Integration and System Testing

The individual program units or programs are integrated and tested as a complete system to ensure that the software requirements have been met. After testing, the software system is delivered to the customer.

5. Operation and Maintenance

Normally this is the longest life cycle phase. The system is installed and put into practical use. Maintenance involved correcting errors which were not discovered in earlier stages of life cycle, improving the implementation of system units and enhancing the system's services as new requirements are discovered.

In practices, these stages overlap and feed information to each other. During design, problems with requirements are identified; during coding, design problems are found and so on.

2.2.2 Boeing's Spiral Model

As the rapid application development strategies become more popular, the spiral life cycle, proposed by Boehm, is becoming more and more applicable. Boehm's model takes the form of spiral (Figure 2.5) [BAR89]. Each loop in the spiral represents a phase of software process. The spiral model for SDLC has been developed in response to the criticism that traditional SDLC models discourage or at least do not encourage the use of prototyping and software reuse.
The spiral model is a risk driven approach to software development that encompasses the best features of both classic life cycles and prototyping. The most important distinction between the spiral model and software process models is the explicit consideration of risk in the spiral model.

Cumulative Cost

Determine objectives alternatives constraints

Evaluate alternatives identify, resolve risks

Risk analysis

Risk analysis

Risk analysis

Review

Requirements plan Life cycle

Concept of Operation

S/W Requirements

Integration and test plan

Requirement Validation

Design V & V

Integration Test

Acceptance Test

Development plan

Prototype 1

Prototype 2

Prototype 3

Operational prototype

Simulation, models, benchmarks

Product design

Detail design

Code

Unit test

Acceptance

Develop, verify next level product.

Plan next phase

Figure 2.5 Boehm's Spiral Model of Software Process
Some of the advantages of spiral model are [BAR89]:

1. It could be used as effectively for system enhancement as for system development.

2. Most life cycle models can be considered as special case of spiral model.

3. The embedded risk analysis built into the model avoids many of difficulties that arise in other models.

Each loop in the spiral is split into four sectors [BAR89]:

1. Objective setting

Specific objectives for the phase of project are defined. Constraints on the process and the product are identified and a detailed management plan is drawn up. Project risks are identified.

2. Risk assessment and reduction

For each of the identified project risks, a detail analysis is carried out. There are some steps are taken to reduce the risk. For example, if there is a risk that the requirements are inappropriate, a prototype system may be developed.

3. Development and validation

After having risk evaluation, a development model for the system is then chosen. For example, if user interfaces risks are dominant, an appropriate development model might be evolutionary prototyping. If safety risks are main consideration, development based on formal transformations may
be not the most appropriate development model if the main identified risk is sub system integration.

4. Planning

The project is reviewed and a decision made whether to continue with a further loop of the spiral. If it is decided to continue, plans are drawn up for the next phase of the project.

2.3 Software Tools

2.3.1 Definition of tools

Generally, a tool is defined as any device or instrument used to carry out mechanical function whether manually or by a machine [JUD95]. In the field of computing, tool is refers to an instrument or automatic system for accomplishing something in a better way [SHA98].

This "better way" can mean that the tool makes us more accurate, more efficient, or more productive or that enhances the quality of the resulting product.

As these tools are devised for helping programmers in developing other programs, they are also known as programmer's support tools.

2.3.2 Classification of software tools

As computing is a broad field, there is a wide range of tools in this discipline. This classification of tools focuses mainly on those that are necessarily and extensively required by users and programmers.
Six categories based on their applications are identified [IAN98]:

1. **Software Tools** for the automation of software development, maintenance and analysis.

2. **Authoring Software Tools** for the development of computer Aided Instruction (CAI) courseware.

3. **Word Processing Tools** for program development, text processing and documentation purposes.


6. **Software Design Tools** for system designing, software designing and programming applications.

### 2.3.3 CASE technology and CASE Tools

Automated tool support for software engineers should therefore lead to improvements in software productivity. Since the early 1980s, a wide range of tools to support software development have been developed. The term computer aided software engineering (CASE) is now generally accepted as the name for this automated support for the software engineering process. CASE is the
automation of automation. It offers a very practical solution to software productivity and quality program. Many software developers are looking to CASE tools to make dramatic improvements in productivity and reliability in the next future [JOC94].

CASE technology can be applied in a cost effective way to reduce software costs and development time significantly and to increase software quality. The available of CASE support makes it possible to re-engineer some software processes so that they are more effective. Thus, CASE has contributed and will continue to contribute to the development of software engineering.

The area of CASE has developed very rapidly and, as a consequence, different terminology is used by different people to describe CASE systems. Terms such as tools, toolkits, workbenches and environments are used inconsistently.

Table 2.2 lists a number of different types of CASE tool and gives specific examples of each tool [IAN98].
Table 2.2 Functional classification of CASE tools

<table>
<thead>
<tr>
<th>Tool type</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Management tools</td>
<td>PERT tools, estimation tool.</td>
</tr>
<tr>
<td>Editing tools</td>
<td>Text editors, diagram editors, word processors.</td>
</tr>
<tr>
<td>Configuration management tools</td>
<td>Version management system, change management systems.</td>
</tr>
<tr>
<td>Prototyping tools</td>
<td>Very high languages, user interface generators.</td>
</tr>
<tr>
<td>Method support tools</td>
<td>Design editors, data dictionaries, code generators.</td>
</tr>
<tr>
<td>Language processing tools</td>
<td>Compiler, interpreters.</td>
</tr>
<tr>
<td>Program analysis tools</td>
<td>Cross reference generators, static analysers, dynamic analysers.</td>
</tr>
<tr>
<td>Testing tools</td>
<td>Test data generators, file comparator.</td>
</tr>
<tr>
<td>Debugging tools</td>
<td>Interactive debugging systems.</td>
</tr>
<tr>
<td>Documentation tools</td>
<td>Page layout programs, image editors.</td>
</tr>
<tr>
<td>Re engineering tools</td>
<td>Cross reference system, program restructuring systems.</td>
</tr>
</tbody>
</table>

Quality of CASE support for software process activities are illustrated in Figure 2.6 [IAN98].
Figure 2.6. Quality of CASE support for software process activities

The breadth of support for the software process offered by CASE technology is another possible classification dimension. Fuggetta (1993) proposes that CASE system should be classified in three categories [FUG93]:

1. **Tools** support individual process tasks such as checking the consistency of a design, compiling a program, comparing test result and so on. Tools may be general purpose, stand-alone tools (for example, a word processor) or may be group into workbenches.

2. **Workbenches** support process phases or activities such as specification, design and so on. They normally consist of a set of tools with some greater or lesser degree of integration.

3. **Environments** support all or at least a substantial part of the software process. They will normally include several different workbenches which are integrated in some ways.
Figure 2.7 illustrates this classification and shows some examples of these different classes of CASE support.

Figure 2.7 Tools, Workbenches and Environments

In practice, the boundaries between these different classes are not clear-cut. Tools may be sold as a single product but embed support for different activities. For example, many word processors now provide a built-in diagram editor.
Individual CASE tools are useful and cost effective. However, more leverage is obtained when CASE tools work together in an integrated way. With an integrated system, training cost are potentially reduced as existing software is reused when new system are added. Examples of situations where benefits are gained from integration include:

i. The integration of a design workbench with a documentation workbench. The documentation automatically generated by the design tools can be formatted neatly and included in system documentation produced using the documentation workbench.

ii. The integration of specification, design and programming tools with a Configuration Management (CM) workbench. The outputs from the tools can be managed using the CM system. The organization can keep track of different changes, versions, releases and so on.

The introduction and the use of CASE technology requires careful planning. CASE tools are expensive and they must be maintained until all of the software produced using these tools become obsolete.
CASE systems in an organization follow a life cycle from initial procurement through to ultimate obsolescence. Figure 2.8 illustrates the stages of this CASE System Life Cycle [IAN98]. These stages are:

1. Procurement
   This involves choosing an appropriate CASE system for the type of software being developed.

2. Tailoring
   This phase adapts a CASE system to a particular set of organizational or project requirements.

3. Introduction
   This phase introduces the CASE system into a working context. During this phase, engineer are trained in the use of the system.

4. Operation
   This is the phase where the CASE system is in everyday use for software development.

5. Evolution
   Evolution is not really a separate phase but is a continuing activity throughout the life of the CASE system. It involves modifying the hardware or software to adapt the system to new requirements.
6. Obsolescence

During this phase the CASE system is taken out of use. It must be ensured that software developed using the system can still be supported by the organization.

Figure 2.8 The CASE System Life Cycle