COMPUTER SCIENCE EDUCATION 5, 189–210, (1994)

Integrating CASE Technology into Software Engineering Education

Hossein Saiedian
University of Nebraska at Omaha

Cost-effective development of software has been extensively studied within the past decade, and a considerable number of technologies have been introduced, some of which have proven effective in practical cases. These technologies include software engineering methodologies, integrated environments, special purpose languages, and automated tools. Among these, automated or *CASE* tools have received significant recognition as having the potential to increase the productivity in software development and maintenance. We briefly discuss the role of CASE tools in both developing and maintaining software, but focus our attention on the role of CASE technology in the context of software engineering education and the importance of integrating CASE tools in software engineering courses. The qualitative and quantitative benefits of CASE tools for preparing our students to participate in large software projects and subsequent career growth are discussed. Our approach in and choice of CASE tools for use in the classrooms is also presented. At the end, the use of CASE tools to *accommodate* students' learning process and the potential role of these tools in enhancing students' understanding of software engineering concepts are discussed.

1. INTRODUCTION

A phenomenal proliferation of computers into society has been observed during the past three decades. As the cost/performance ratio of com-

This research was partially funded by a Summer Fellowship grant from the Center for Faculty Development, University of Nebraska at Omaha.

The author is grateful to an anonymous referee for important suggestions that substantially improved the overall presentation of this article. My next-door colleague, Thomas Spencer, provided me with the descriptions of *SetPlayer, GraphPack, NPDA, XTuring,* and *Xtango.* He is one of the principle designers of *GraphPack.*

Correspondence and requests for reprints should be sent to Hossein Saiedian, Dept. of Computer Science DSC 203, University of Nebraska, Omaha, NE 68182.

tools for further study. In Section 3 we review the industry's view of CASE technology. In Section 4 we discuss the integration of CASE technology into Computer Science education, and in Section 5 we elaborate on CASE tools as a learning aid for students. In particular, we describe how a CASE tool can be used to improve a student's understanding of *formal methods* of software development. We conclude with Section 6.

2. WHAT IS CASE?

During the past three decades many methodologies have been suggested to improve the software development process. Similarly, many automated tools have been introduced to support these methodologies. In the mid-1980s, the term "CASE" was coined to describe these tools, and it is used to describe any computer-aided tool that supports some software development activity. It can be as simple as a single editing tool or as complex as an integrated software engineering environment that includes many tools, databases, operating systems, and networks which together provide an integrated, flexible development environment to support all aspects of software engineering activities. Major advantages of CASE tools can be summarized as follows [3]:

- Automation of the entire software development process, which includes the critical early analysis and design phases;
- Interactive development environments characterized by improved response time and on-line error detection;
- Automatic analysis of specifications for completeness and consistency; and
- Generation of efficient code from design specifications.

Other advantages include support for prototyping, simulation, and project management and planning, all through sophisticated, attractive graphical user interfaces.

CASE tools may be categorized in a variety of ways. They can be categorized as *upper* or *lower* CASE. Upper CASE (also known as front-end CASE) products provide support for the early phases of development such as planning, analysis, and design. Lower CASE (or back-end CASE) products provide support for the later stages of development such as code generation, integration, and testing. Another common method of categorizing CASE is to consider its role as an instrument for different groups of people (e.g., programmers vs. managers), or by its functionality. Some CASE products may fit into more than one category. For a comprehensive survey of CASE tools see [3–5]. In the remainder of this sec-

tenance activities and/or improving the structure of existing programs in order to reduce maintenance efforts. For example, restructuring and reengineering tools may analyze an unstructured program and reproduce a structured, optimized version. Pretty-printer tools may be used to improve the visual organization of existing programs and unify coding conventions used in different programs, making programs more readable, and by extension, more maintainable. Reverse engineering tools may assist in the postdevelopment analysis of existing programs. For example, a CASE tool may take as input a program that has no documentation and deduce its logical design and specifications, and in the process identify its missing and/or inconsistent units.

3. INDUSTRIAL VIEW OF CASE TECHNOLOGY

The role of CASE technology in the real world is so prevalent that it necessitates integrating these concepts in Computer Science education. According to a study by Clark and Hughes [6], the CASE market in 1990 was estimated at $2 billion. Of the organizations surveyed, only about one-third had obtained and utilized CASE tools; the rest (68.8%) had not used CASE tools, primarily because of a lack of knowledge and familiarity with them. Given these reasons for not utilizing CASE tools, we believe that a CASE approach to software development and integration of CASE concepts into Computer Science education would better prepare our new graduates for career growth, infuse the technology into industry, and cause the statistics just cited to drop.

The advantages of CASE are realized in the long term, but testimony abounds as to how they boost productivity, upgrade quality, and prevent defects [7]. Better communication and documentation have been the most frequently mentioned reasons for this improvement [8]. Hughes and Clark [9] identified, through observation of software engineering in practice, five stages of CASE usage, all of which are expensive. We believe two of these can be reduced or eliminated, with improved CASE tool education. The stages are (1) disenchantment, (2) resignation, (3) commitment, (4) implementation, and (5) maturity.

The first stage, disenchantment, can be eliminated if the user knows which tools are useful replacements for which manual methods as well as the desired application before the investment is made. The second stage, resignation, involves a half-hearted adoption of the tool the organization is stuck with. The reason this is usually an unenthusiastic embrace is not because the tool is lacking in some feature or quality, but because the user does not fully understand that CASE requires the adoption and enforcement of a methodology. If an organization and its

incomplete. The feedback would act more as a teaching tool than it would if used in a business application, because one does not expect the machine to do all the thinking for the student, but at the same time, one would not want to discourage him or her with a deluge of syntactic errors. In using CASE tools in industry, one wishes to speed up the development processes as much as possible. In using CASE tools in education, our goal is to speed up only the communication and documentation activities that are tedious to the student and may otherwise prevent him or her from designing at the proper levels of detail.

In the next section we elaborate on how integrating CASE technology in Computer Science programs and in particular, on how a CASE-based approach to software engineering will make new Computer Science graduates more "marketable" and provide them with greater opportunities for career growth.

4. INTEGRATING CASE INTO THE CURRICULUM

The first curriculum question is: Should there be a separate course in CASE tools and methodologies, or should the CASE fundamentals be integrated into a software engineering (or a similar) course? We believe the latter approach is more appropriate. There are several reasons for our argument:

- A separate course in CASE would likely be a course in a specific tool instead of a course in automated software engineering methodologies.
- If students are exposed to an appropriate CASE tool within a software engineering course, they will see it as part of a "normal" environment for understanding, developing, and maintaining software, and, in the process, they will experience the potentials and limitations of such tools. As students do their software engineering projects using a CASE tool, they will learn how to automate software development in the future and thus suffer less when they are exposed to the concepts in a real job.
- Since most software engineering courses are project-intensive, the use of an appropriate CASE tool would permit students to concentrate on the creative aspects of the project, while allowing the tool to perform the repetitive and tedious tasks.
- There are always problems associated with developing and offering a new course. Sometimes it may take a long time before a course is approved.

inadequate. And, of course, the faculty member needs to learn the CASE tool so that he or she can present it to the students and answer their questions.[2]

4.1 A Typical CASE Environment

An ideal CASE environment includes an integrated collection of tools that support the entire software development life cycle, from requirements analysis to implementation and testing and preparation of a user's manual. There are relatively large numbers of such tools available for both the MS-DOS environment and UNIX-based workstations.[3] An example is **Team***work*[4] which we have used for both of our graduate and undergraduate software engineering courses. **Team***work* has a number of components. The most important ones are briefly described.

Team*work*/SA is a multiuser, multitasking, workstation-based environment for systems analysis (based on the Yourdon–DeMarco structured analysis). It allows students to easily and quickly build, store, review, and maintain structured specifications, including dataflow diagrams, task specifications, and a data dictionary. The automated facilities of **Team***work*/SA help students ensure quality by checking for the completeness and correctness of models. The **Team***work*/SA editor and consistency check interact with each other to detect errors as specifications are being created. **Team***work*/SA has two special-purpose editors, one for creating dataflow diagrams (called a DFD Editor) and one for process specifications (called a P-Spec Editor). **Team***work*/SA supports a multiple-window interface, allowing students to edit and view several diagrams at the same time.

Team*work*/SD is an environment for systems design that allows students to create structured designs (based on the Yourdon–Constantine method). It provides facilities for partitioning a system into single-function modules, checking and quantifying the level of coupling, and extensive cross-referencing features. For example, students can list the mod-

[2]Most of these time-consuming efforts need to be done only once and we must continuously remind ourselves that the end result is worth our effort.

[3]Actually Mynatt and Leventhal [13] refer to a UNIX environment as an example of a general CASE environment. We share their views. A typical UNIX environment includes over 200 utility tools, which can be used for creating and archiving programs, data files, and documents, for manipulating files and directories, and for a variety of other tasks such as debugging and version controlling. UNIX systems are conceptually simple and flexible and allow programs to be rapidly developed, tested, and re-used. Saiedian and Wileman [14] provide a detailed discussion of the advantages of a UNIX environment for improved productivity and management of computing resources.

[4]**Team***work* is a registered trademark of Cadre Technologies Inc.

ative activity. Since in the real world software development is a team-oriented activity, university students should be exposed to such activity in order to be able to meet the needs and demands of industry. This can sometimes be achieved through internship, but internship opportunities are not available to all. However, experience with a realistic group project in an academic environment can be a valuable substitute [15,16]. That is why team projects are becoming an essential part of courses like software engineering, database management systems, and systems analysis and design. Early participation in and exposure to team projects helps a student to understand the communication aspects, interpersonal skills, and so forth that are vital for professional competence in the real world. As noted by Bruegge [15], by carrying out the entire software development life cycle, students observe and learn how to execute all the roles (such as planner, analyst, designer, project leader, liaison, and system integrator) that arise during the realization of a software project.

4.2.2 Team Organization. The class is divided into groups of five or six students.[5] Students are strongly encouraged to form their own groups of compatible people. Each team elects a team leader who is responsible for coordinating the activities of the other team members as well as for communicating with the instructor.[6] The team leader is also responsible for final technical decisions and making sure everyone attends meetings and does his or her share of the work. The projects will take most of the semester, with major write-ups at three-week intervals. There will also be a formal oral presentation and a final demonstration of the finished project at the end of the semester.

Students are graded on the quality of the work they produce, however, they are requested to create professional-looking documents, not only for "clients," but also to improve communication among themselves. Portfolios, labeled theme binders, and the like are recommended. Each part of the project is graded based on the accuracy, consistency, and completeness of its content as well as its organization (e.g., appropriate title, section, and paragraph names) and appearance (e.g., consistent page numbers).

4.2.3 Project Structure. The course project is organized similarly to the one described by Kant [17] and Tomayko [18]. Basically, the students must produce a document for each major phase of the waterfall life

[5]Large groups are chosen partly to force students to face issues of project management. Smaller groups teach fewer lessons about software development in groups.

[6]Teams are encouraged to assign other roles to members, such as "project administrator," "configuration manager," "quality assurance manager," or "maintenance engineer."

core system. Thus, major modules are identified; the relationship between these modules is established, and the interfaces are clearly defined. Minimally, each module includes the number and type of its formal parameters, an English description of its behavior, a set of pre- and postconditions, and a listing of all modules (along with an appropriate number and type of actual parameters) that are invoked. The end result is usually given in terms of structure charts which include at least 15 to 20 modules. CASE tools are also used quite extensively during this stages for preparing the structure chart.

4.2.3.3 Detailed Design. During this phase of the project, students define all data structures and devise algorithms needed to realize the behavior of each module. In addition, error-handling parts, concrete representations of certain implementation parts (e.g, data structure declarations), and side effects are clearly defined, and alternative design strategies are explored. Metrics (e.g., effort/time estimation) are applied to ensure that the project is within control and manageable. The document for this part of the project runs about 30 to 40 pages.

4.2.3.4 Implementation and Testing. During implementation design algorithms and data structures are translated into program code. Students must follow programming guidelines very carefully, and they are asked to gear their overall implementation toward modularity and expandability rather than speed. Individual modules are tested first. Each team must carefully plan and document the order in which modules are to be integrated. Functional testing, performance testing, and acceptance testing are scheduled. Each team prepares a test plan that includes a statement of objectives and success criteria, integration plans, testing methodologies, and schedules. Some team members use CASE tools at this stage to produce partial Ada code for Ada packages.

Each team is also responsible for developing a user's manual. The user's manual should have a structure that is evident both to someone reading it straight through and someone who will look for a particular topic. The user's manual is organized as follows: a table of contents, an introduction that concisely describes the system, an overall description of the style of user interaction, detailed systems operations, a list of known features and deficiencies, and an index.

CASE tools are used primarily during the first three phases of development. The instructor gives introductory lectures and/or materials on using the CASE tools prior to a particular phase of the project. Table 1 gives an approximate schedule for each of these phases during a 15-week semester. Each part of the project (including presentation and demonstration) is worth 100 points for a total of 500 points and accounts

enhanced their productivity, generated better documentation, made maintenance easier, improved project management, and forced them to adhere to a structured methodology. The last survey item was as follows: "Adopting CASE would lead to an increase in the software development productivity of your team," and to this 83% responded, "strongly agree."

To more closely measure the impact of CASE tools on students' performance, we compared the software engineering projects produced by the students in two different semesters. The criteria for comparison were completeness, consistency, and resources (both manpower and time) used. During the summer of 1992 five software engineering teams designed and implemented five independent modules of a major record-keeping and report-generating system. (The whole system contained over 14,000 lines of codes each module, on average was about 3,000 lines.) During the fall of 1992 another five teams developed five different versions of a class-scheduling system. Each system consisted of more than 3,000 lines of code. During the fall of 1993 another five teams developed five different versions of a simplified airline reservation system. Each version included, on average, nearly 3,500 lines of code. Thus, even though the nature of these projects differed, the implemented programs for each team consisted of at least 3,000 lines of code (in Ada, C, or Pascal).

We studied and compared these projects in terms of their completeness (with respect to the specifications), consistency, and the resources used. For measuring the resources used, we used the log reports generated by each team. The log reports included an approximate count of number of log-ins, number of compilations, and approximate CPU time used.[8]

Our study showed that those teams that used a CASE tool produced much better systems and much higher quality documents. Their systems were more complete with respect to their specification.[9] Overall, those teams that used the CASE tool completed over 80% of their proposed system, while those who did not use a CASE tool completed less than 75% of their system. Furthermore, the teams that used CASE tool produced systems (comprising the requirement specifications, design specification, code, user's manual, and test plans) that were more consistent and uniform. The overall resources used by these teams were substan-

[8]At the beginning of each semester, we ask the students in each software engineering team to maintain a log of the times they spent on the project and a description of what they did during each time, the reason for each computer run, the amount of CPU time, the result of each run, the number of changes, and so forth. The students are told that the logs will help them make better predictions of the times and resources needed during each phase of the software development.

[9]The measure was the degree to which the final system met the requirements identified and documented during the requirements specification.

(abstract) mathematical description of an information system using formal methods before building the system itself. The reason for doing so is to achieve more precision in the description and to explore the validity of a design by reasoning about the descriptions. Since the formal methods deal with the semantics aspect of an information system, they can help students determine which functional properties to capture in an abstract specification and which to focus on in concrete design.

Software engineering, like most other fields (particularly engineering) that have been shown to be amenable to such treatment, can benefit greatly from mathematical treatment. Computer Science curricula thus must embody enough material to ensure that future software engineers are able to use scientific knowledge and formal methods in the construction of computer programs.[11]

One factor limiting the use of formal methods is the lack of investment in automated tools and support structures to reduce the efforts of applying these methods in academic environments. In fact, lack of support tools is often seen as a major barrier to learning formal methods. A key factor in the acceptance of high-level languages has been the presence of a comprehensive set of tools to support the user. If formal languages are to achieve the same level of acceptance, they too require extensive automated support. Support tools may reduce learning time, thereby aiding their widespread use. Automated tools may include (1) a special editing environment, (2) syntax checkers, (3) animation tools, and (4) refinement and proof tools.

A special editing environment for a formal language would, for example, provide a student with a number of pop-up menus from which he or she could view global schemas, local schemas, state schemas, operation schemas, or defined sets. The editor would also make schema creation, modification, and deletion more flexible. In addition, good interfaces to formal specification languages, transformation tools for taking established nonformal methods and converting them into formal methods, and tools for interfering from specifications to assist software validation are needed.

5.2 Visualization Tools for Complex Concepts

It has been our experience that students learn more by active participation than by just observing. Theoretical concepts (such as discrete mathematics and graph theory concepts) should be reinforced with hands-on experience in labs. Since such courses should be taught early in college (to provide the necessary background for higher level courses), educa-

[11]For a detailed exposition of this argument see [19].

30 animations of algorithms and data structures studied in the first two years of our Computer Science curriculum including bubble sort, insertion sort, shell sort, merge sort, quick sort, depth first search, binary search, minimum spanning trees, and many others.

For formal specification purposes, a tool, with similar functions as the above tools can help students in many ways. For example, an integrated environment may provide specialized editors,[12] a static analyzer (parser and type checker), and refinement tools. Visualization is important and can help students learn the concepts more effectively. A specialized CASE environment for popular formal notations such as Z [21] or VDM [22] can assist students in the creation of specification schemas through the use of a visual notation and present the essential structures in diagrammatical form that could enhance learning. Furthermore, a tool that would extract from specifications in Z or VDM a definition for another diagrammatical tool, for example, a structure chart, and generate the corresponding charts would be even more interesting as it would teach students how a formal methodology relates to traditional approaches. Since students may already be familiar with traditional approaches, they can relate to and learn the formal methods approach when they can relate it to concepts with which they are already familiar.

A number of formal methods incorporate tool support as part of the method itself although we have not directly used them in the classroom to see their effectiveness. Examples include OBJ [23], which offers executable subsets, Larch [24], which offers a theorem prover, and ZTC and *f*UZZ, which offer type-checking for Z.[13] CADiZ also offers a suite of tools for Z and supports refinement to Ada code. Many other tools are reported by Bowen and Hinchey [25].

Two tools that we quite often use for pretty-printing of Z specifications include the LaTeX "style" macros zed.sty and oz.sty. Both of these macros are freely available electronically via anonymous FTP. When using these macros, students no longer need to hassle with their editors to typeset special symbols and/or schema boxes. Every Z construct or symbol can be typed in through an ASCII terminal and either one of the above LaTeX macros can be used to generate beautiful Z output. For example, one can type in the following (space and line breaks are ignored):

[12]This is of significant importance since most formal methods use mathematical notations not available in traditional editors.

[13]ZTC is PC-based public domain software while *f*UZZ is a relatively inexpensive commercialized system that runs under UNIX.

6. CONCLUSIONS

It has been our aim to provide our students with software engineering concepts, technologies, and hands-on experience. Our students are exposed to the challenges of working in teams to develop a software system using CASE technology. Because the students are exposed to a CASE tool within a software engineering course, they see it as part of a "normal" environment for understanding, developing, and maintaining software, and, along the way, they experience the potentials and limitations of such tools. As students do their software engineering projects using a CASE tool, they learn to automate their software development in future and thus suffer less when they are exposed to these concepts in a real job. Since most software engineering courses are project-intensive, the use of an appropriate CASE tool permits students to concentrate on creative aspects of the project while allowing the tool to perform the repetitive and tedious tasks. For these reasons, we believe that it is important to integrate CASE tools within the beginning software engineering courses that are project-intensive.

The development of this approach has been a real learning experience for us, and we hope that this article will provide a few suggestions to those who are or will be teaching a software engineering course and may want to consider integrating CASE technology in their courses.

REFERENCES

[1] P. Ng and R. Yeh, Eds., *Modern Software Engineering*. New York: Van Nostrand Reinhold, 1990.

[2] R. Norman and J. Nunamaker, Jr., "CASE Productivity Perceptions of Software Engineering Professionals," *Communications of the ACM*, Vol. 32, No. 9, pp. 1102–1109, 1989.

[3] P. Mimno, "Survey of CASE Tools," in P. Ng and R. Yeh, Ed., *Modern Software Engineering*, pp. 323–350. New York: Van Nostrand Reinhold, 1990.

[4] M. Chen, J. Nunamaker, Jr., and E. Weber, "CASE: Present Status and Future Directions," *Data Base*, Vol. 20, No. 1, pp. 7–13, 1989.

[5] M. Brough, "Methods for CASE: A Generic Framework," in *Advance Information Systems Engineering*, LNCS 593, pp. 524–545. New York: Springer-Verlag, 1992.

[6] J. Clark and C. Hughes, "CASE Utilization Revisited," *Information Executive*, Vol. 4, No. 3, pp. 58–60, 1991.

[7] G. Forte and R. Norman, "A Self-Assessment by the Software Engineering Community," *Communications of the ACM*, Vol. 35, No. 4, pp. 28–32, 1992.

[8] H. Green, "Adapting CASE Tools for More Effective Learning," *Journal of Educational Technology Systems*, Vol. 19, No. 4, pp. 291–298, 1991.

[9] C. Hughes and J. Clark, "The Stages of CASE Usage," *Datamation*, Vol. 36, No. 2, pp. 41–44, 1990.