FILENAME.APP = 6911MS00.tex

Computer Aided Software Engineering

Ladan Tahvildari, Kostas Kontogiannis

Dept. of Elect. and Comp. Eng.

University of Waterloo

Waterloo, Ontario

Canada, N2L 3G1

{ltahvild,kostas}@swen.uwaterloo.ca

Tel : (519)-888-4567-x3819

Fax : (519)-746-3077

Introduction

It is more and more recognized by researchers and software practitioners alike that the process of developing large software applications is so complex that it requires tight process control. The process of establishing requirements and translating them into designs, computer code, and operational procedures is in many respects error prone, slow and laborious. Incomplete, ambiguous, and conflicting requirements are some of the reasons why software development is such a difficult task. These problems can be traced to many factors. First, clients may not understand fully the functional and non-functional requirements for the system they would like to develop until they actually see a prototype. Second, business executives and management are not familiar with the technical issues related to software development, and how they can best apply technology to address specific business needs. On the antipode, computer professionals are not likely to fully understand all aspects of business needs. Under these conditions, there is a high likelihood that system requirements and designs will have to be changed several times during the life cycle of the project in order to accommodate end-users, business needs, and at the same time to be technically feasible.

The end result is that these laborious and error-prone aspects of software development make software brittle and difficult to maintain. It is of no surprise that more and more time and money are spent on maintaining previously developed software. This is referred to as the "*software maintenance crisis*". A possible solution to this problem is to utilize highly flexible computer-assisted development environments that allow software engineers and managers to better perform specific design, test, and maintenance tasks. Such environments are referred to as Computer Aided Software Engineering (CASE) environments.

CASE by definition is a technology by itself and aims on providing automated support to software specifications, design, development, testing, maintenance, and project management for the information technology sectors [9]. The ability to even partially automate the engineering of large software systems allows for greater flexibility for the software development process in a way that may better accommodate business requirements. It has been argued that CASE technology can substantially reduce the design and development complexity inherent in medium and large software projects. This can be achieved by automating creation of many software artifacts (i.e., designs, specifications, test cases) specified by the software architects and software engineers.

This article is intended to provide an overview of the Computer Aided Software Engineering technology and discusses the basic requirements for CASE environments and the fundamental issues these environments attempt to address. Several illustrations and examples are included in order to explain the related issues and concepts. Readers interested in pursuing topics in more depth are invited to review the bibliography at the end of this article.

The Software Development Process

There are many steps and activities involved in specifying, designing, building, deploying, and maintaining a software product. These steps are referred to as the "software life cycle" as illustrated in Figure 1. The order in which these steps and activities are performed defines a specific *process model* or *software production process* [1,12,22]. In the relevant software engineering literature, several such process models have been proposed. These include the *Rapid Prototyping Model*, the *Waterfall Model*, and the *Spiral Model* [1,12,22].

Production and manufacturing processes models have been investigated in any discipline whose goal is to produce high quality products. The goal of a process model is to make production reliable, predictable, repeatable, and efficient. A well-defined production process model, as used, for example, in automobile production, has many benefits including support for automation and standardization. By defining a model for the software production process, we can achieve simpler benefits. However, one must also keep in mind the distinguishing characteristics pertaining specifically to software, that is, software is characterized by complex behavior, and lacks any measurable physical characteristics to determine its quality. Moreover, requirements change constantly and as a consequence, the end-products themselves are often become an amalgamation of conflicting requirements.

Overall, we can identify three major phases for software development. The three phases, illustrated in Figure 1, *definition*, *development*, and *maintenance*, are encountered in all software development, regardless of application area, project size, or complexity. The sections below discuss these phases in the light of CASE technology and further elaborate on the issues involved.

The Definition Phase

The definition phase focuses on *what* the software system is supposed to do, and not on how its functionality can be achieved. That is, during the definition phase, the software architect identifies what information needs to be processed, what functions and performance enhancements are required, what interfaces must be established, what design constraints exist, and what validation criteria are needed to measure whether the delivered system would meet its requirements and design objectives. In this phase, the key requirements of the system and the software must be clearly identified. Although the specific techniques applied during the definition phase will vary depending upon the software engineering paradigm (or the combination of paradigms) applied, two specific activities will most probably occur in this phase. These are :

- *Feasibility Study :* The purpose of this activity is to produce a *feasibility study document* that evaluates the costs and benefits of the proposed application. To do so, it is first necessary to analyze the problem, at least at a global level. Obviously, the more the problem is understood, the better one can identify alternative solutions, their costs, and their potential benefits to the user. Therefore, software engineers must perform a thorough analysis of the problem domain and a well-founded feasibility study. Unfortunately, this is often too ideal in industrial settings because of time, budget, and personnel constraints. In a nutshell, the feasibility study aims on presenting definitions as well as, time and cost ground rules and assumptions for the development of the specific application. The result of this study is a document that should contain at least : (1) the definition of the problem, (2) alternative solutions and their expected benefits, (3) ground rules, assumptions, and rough estimates of required resources for each proposed alternative solution (i.e. develop from scratch, out-source, by off-the-shelves, maintain existing status).

- *Requirements Analysis and Specification :* The purpose of a requirements analysis is to identify the functional and non-functional characteristics of the application, in terms of functionality, performance, ease of use, portability, and so on. In this step, the focus is on *what* qualities the application must exhibit, and not *how* such qualities can be achieved by design and implementation. The reason is that requirements should not unduly constrain the software engineer in the consequent design and implementation activities. The first deliverable of this phase is a *requirement specification document*, that presents the functional and non-functional characteristics of the system to be developed. The purpose of this document is twofold. First, it provides means for the customer to verify whether the intended system conforms with all of the customer's expectations. On the other hand, it is used by the software engineers to develop a design and a consequent implementation of the system in a way that it meets the agreed requirements. Second, the requirements analysis phase is intended to produce a definition of the *system test plan.* In fact, during system testing the system is expected to be tested against its requirements. The way testing will eventually be done may be agreed upon with the customer at this stage, and documented along with the requirements specification document.

The Development Phase

The development phase focuses on *how* the software system delivers its functionality and how it can conform with its functional and non-functional requirements. That is, during development the software engineers architect the system, design the data structures to be used, specify procedural details to be implemented into source code, and define how testing will be performed. The methods applied during the development phase may vary depending upon the software engineering paradigm (or combination of paradigms) that are deemed appropriate to apply for each specific project. However, three specific steps will always occur as follows:

- *Design and Specification:* Design involves decomposing the system into modules that each one is intended to deliver specific functionality. The result is a *design specification document*, which contains a description of the software architecture that is, what each module is intended to do and what are the data and control dependencies among modules. The specification process may proceed iteratively in a bottom-up or top-down way and can be described through different levels of abstraction or different architectural views. Each module identified at some step may be decomposed into submodules. Some models place a limit into the number of levels of design that can be specified. However, it is customary to distinguish between *preliminary* (or *high-level*) design and *detailed design*, but the meaning of these terms varies considerably from case to case. Preliminary design is intended to describe the modular structure in terms of relations (such as *USES*, *IS_COMPOSED_OF*, and *INHERITS_FROM*), whereas detailed design deals with specifying module interfaces (both syntactically and semantically) [12]. Others use the terms to imply a distinction between a logical decomposition (high-level design) and a physical decomposition of the program into programming language units [22]. Still, others refer to module decomposition as preliminary design, and refer to detailed design as defining the main data structures and algorithms for each module [1]. The exact format of the design specification document is usually defined by company-wide, or process specific standards [20]. These standards may also propose specific design methods and practices, along with notations that should be used to document the design of the system being built.

- *Coding and Module Testing:* Coding and module testing is the activity in which software engineers actually implement and test their designs using a programming language. Until recently, it was the only recognized step in the development phase. However, it is merely just one out of several steps as these are specified in various process models such as the Waterfall Process [22] illustrated in Figure 2. The output of this phase is a collection of source files and compilation units. In addition to the guidelines provided by the selected process model for this phase, coding can be also subject to company-wide standards, which may define the suggested layout of programs, such as the headers for comments in every unit, naming conventions for variables and subprograms, the maximum number of lines of executable source code in each unit, and other aspects that the company deems worthy of standardization. Module testing is also a task that encompasses precise definitions of a test plan, the definition of testing criteria to be followed (e.g., black-box versus white-box, or a mixture of the two), the definition of completion criteria (when to stop testing), and the evaluation of test results. Module testing is the main quality control activity that is carried out in this phase. Other related activities may include code inspection to check adherence to coding standards, and software qualities other than functional correctness (e.g., performance, maintainability).

- *Integration and System Testing:* Integration amounts to assembling the application from the set of its components that were developed and tested separately. This step is not always recognized as being separate from coding. In fact, incremental developments may progressively integrate and test components as they are developed. Although the two stages may be integrated, they differ conceptually in the scale of problems that they try to address. The former deals with programming-in-the-small, while the latter with programming-in-the-large. The integration testing occurs after unit testing has commenced and aims on assuring that a group of modules when linked into forming a coherent application will perform a computational task as designed. Often, this is done not in a single step, but incrementally by including progressively new sets of modules, until the entire system is built. At the final stage, the development organization performs system testing on the running application. Once the application has undergone system testing, it may be put through "actual" use within the development organization. Internal standards may be adopted both on the way integration is to be performed, such as top down or bottom up, and on how to design test data and document the testing activity. The purpose of this step is to test the system under simple but realistic conditions. This kind of testing is called *alpha testing.* The initial delivery of software system after alpha testing is often done in two stages. In the first stage, the application is distributed among a selected group of customers prior to its official release. The purpose of this procedure is to perform a kind of controlled experiments to determine, on the basis of feedback from users, whether any changes are necessary prior to the official release. This kind of system testing done by selected customers is called *beta testing.* In the second stage, the system is distributed to the customers and the product enters its official maintenance phase.

The Maintenance Phase

The maintenance phase focuses on all issues related to the evolution of the system, and is associated with error correction, adaptation to new platforms, and modifications due to new functionality that needs to be added. Overall, maintenance is defined as the set of activities that are performed after the system is delivered to the customer. The maintenance encompasses many steps taken on the development phase (e.g., design, testing), but does so in the context of the constraints of an existing software system. In literature, three types of maintenance activities have been identified [12] :

- *Corrective.* Even with the best quality assurance standards and tasks, it is likely that the customer will discover defects in the software. *Corrective maintenance* attempts to correct these defects.

- *Adaptive.* Over time the original environment (e.g. CPU, operating system, peripherals) for which the software was developed is likely to change. *Adaptive maintenance* results in modifications to the software to accommodate changes in its operating environment.

- *Perfective.* As software is used, the customer/user may define additional functionalities that may be of future benefit. *Perfective maintenance* extends the software beyond its original functional requirements, by adding new functionality or improving the existing one.

In the recent studies, it has been found that the cost of maintenance requires more than 60% of the total cost of software development. Out of this cost, about 20% of maintenance costs may be attributed to each of corrective and adaptive maintenance, while over 50% may be attributed to perfective maintenance. Based on this breakdown, the software engineering community has indicated that *evolution* is probably a better term than maintenance, although the latter is used more widely. In this respect, CASE technology may play an important role on minimizing software costs and yielding higher quality products.

CASE Technology

Since the early days of developing software, there has been an awareness of the need for automated tools to help the software developer. Initially, the focus was on tools interfacing the programming language source code with the hardware platform that run the application. These included translators, compilers, assemblers, macro processors, linkers, and loaders. However, as computers became more powerful and the software that ran on them grew larger and more complex, the range of support tools began to expand. In particular, the use of interactive time-sharing systems for software development encouraged the development of syntax directed program editors, debuggers, code analyzers, and program-pretty printers. As computers became more reliable and of greater use, the need for a broader notion of software development became apparent.

Software development came to be viewed as a large-scale engineering activity involving significant effort to establish requirements, design an appropriate solution, implement that solution, test the solution's correctness against the specifications, and finally provide high quality documentation for the end system. This engineering activity is viewed as a long-term process that aims on producing software that is subject to "continuous engineering" or evolution that occurs throughout its operational lifetime. The implication is that the structure of a software system must enable new functionality to be added easily, and detailed records of the requirements, designs, implementations, and testing results to be kept in a way that assists software engineers to efficiently develop and maintain a system. In addition, multiple versions of all artifacts produced during a project must be able to be shared among many software engineers in order to facilitate collaborative software development. Finally, it is recognized that software development is a group activity involving interaction among a number of different people. Team members must be able to cooperate in a controlled manner, and have consistent views on the state of the project.

This view of "programming-in-the-large" generated the need for the development and deployment of a wide range of support tools as illustrated in Figure 3. These tools can help to substantially reduce the complexity that is inherent in medium and large scale software projects by automatically generating and maintaining important software artifacts for the project such as specifications, designs, source code, testing results, source code revisions, and customer reports.

Definition and Objective

Computer Aided Software Engineering (CASE) technology has only become recognizable to system and data-processing professionals in the last ten years, although its beginnings can be traced well back into the 1970s [21]. In the research literature, many definitions and descriptions of CASE have been provided [7,9,14,28]. We choose to present a broad definition as this is found in [19]:

A CASE tool is a computer-based product aimed at supporting one or more software engineering activities within a software development process.

A common analogy is one that compares CASE for developing software systems to Computer-Assisted Design(CAD) and Computer-Assisted Manufacturing(CAM). The introduction of CAD and CAM revolutionized the manufacturing design process. These technologies have helped accelerate the design process, improve design artifacts, and reduce errors and manufacturing expenses. CASE aims on addressing similar issues for the information technology industry. The ultimate goal of CASE technology is to separate the application program's design from program's code implementation. Generally, the more detached the specification and design process is from the actual code generation, the better the potential for a high quality product is.

The objective of CASE is to maintain software requirements, designs, and implementation artifacts in a consistent and usable state, and to support the transformation of system requirements to complete design and testable source code. CASE technology aims to provide project management tools and a flexible environment that facilitates change and automates labour intensive, repetitive and error prone tasks. CASE also aims on introducing automated design verification prototyping, code generation, maintenance, and documentation.

Environment

A typical CASE environment consists of a number of CASE tools operating on a common hardware and software platform. In addition, there are a number of different categories of users of a CASE environment. Some users such as software developers and managers, wish to make use of CASE tools to obtain support in developing application systems and monitoring the progress of a project respectively. On the other hand, administrators are responsible for ensuring that the tools operate on the software and hardware platform available. The system administrator's role is to maintain and update the hardware and software platform itself. Therefore, we can define a CASE environment by emphasizing the importance of these interactions.

A CASE environment is a collection of CASE tools and infrastructure that supports most or all of the interaction that occurs among the environment components, and between the users of the environment and the environment itself.

The different ways of providing the "glue" that links CASE tools together inevitably leads to a spectrum of approaches for implementing a CASE environment. A complete CASE environment system includes the following :

- Support for the common specification, design, implementation, and testing practices with built-in audit capabilities that ensure compliance to a selected process model [2].
- An information repository for storing the elements of software engineering process, including specifications, design, graphics, and pseudo code [25].
- A graphical interface for drawing specification artifacts structure diagrams, flow diagrams, data structures, and project management support data.
- Automated code and dictionary generation from design specifications.
- Prototyping of new designs and reverse engineering (converting existing software back into design specifications for modification and software regeneration).
- A highly integrated set of tools to manage every phase of the development life cycle and provide vital statistics and metrics for the projects.

Benefits

This section presents some of the benefits of CASE technology. However, it should be realized that much of the actual value received from Computer Aided Software Engineering largely depends on how well it is integrated into the software development processes of an organization as is discussed in "*CASE Adoption*" section later. Bellow, a more detailed look at the benefits of applying CASE technology is presented.

In a nutshell, there are two areas of benefits CASE is recognized for. First, the qualitative benefits that pertain to the improvement of specific measurable qualities of the system being built, or the process model being followed. These qualities include time to completion, performance, robustness, testing coverage, and maintainability characteristics of the system to be delivered. Second, the qualitative benefits that pertain to the enhancements in the non measurable qualities of the software life cycle. These include the creation of well documented steps that can be followed for the completion of specific tasks, the ease of monitoring and controlling the different phases of the project, and the ease of repeating the process for similar projects. Other qualitative benefits include increased morale for software developers and management, and better understanding of the process followed. Below, we list some of the qualitative and quantitative benefits of CASE.

- **Quality.** CASE aims to improve end-product quality by automating the transformation of the output from each life cycle phase into an input for the next phase. For example, CASE tools can assist turning designs into source code, or source code into test cases.
- **Flexibility.** CASE aims to improve flexibility by simplifying software changes and testing by automating "*what-if*" scenaria. A central issue in CASE technology is the concept of a repository that maintains specifications, designs, and source code artifacts. This repository allows CASE practitioners to alter design specifications or source code artifacts and examine the impact these changes may have on specific product qualities.

- **Productivity.** CASE aims to accelerate the availability of software by automating software development. Repositories of design can be used to generate skeleton source code that is highly reusable. CASE aims to improve productivity by utilizing a repository for reusable system design elements, by automating labour intensive aspects of the development cycle (such as coding and documentation), and by automating software maintenance [5].

- **Practical Prototyping.** CASE aims to make prototyping practical. Software developers have long been aware that it is difficult for end users to visualize the impact of systems until they see the end result. The labor-intensive nature of software development makes prototyping impractical. With the introduction of CASE technology, this technique becomes more practical.

- **Simplified Maintenance.** CASE aims to simplify maintenance in two ways. Applications can be regenerated from a design repository as specification changes are identified. CASE also provides an opportunity for reverse engineering. Reverse engineering takes existing software and generate part or whole of the design specifications for this software, and consequently store these specifications in a repository for future analysis and modification [16].

- **Project Management.** CASE aims on providing the necessary tools and measurements for the management to follow the progress of the project and identify areas of improvement.

CASE Categories

During the software development cycle, any number of failures can occur. Most failures, however, result from poor planning, insufficient requirements analysis, and ill-conceived design specifications. A good indicator of improper requirements analysis and design specifications is when the implementation phase exceeds the time duration for which it was originally planned. The more effort invested in the requirements specification and software design, the lower the risk of a failed and prolonged implementation phase is.

Table 1 lists common problems and their symptoms occurring at different points in the software development cycle. This illustrates why and where the software development process is likely to break down. As Table 1 illustrates, the cause of most implementation shortfalls and reliability problems stem from improper or insufficient requirements analysis and design specification. During the 1960s and 1970s, several *structural methodologies* were developed to impose rigid structures on the requirements analysis and design specification phase of the software development cycle. These formal, structured mythologies form the backbone of Computer Aided Software Engineering. CASE tools provide automatic support to labor intensive manual tasks. CASE tools also help enhance creativity by enabling software engineers to construct more thoughtful designs in a shorter amount of time.

In this context, CASE tools are divided into two categories: *toolkits* or *workbenches* [7]. The differentiation is based on whether the tool supports a phase of the development life cycle (toolkit) or provides automated support for the full development life cycle (workbench).

Toolkits

A toolkit is a set of tools that support one or more of the software development functions: planning, analysis, programming, maintenance, and project management. Toolkits may be generic or they may support one or more design methods. The functions and the interfaces within one toolkit are tightly integrated. As a result, toolkits can be used alone or in conjunction with other toolkits. Toolkits provide the option of mixing and matching the best products from different vendors and thereby the ability to implement CASE technology in a piecemeal manner [18]. The disadvantages of this fragmented approach are that there is no single repository for the automated design and that the toolkits do not offer a view of the global state of the system being built. Moreover, manual interfaces between toolkits tend to decrease the quality of the process and affect productivity in a negative way.

Planning Toolkit

Information technology planning toolkits automate the process of identifying, categorizing, and prioritizing the areas in which information technology can contribute to the objectives of an organization. Such toolkits provide facilities for documenting and analyzing organizational information requirements. They also provide facilities for decomposing these requirements into data elements and then for structuring the data elements into databases and system requirements.

Analysis Toolkit

Analysis toolkits automate the process of defining system requirements and converting them into corresponding conceptual and functional system designs. Such toolkits typically include facilities such as diagrammatic tools, prototyping facilities, an audit or checking facility to guarantee completeness, a repository to store data flow diagrams, data entity relations, screens, and reports, and a link to a data dictionary or other repositories. The outputs from an analysis toolkit are functional requirements, data structures, database or dataset designs, and screen and report definitions.

Programming Toolkit

Programming tools automate the process of converting a system design into a fully tested and executable program code. Such toolkits include Integrated Development Environments (IDEs) and facilities for generating, maintaining, and checking pseudo code against specifications.

Because automatically generated code is difficult to understand in some cases, such tools have limited appeal to developers.

Maintenance Toolkit

Maintenance toolkits aim to provide support for the software evolution activities. Both types of activities are concerned with the analysis and re-engineering of existing software systems to meet new specifications [17]. A maintenance toolkit includes facilities such as: (1) a maintenance facility to manage and document all changes, (2) an effort, time, risk analysis, and cost estimation facility, (3) a testing facility for generating test data and analyzing output and a code analyzer for analyzing program logic. A maintenance toolkit may also include tools to ensure a smooth transition from test to operational status of the new enhanced system.

Project Management Toolkit

A project management toolkit provides automated software development project management tools. It provide facilities such as: (1) project definition forms, (2) time, cost, and and action plans, (3) task assignment system estimating tools, (4) change and version control mechanisms, (5) project status reporting facilities. It also can include stand-alone tools such a word processing, spreadsheets, and interface into electronic mail.

General Characteristics of a Toolkit

Fundamentally, CASE tools must meet several criteria in order to be successfully adopted and be used in the software development process. Meeting these criteria is essential for the tools to be integrated into the organization's standard practices. It is therefore important for CASE tools to address the following issues [7]:

- **Task complexity reduction.** A major goal of CASE technology is to decompose requirements and designs into manageable components. Their functions is to simplify, explain, and reduce.

- **Presentation flexibility.** CASE tools must provide artifacts that can be easily understood by all parties involved that is by software engineers, managers, and end-users. CASE tools should be to software engineering what CAD (Computer Aided Design) programs are to mechanical and electrical engineering. For end-users and developers alike, it is much easier to comprehend a graphical illustration than to read several pages of textual descriptions.

- **Cost reduction.** Using a CASE tool should be cheaper and more efficient in the long run than building the software system using traditional methods. CASE tools should substantially reduce implementation and maintenance efforts by yielding higher-quality specifications and design.

- **Quantitative and verifiable specifications.** The specifications and designs generated by CASE tools must accurately and concisely articulate the software features and components to be built. Each requirement in the software implementation must be verifiable and traceable back to the requirements document. Performance criteria, boundaries, and error conditions must be established as part of the design. Specifications and designs built or produced by a CASE tool must be adaptable as the requirements and design goals of the project change. A design document that falls out of synchronization with the underlying code becomes useless and may actually cause developers to waste time while maintaining the software.

- **Ease of adoption.** The development of relevant skills and sufficient knowledge to assimilate a technological innovation such as CASE is an iterative process that evolves over time. The longer an organization adopts a technology, the greater will be its accumulated experience and learning. Training and ease of adoption for CASE technology is a cornerstone issue for the successful application of CASE in an organization. Moreover, training into new technology is a very expensive task, and there are many different groups that each may have its unique training needs [13]. CASE tools to be successful must be well supported and easy to use. Furthermore, they must adhere to the basic principles of one of the widely adopted and of successful software process model.

Workbenches

A workbench is an integrated software development environment that supports either the full range of software development life cycle or a large portion of the front-end or back-end of the software development life cycle. Like a toolkit, a workbench can be used in direct support of one or more structured design methods, or it can be generic. The more important advantage of a workbench is the automated output of deliverables from one system development phase into the next phase.

Front-End Workbench

A front-end workbench supports the system specification portion of the software development life cycle from requirements to detailed design. A front-end workbench includes tools to automate the requirement specification, conceptual design, and system design. In some cases, a front-end workbench may include the generation of pseudo code as input to a code generator. The results of this automated process are stored in an automated repository, which is a system specification database available to all system development staff. The system specifications can be used as raw input for generating new systems or for regenerating an existing system.

A front-end workbench includes facilities such as: (1) data, system flow, and logic diagramming tools to facilitate the development of logical system and data flow structures, (2) an automated repository to store and manage system and program specifications that makes the specifications maintainable and reusable, (3) the ability to create or use external databases and dictionaries, (4) interfaces for visual representation of output data, and (5) automatic source code generation utilities.

Back-End Workbench

A back-end workbench supports the code generation portion of the software development life cycle from system design to code generation and testing. A back-end workbench includes tools to automate the generation of the pseudo and actual source code, test cases, and documentation. The output from a back-end workbench is thoroughly tested, documented, and executable programming code. The code is available for the development staff to maintain by using artifacts related to the system that are stored in a repository. In a nutshell, a back-end workbench supports program code generation and testing.

A back-end workbench includes facilities such as: (1) a facility for the input, change, storage, and control of program specifications, (2) prototyping data modeling and data dictionary facilities, (3) screen and report tools for visual representation and code generation of output medium, (4) the ability to create or use external databases and dictionaries, (5) a code generation and documentation facility, and finally (6) testing and debugging tools.

Integrated Workbench

An integrated workbench supports the complete system generation process from requirement specification to program code generation and testing. The workbench combines the tools included in both front-end and back-end workbenches. No single vendor provides an integrated workbench that adequately encompasses all the features of the development process. However, some vendors have developed open interfaces that may result in reasonably complete integrated workbench.

Issues in CASE Adoption

The adoption of new technology into an organization is not a simple matter. This is especially true when adopting a new CASE technology into a large organization. There are many factors that have an impact on the ultimate success or failure of a new CASE tool. Potential users should be aware of these factors, so they can consider what ramifications these may have in their organization.

Any decision to bring a CASE tool into an organization should be made after analyzing both short-term and long-term implications tool adoption may have on the existing process [8]. Over the short-term, organizations adopting CASE tools should be willing to accept issues such as: a potential decrease in productivity, dissatisfaction on the part of employees adopting the new technology, changes to process and methods, potentially extensive training, and significant expense. Over the longer term, CASE organizations must address issues such as: longterm maintenance costs of CASE tools, frequent releases of new versions of the tool, and continuous costs for training new staff and upgrading the skills of existing staff. The success or failure of a CASE adoption effort depends largely on the ability of an organization to manage these short and long term costs. Organizations which have addressed these problems in a well conceived adoption process stand the best chance of success. This approach contrasts with others which focus primarily on the mechanics of choosing a particular tool.

CASE Adoption Stages

Tornatzky in [27] outlines the stages of incorporating a new tool or practice as a variant of a recurring pattern: awareness-problems, matching-selection, adoption commitment, implementation, and routinization. Moreover, the Software Engineering Institute (SEI) has built on findings presented in [27], and has modified them specifically for CASE technology [19]. The resulting model postulates six stages for the CASE adoption process such as: awareness, commitment, selection, trial implementation, and frequency of use. These stages illustrated in Figure 4, represent a cycle where each stage provides the input for the next. Depending on the maturity of an organization prior to the adoption effort, some of the preliminary stages may be already achieved.

Awareness and Commitment

Most organizations perform a preliminary search for information about CASE tools before they make any commitment to adopt CASE technology. The commitment stage consists of the decision process to adopt CASE tools. Commitments from both management and those who will be using, or otherwise being affected by the tool adoption decision, is essential. A common pitfall in this phase is to diminish the importance of commitment from the managers, engineers and support personnel whose daily activities will be affected by the incorporation of this new technology.

Ironically, one of the greatest barriers to achieve increased productivity by utilizing CASE tools is the slow transition to practices required to introduce the organizational changes needed for the successful implementation of CASE [11]. The easiest way to proceed with CASE tool adoption is to dedicate resources for the required changes and to prove that substantial gains in productivity and quality can be achieved. Unfortunately, this evidence is not readily available in most organizations. Other methods of increasing the commitment of tool adoption include developing precise definitions about the changes required and the methods of measuring them, providing education about the need for tool support, and establishing common goals and objectives for the project.

Selection

While most CASE tools can be (and often have been) purchased in isolation, a more effective approach is first the adoption of a standardized strategy, for CASE tool adoption and use, throughout on corporate-wide organization. The strategy should aim to address both the short and long term needs of the organization, based on overall process and technology improvement models and directions.

Based on the needs identified in a corporate-wide CASE tool strategy, the choice of an individual tool can begin. An appropriate tool selection approach includes steps such as: narrowing down the list of available tool options to a small number of tools, determining how the new tool will interact with other tools in the environment, analyzing candidate tools according to both technical and non-technical criteria, and testing the candidate tools.

Trial Implementation

Once a tool has been tentatively selected, it is important to try out the tool on a test project. Many organizations skip this step, as it entails devoting significant resources, including personnel, time, and money [13]. However, only a test project carried out under actual conditions, can help to determine what the tools offer, how they work, how effectively perform their tasks, and what are their shortcomings. These issues are simply too complex to make an informed decision without a trial evaluation. Vendor demonstrations can be helpful, but are not sufficient for making informed decisions. Although most vendors will provide an evaluation copy at little or no cost, organizations must ensure that the evaluation copy of the tool reveals all aspects of the tool. Tools best demonstrate their true capabilities and shortcomings with real data and not in the contrived environment of a vendor's tutorial.

If management deems that the tool assists the development process during a test project, they may support its adoption. However, management and users must be clearly informed about

CASE technology and hold realistic expectations for the potential of the new technology they are about to adopt.

During the hands-on testing period, it is important to perform an objective analysis through a full development cycle, with realistic simulations of database size and multiple users. This type of hands-on testing can give a better idea of the specific functions provided by individual tools and the way various tools can work together.

It has been argued that proper adoption of CASE tools can lead to better quality software [14]. However, finding the best way to use these tools is a difficult process. Small test projects can facilitate training and step-wise tool adoption can allow for confidence building.

Frequency of Use

It has been argued by researchers and practitioners alike, that software maintenance is the longest and most expensive phase of the software life cycle. For a successful and cost effective maintenance process, an infrastructure should be built to facilitate the incorporation of periodic upgrades, provide training, and support corporate decisions related to new process models. Frequent use of a tool is important for its successful adoption. A number of efforts to adopt tools have failed because of the inability to incorporate the tool into the day to day activities and planning.

A major challenge of CASE adoption is the indoctrination of new employees into the system and the continuous enhancement of skills of existing employees. A common pitfall in tool adoption is to provide initial training for a group of early users, that is followed by only minimal ongoing training. Unfortunately, it is the larger group of users who are not CASE tool "pioneers" and potentially require more training.

Another common pitfall is to underestimate the resources necessary to support continual use of complex CASE tools. Many CASE tools require experienced personnel capable of managing the tool databases and responding to problems. A second factor involves possible frequent releases of a CASE tool that are potentially non compatible with each other. While many of the tools have matured to the point where incompatibility problems between versions are minimized, there are still be problems related to configuration mismatches between the tool and its operating environment (i.e., other tools, operation system dependencies, etc).

CASE Tools Integration

While this article is not intended as a tutorial on CASE tool integration, a background in the types and techniques of tool integration may be insightful. A thorough analysis of the issues pertaining to the CASE tool integration can be found in [4], while a more complete explanation of the types of integration is available in [26]. The most widely known model of integration is the five-dimensional one that distinguished five areas [30] namely: platform, presentation, control, data, and process integration classification. However, analysis of CASE tool integration is usually separated only into three functional areas of those five types, namely *data*, *control*, and *presentation* integration [29]. The following subsections describe each of them in detail.

Platform Integration

Platform integration refers to the incorporation of a tool with a common set of services provided by the computing environment. In some respect, this is the least interesting form of integration because it does not deal directly with tool-to-tool integration [23].

Presentation Integration

Presentation integration refers to the provision of a consistent user interface across various tools. Such consistency can greatly simplify the use of a tool-set as illustrated in Figure 5.

In addition, the time and the cost of comprehensive training and support can be reduced. Standardization of user interfaces can ultimately lead to greater frequency of use and better adoption. Organizations have long tried to achieve a form of presentation integration by developing user interface standards for internal tools and by providing graphical user interface "wrappers" around (primarily command line based) external tools [3].

Control Integration

Control integration refers to the ability of tools to inform other tools of their actions and to request actions by other tools through a trigger mechanism as shown in Figure 6. A very rudimentary form of control integration is represented by command line invocation of one tool by another. Unfortunately, command line invocation is inadequate to provide the level of integration required by users. Users require that integration occurs at the programmatic interface of each individual tool. Thus, at the moment of an action taken within a CASE tool, notification to other tools and the maintenance of a global consistent state between tools would ideally be immediate.

Organizations have long achieved a rudimentary form of control integration by using mechanisms such as UNIX shell scripts to invoke tools in order to achieve an ordering of tool functioning. An increasing number of tools are incorporating more sophisticated mechanisms such as programmatic interfaces (APIs) that provide access to the inner-working of each tool. Unfortunately, the use of these interfaces often leads to point-to-point-integration of individual tools. This is also amplified by the fact that there is no universal standard for the format and functionality of programmatic interfaces of CASE tools. Such point-to-point integration is expensive both to create and maintain, and effectively limit the user's flexibility when replacing a tool.

A more complex but promising mechanism includes a monitor that receives event notifications or requests and subsequently send appropriate notifications and requests to other tools in the environment. The technique requires that the monitor maintain a global state of both the tools in the environment and the actions that need to be triggered. This technique offers the advantage of centralizing control integration processing. Unfortunately, tool vendors have yet to agree on the events involved in the sending and receiving of a message, however there is industry interest in generating such standards. Markup languages and XML (eXtensible Markup Language) based protocols such as SOAP, WSDL, WSFL may play an important role in the near future towards control integration [31].

Data Integration

Data integration refers to the transfer of data between tools, and the definition of relationship mappings between data schemas utilized by different tools. One common method of data integration requires that individual tools to agree on specific interchange formats or interfaces. This approach is relatively simple to implement and widely applicable to many types of tools. Perhaps the most common of such interchange formats is represented by ASCII files. More elaborate interchange standards, such as CDIF (CASE Data Interchange Format) [6] are also supported by a number of tool vendors. Such methods, however, provide only for the exchange of data and are not effective at establishing links between data maintained by different tools or at maintaining the semantic context of data.

The second approach to data integration has been the development of filters which extract portions of data from individual tools and store this data into a secondary database for processing by other tools. This approach has been commonly used to extract data from tools, organize the data along some schema, store it in a central database, and then use the data to generate reports and documents as shown in Figure 7. This approach allows for the generation of arbitrary relationships between data but like the previous approach results in point-to-point integration(between filters and tools), as well as duplication of data in the individual tools and in central databases.

A third method for achieving data integration involves the development of a shared repository in which a variety tools store information. A fully functioning repository would provide the capability of maintaining a core semantic content of objects together with tool-specific views, and because of the common dictionary would permit several tools to work together [15]. Markup languages and standards such as RDF and XML schema may play an important role in the near future towards data integration [31].

Process Integration

Process integration refers to the automation of the sequence of activities to support the organization's defined process for the software life cycle. To achieve a high level of process integration, mechanisms for presentation, control, and data integration are used. While process integration had been the overall goal of many integration attempts, little is known about the characteristics and parameters to assess the quality of process integration. Ideally, a well integrated process would support an organization's activities without mandating a single process model. A well integrated process would ensure that milestones and standards are met, and provide flexibility to its users to alter specific task for meeting their objectives.

Corporations involved in the development of integrated environments must address the conflicting requirements for standardization and flexibility. In this sense, an environment which provides an adequate degree of process integration, while at the same time allowing adequate flexibility to support a wide customer base needs to be considered.

CASE Impact

The potential impact of a tool on an organization is difficult to predict because many factors are involved. The impact of CASE technology depends on a collection of interrelated factors as illustrated in Figure 8. Although much has been published about CASE, the early anecdotal studies of the impact of CASE tools did not distinguish among the effects of tools, associated changes to the organization's process and methods, or the way in which tools were adopted. Such factors can have major effects on productivity or quality. Additional research will be needed to investigate these factors.

Figure 8 suggest that tool characteristics, while important, represent only one of the factors that determine the effectiveness of tools. There is a need to carefully account for the organization specific factors, such as size, resources, and culture [10]. The way in which tools are adopted can also have an additional long term impact.

Technological Trends in CASE

In the past, CASE technology focused on general purpose requirements analysis and design specification tools. Nowadays new developments in CASE technology are emphasizing on *specialty development tools* [24]. As with all developing technologies, certain parts of Computer Aided Software Engineering are more advanced than others. This makes it difficult to summarize today's state of the art. However, as Figure 9 illustrates, there is a general trend of emerging tools being built to cover the entire software development cycle, including automatic code generation.

Ideally, we would like to write as fewer code as possible. We may imagine a software development environment so powerful and robust that we simply input the application's requirements specification, push a magic button, and out comes the implemented code, ready for release to the end-user community. Testing would be unnecessary because this magic "application generator" produces perfectly correct software. If the application's requirements changed at some point, we would merely update the requirements specification, push the magic button again, and out would come fresh code implementing the new requirements. It means that even the beta testing phase can be bypassed because the generated code is "bug free". The framework for bridging the automatic code generation gap is already in place. The ability to facilitate the generated software is the key to making the next generation CASE environment a successful commercial reality. Unfortunately, we are still far from this ideal scenario. However, it is not unlikely to see in the future tools in which specifications are converted to source code much like nowadays source code is converted to machine code by using a compiler.

Conclusion

The term Computer Aided Software Engineering (CASE) was first applied to tools that provided support for the analysis and design phases of software development cycle. Many of the early tools assisted software developers on fundamental tasks but were infrequently used due in part of the lack of automated support.

Other categories of tools that have emerged the past decade provide support for every phase of the software life cycle. Currently, the vision of CASE is that of an interrelated set of tools which provide specialized support in different aspects and phases of the software development process. These include tools which support analysis, design, project management, source code, configuration management, documentation, and maintenance.

Finally, because of the nature of the software development problem, many organizations are attempting to increase the productivity and quality of their software development efforts. Some have turned their attention to CASE tools as an aid in developing better software. Initially, CASE tools were hailed as a panacea for software development problems, with the assumption that the use of tools would by themselves produce dramatic increase in productivity. Recently, there has been a recognition that tools represent only one factor in the improvement of the software development process. As an epilogue, CASE tool technology provides a promising trend on standardizing and automating labour intensive and error-prone tasks in the software development process and deserves with no doubt the continuous attention of the software engineering community.

FILENAME.APP = 6911BI00.tex

Bibliography

1. E.J. Braude, *Software Engineering : An Object-Oriented Perspective*, John-Wiley & Sons, 2001.

2. A.W. Brown, Why evaluating CASE environments is different from evaluating CASE tools, In *Proceedings of the IEEE Symposium of Quality Software Development Tools*, Los Alamitos, CA, USA, pp. 4-13, 1994.

3. A.W. Brown, D.J. Carney, E.J. Morris, D.B. Smith, and P.F. Zarrella, *Principles of CASE Tool Integration*, New York : Oxford University Press, 1994.

4. A.W. Brown and J.A. McDermid, On integration and reuse in a software development environment, *Software Engineering Environments'91*, F.Long and M. Tedd (Editors), Ellis Horwood, March 1991.

5. T. Bruckhaus, N.H. Madhavji, I. Janssen, and J. Henshaw, The impact of tools on software productivity, *IEEE Software*, pp. 29-38, September 1996.

6. C. Chappell, V. Downes, and T. Tully, *CDIF - Integrated Meta-model, Common Subject Area*, Interim Standard, Electronic Industries Association (EIA), 1996.

7. E. Chikofsky, *Computer Aided Software Engineering*, Los Alamitos : IEEE Computer Society Press, 1993.

8. A.M. Christie, *Software Process Automation : The Technology and its Adoption*, Berlin : Springer-Verlag, 1995.

9. A.S. Fisher, *CASE : Using Software Development Tools*, New York : John Wiley & Sons, 1991.

10. T. Flecher, J. Hunt, *Software Engineering and CASE: Bridging the Culture Gap*, New York: McGraw-Hill, 1993.

11. G. Forte, CASE: an industry in flux (Part 2), *CASE Outlook*, 2(4), pp. 1-17, 1988.

12. C. Ghezzi, M. Jazayeri, and D. Mandrioli, *Fundamentals of Software Engineering*, New Jersey: Prentice-Hall, 1991.

13. C.F. Kemerer, How the learning curve affects CASE tool adoption, *IEEE Software*, pp. 23-28, May 1992.

14. T.G. Lewis, *CASE: Computer Aided Software Engineering*, New York: Van Nostrand Reinhold, 1991.

15. P.K. Linos, ToolCASE: a repository of computer-aided software engineering tools, *ACM SIGSOFT Software Engineering Notes*, pp. 74-78, April 1992.

16. J. Mayrand, C. Leblanc, E.M. Merlo, Experiment on the automatic detection of function clones in a software system using metrics, in *Proceedings of IEEE International Conference on Software Maintenance*, pp. 224-253, 1996

17. E.M. Merlo, K. Kontogiannis, J.F. Girard, Structural and behavioral code representation for program understanding, In *Proceedings of* 5th *International Workshop on Computer-Aided Software Engineering (CASE)*, pp. 106-108, 1992.

18. R. Mylls, *Information Engineering: CASE, Practices and Techniques*, New York: John Wiley & Sons, 1994.

19. K.S. Oakes, D. Smith, and E. Morris, Guide to CASE adoption, Technical Report CMU/SEI-92-TR-15, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, USA, November 1992.

20. R.W. Peach, *The ISO 9000 Handbook*, Chicago: Irwin Professional Publication, 1997.

21. S.L. Pfleeger and W. Menezes, Marketing technology to software practitioners, *IEEE Software*, pp. 27-33, January/February 2000.

22. R.S. Pressman, *Software Engineering: A Practitioner's Approach*, Boston: McGraw-Hill, 2000.

23. D. Schefstrom and G. Van Den Broek, *Tool Integration: Environments and Frameworks*, John Wiley & Sons, 1993.

24. K. Spurr and P. Layzell, *CASE: Current Practice, Future Prospects*, Chichester: John Wiley & Sons, 1992.

25. A. Tannenbaum, *Implementing a Corporate Repository: The Methods Meet Reality*, New York: John Wiley & Sons, 1994.

26. I. Thomas and B.A. Nejmeh, Definition of tool integration for environments, *IEEE Software*, 9(2), pp. 15-23, 1992.

27. L. Tornatzky and M. Fleischer, *The Process of Technological Innovation*, Lexington Books, 1990.

28. L.E. Towner and J. Ranade, *CASE: Concepts and Implementation*, New York: Intertext Publications - IBM Series, 1989.

29. K.C. Wallnau, Issues and techniques of CASE integration with configuration management, Technical Report CMU/SEI-92-TR-5, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, USA, March 1992.

30. A. Wasserman, Tool integration in software engineering environment, F.Long (Editor), *Lecture Notes in Computer Science*, Vol. 467, Springer-Verlag, pp. 137-149, 1990.

31. World Wide Web Consortium (W3C): *Leading the Web to its Full Potential*, available at: http://www.w3.org/.

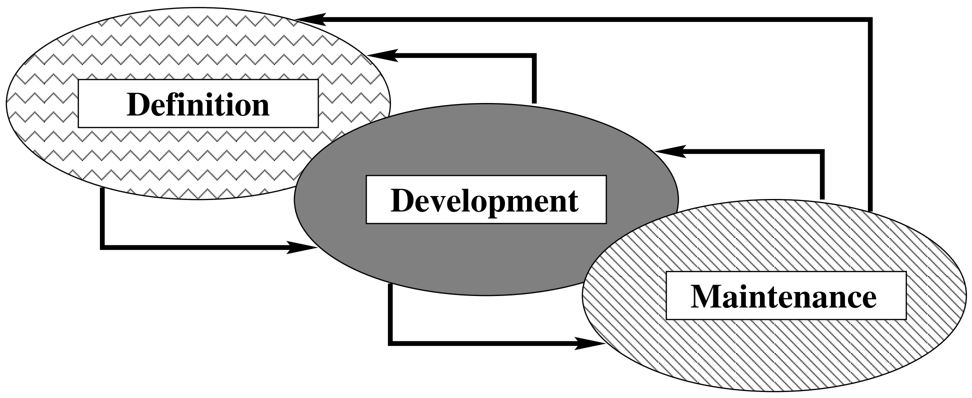
FILENAME.APP = 6911FL00.tex

FILENAME.APP = 6911TB01.tex

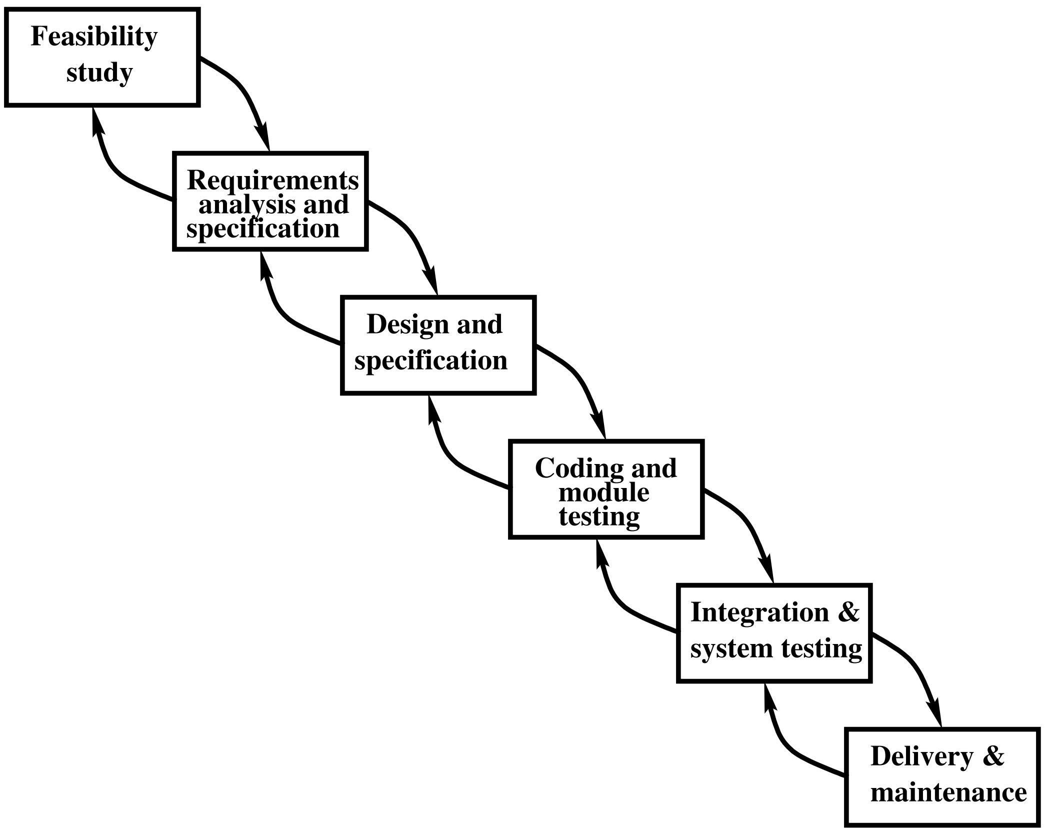
Table 1: Failure Points in the Software Life Cycle.

Software Development Process	**Failure Symptom**
Requirements Analysis	No written requirements
	Incompletely specified requirements
	No user interface mock-up
	No end-user involvement
Design Specification	Lack of, or insufficient, design documents
	Poorly specified data structures and file formats
	Infrequent or no design reviews
Implementation	Lack of, or insufficient, coding standards
	Infrequent or no code reviews
	Poor in-line code documentation
Unit Test & Integration	Insufficient module testing
	Lack of proper or complete test suites
	Lack of an independent quality assurance group
Beta Test Release	Complete lack of a beta test
	Insufficient duration for beta test
	Insufficient number of beta testers
	Wrong beta testers selected
Maintenance	Too many bug reports
	Fixing one bug introduces new bugs

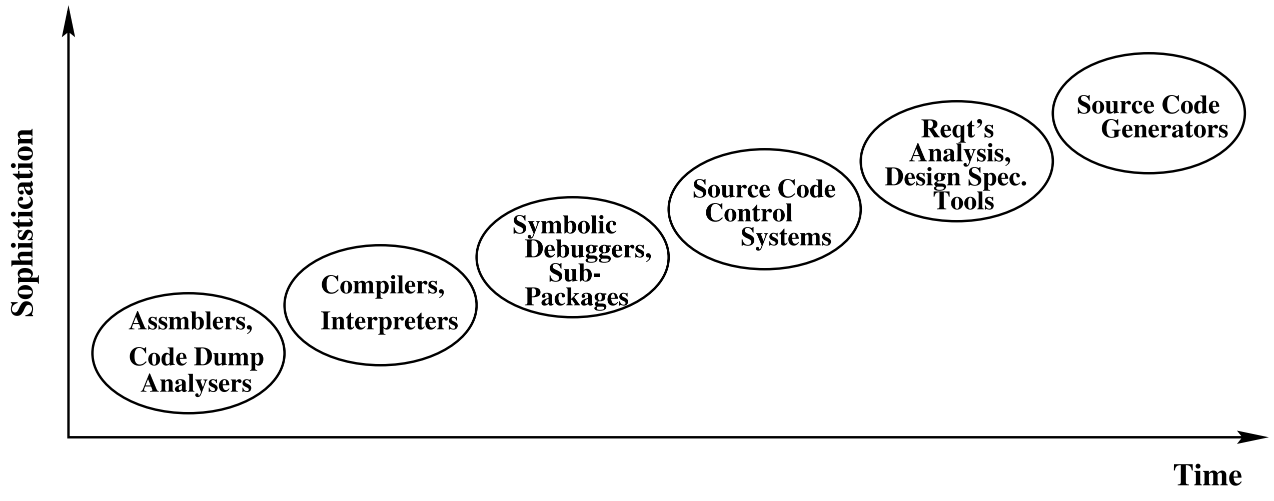
FILENAME.APP = 6911FG01.eps



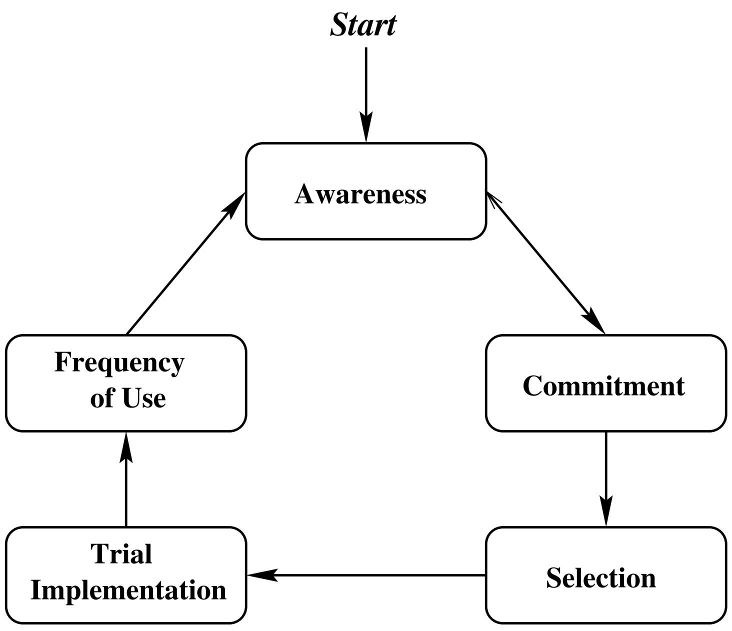
FILENAME.APP = 6911FG02.eps


Feasibility study
Requirements analysis and specification
Design and specification
Coding and module testing
Integration & system testing
Delivery & maintenance

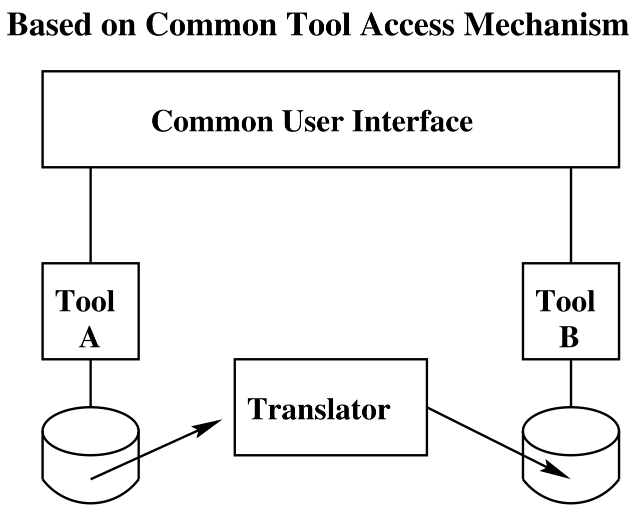
FILENAME.APP = 6911FG03.eps


Sophistication
Time
Assmblers, Code Dump Analysers
Compilers, Interpreters
Symbolic Debuggers, Sub-Packages
Source Code Control Systems
Reqt's Analysis, Design Spec. Tools
Source Code Generators

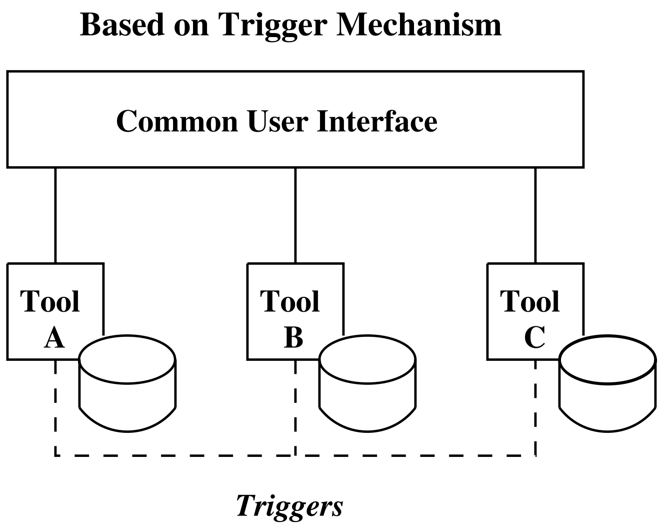
FILENAME.APP = 6911FG04.eps



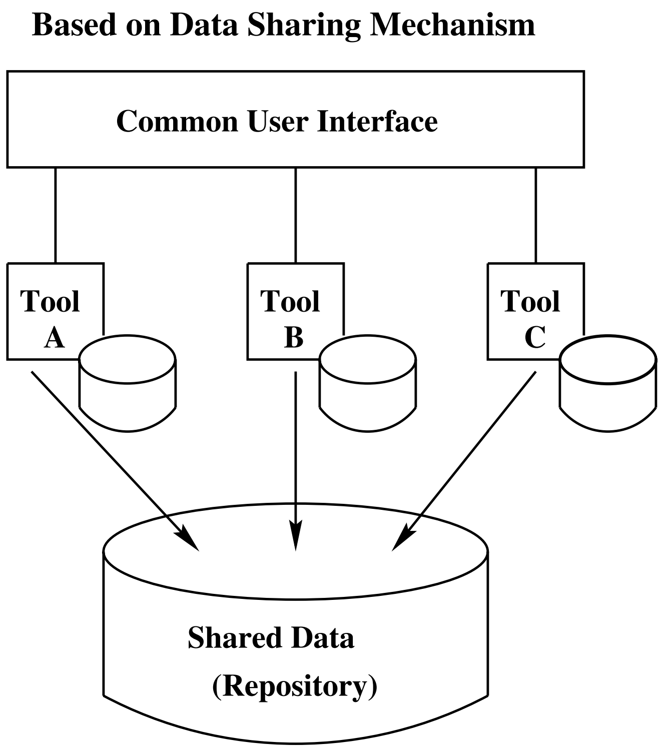
FILENAME.APP = 6911FG05.eps



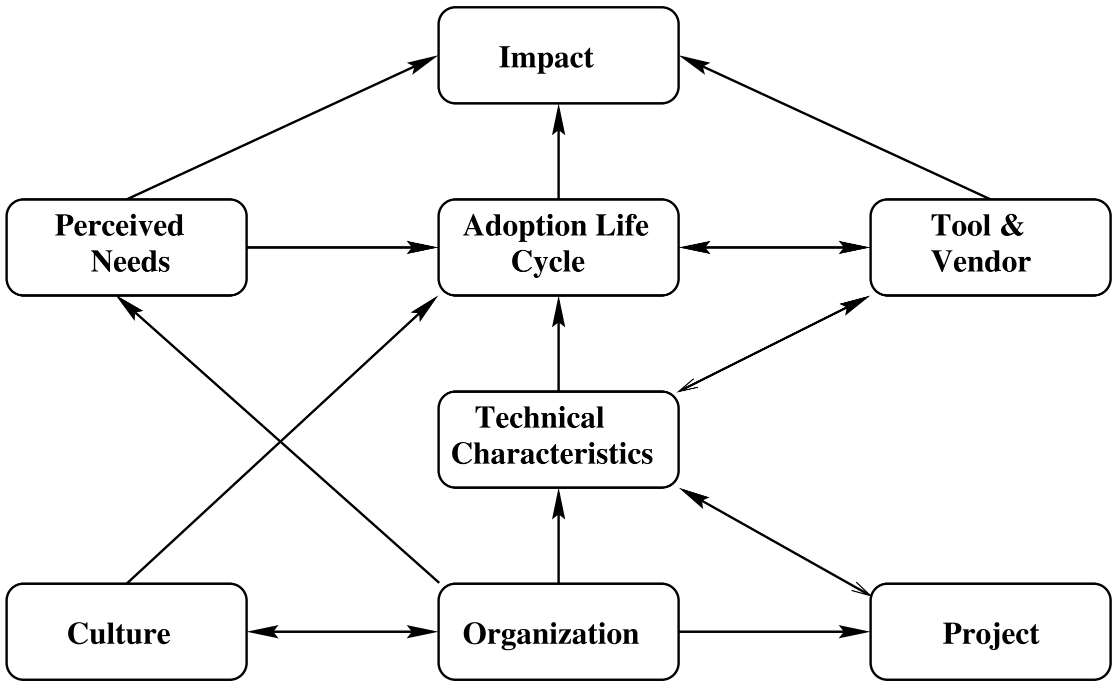
FILENAME.APP = 6911FG06.eps



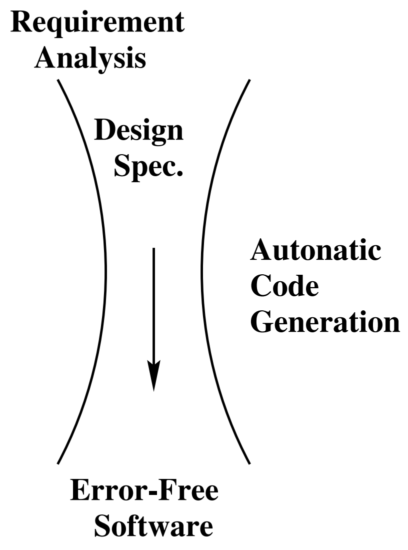
FILENAME.APP = 6911FG07.eps



FILENAME.APP = 6911FG08.eps



FILENAME.APP = 6911FG09.eps


Requirement Analysis
Design Spec.
Autonatic Code Generation
Error-Free Software