THE ROLE OF DEBUGGING WITHIN SOFTWARE ENGINEERING ENVIRONMENTS

Monika A.F. Müllerburg

Institut für Software-Technologie
Gesellschaft für Mathematik und Datenverarbeitung mbH Bonn
Schloss Birlinghoven, D-5205 St.Augustin 1, F.R. Germany

Abstract

Programming environments (PEs) support a single programmer developing small- to medium-scale programs, whereas software development support systems and software engineering environments (SE^2s) support whole project teams, developing large-scale software. There is no reason to believe that one and only one support system may exist. The usefulness of one or the other depends on the particular situation of software development.

Debugging is distinguished from testing and defined not only for removing bugs from programs (dynamic debugging) but also from documents describing the programs (static debugging).

The key problem of debugging is understanding the software. Debugging may be supported by static analysis tools and by interactive debugging systems which help both to understand the software better and to estimate the impacts of an intended change. Graphical representations are also very useful for better understanding system structures and for recognising faults and clashes faster. Tools may furthermore be used to report errors and corrections, and to maintain these reports.

In the context of PEs and SE^2s the tools supporting debugging are connected. Tools can be based on a uniform internal representation, allowing a uniform user interface. Tasks and corresponding tools can be tailored to each other, avoiding duplication of work. One task knows what the others have already done. One knows if certain types of errors have been prevented or removed, i.e. if static analysis tools prevent data flow errors during runtime. Tools and results of other tasks may be used, i.e. cross reference lists and test path reports. Similarities and differences of debugging in PEs and in SE^2s are discussed by some example systems. The discussion is concluded by demonstrating possible influences on future software development by personal computers, knowledge engineering, and predicative programming.

Keywords

Static debugging, dynamic debugging, validation, static analysis, testing, programming environment, software engineering environment, software development support system

---

This work is partially supported by the Commission of the European Communities.

Contents

1 INTRODUCTION

1.1 Characteristics of Software Development

Two very different kinds of software development may be distinguished:

- An individual programmer develops small- to medium-scale programs: Small-scale software can be understood, changes are traceable.
- A team of developers develops large-scale software systems: Large-scale software is in most cases long-life software, because of the costs of its development. It is generally not easy to understand. Changes may have unexpected impacts in the whole system, at least if the system is badly structured.

  Developing large-scale software needs more than programming: The problem to be solved must be carefully analysed and the solution to this problem, the software to be developed, must be carefully designed. So software development consists of requirements analysis and specification, software design and specification, and programming. The result and the intermediate results must be validated.

  Large-scale software is usually developed by teams, thus leading to communication problems and the necessity for project management, that is, planning, organizing and controlling it.

  Furthermore the wide application of computer systems has led to problems for users of such systems and of others who are affected by system usage. It is not enough to look only at the computer and its software - the surrounding system must also be considered. Generally the computer system is only a part of a surrounding organizational system. The introduction of a computer system into an organization may change the whole organization. The interests of humans in the process of system development and application should be taken into account already in the construction process - not as an afterthought. Therefore it seems to be necessary to anticipate impacts and to be able to model influences. The application related problems currently seem to be underestimated.

In summary:
Development of large-scale software includes the problems of development of small units. In addition requirements must be analysed, the system must be designed, integrated and maintained, and problems of communication within the development team and problems of project management must furthermore be solved. Which kind of software development is appropriate, depends on the aims of the development.

1.2 The Term Software Engineering Environment

The history of software engineering environments began with single tools and methods which have evolved into programming environments, then into development support systems, and finally into software engineering environments. In contrast to single tools, tools in a support system can be integrated in one sense or another. They are, for example, built around a data base or a programming language. A uniform internal representation of the objects allows a uniform user interface. What are the differences between these support systems?

- Programming environments provide tools and methods for a particular programming language. The task of programming is currently the best understood task of software development, and the tools are well elaborated. Examples of programming environments are INTERLISP, The Cornell Program Synthesizer, and Smalltalk.
- Software development support systems provide tools and methods for the technical tasks of software development, such as requirements analysis, software design, programming, and software validation. The tasks of software design and software validation were emphasized by research and development, thus leading to tools and methods, whereas tools and methods for requirements analysis and specification are rare. Examples of software development support systems are HDM and SDS.
- Software engineering environments provide tools and methods not only for the technical tasks of software development, but also for the other tasks, namely project management and application and its preparation. A software engineering environment provides tools, models of the product and of the production process, methods and concepts, as well as means for representation. Such a large system of course does not pay if experimental software is produced or if a single programmer develops small programs. Examples

for software engineering environments are SDEM/SDSS and SOFTORG.

In summary:
- Programming environments (PEs) support a single programmer developing small- to medium-scale programs, whereas
- Software development support systems and software engineering environments (SE^2s) support whole project teams, developing large-scale software.

There is no reason to believe that one and only one support system may exist. An SE^2 is an expensive system whereas a PE is not. The usefulness of one or the other depends on the particular situation of software development.

In [Hünk81, Wass81] various support systems are described. A bibliography may be found in [Haus81a] and detailed discussions of several aspects of PEs and SE^2s in [Haus81b, Haus82b, Prent81, SEN 81].

2 DEBUGGING

2.1 The Term Debugging

The term debugging is often mixed up with that of testing (or dynamic analysis). But the two activities should be carefully distinguished. The following steps are a sequence of testing (step 1), debugging (steps 2 and 3) and retesting (step 4):
(1) Show the existence of an error,
(2) locate exactly its original cause,
(3) propose a correction, look at its impacts and make the change, and finally
(4) see whether the error is really corrected and no new errors have been introduced.

The difference between testing and debugging is essential, at least in the context of long-life software. This distinction may be ignored when a developer is testing a module for the first time, or if he is the only person to be persuaded that the program behaves as intended, but it is essential for official test runs with planned test cases. Testing of long-life software needs careful documentation, because in its later development, information must be available about what has been tested and with what results. This includes information about test paths and (intermediate) execution outputs. It may furthermore be necessary to prove to a customer or to a court that the system has been tested to the best of the knowledge of the developers or testers. Such a proof can be based on documentation of test runs, consisting of information about test cases, test results and coverage grade. Since the behaviour of the program during test must be documented, it must not be changed interactively during a test run. The best way for executing test runs seems to be a kind of batch process.

In contradiction, the documentation of debugging has only to report errors and changes; the process of debugging need not be documented for later phases. The person doing the debugging looks for possible causes of the error, trying perhaps several ideas. The best support for debugging is an interactive tool, allowing one to observe and to change program behaviour.

In spite of the differences between testing and debugging, the tools for both activities are based on the same internal functions.

The term debugging is primarily used for the isolation and removal of faults, the existence of which has been shown by dynamic analysis. But bugs already exist in earlier phases of the software life cycle, many of which have hopefully been detected and removed before testing. So why not call these activities debugging also? This would lead to terms such as requirements debugging, design debugging, code debugging, and program debugging.

2.2 Problems of Debugging

In debugging fault recognition and repair is the job of humans. Therefore the key problem of debugging is understanding the software. Debugging can be supported by static analysis tools and interactive debugging systems, which help both to understand the software better and to estimate the impacts of an intended change. Graphical representations are also very useful for better understanding system structures and for recognising faults and clashes faster. Tools may furthermore be used to report errors and corrections, and to maintain these reports.

The problems of debugging and the possibilities for support are influenced by
- the type of software to be developed,
- the methods (especially design methods) used,
- the means for representing the software, and the validation techniques used during development.

The difficulties of debugging depend primarily on the size and complexity of the software to be developed. (This is true for debugging as it is for development and validation.) Small pieces of software are easier to understand than large ones, and the impact of changes can more easily be traced. Unfortunately software is often not well documented, and even if it is documented, the large amount of information about large software systems is difficult to handle, understand and trace.

Concepts such as modularization, information hiding, and abstract data types, make the location and removal of errors significantly easier. The system is easier to understand, the impacts of changes on other parts of the system are restricted, and the specifications of the interfaces allow them to be traced.

Representation means and corresponding tools influence the types of errors possible. Languages with strong type checking, such as PASCAL and ALGOL68, prevent errors resulting from inconsistent type assignments. A language like FORTRAN, not requiring variable definitions, allows spelling errors, such as sqpt instead of sqrt. Many errors could be found by the compiler or other tools, such as data flow analysers and interface checkers.

The types of errors, some of which may remain until dynamic analysis, depend also on the representation means and the tools used for validation during software development.

2.3 Support for Debugging

Success and efficiency of debugging depend heavily on the experience of the person doing the job. He or she may be supported by methods and tools. But the tools should not be used instead of thinking, they only should support thinking [Myer79]. That is, the tools should not be used for ad-hoc experiments, but only for systematical ones (think first). Tools may be used to support better understanding of the software, estimating the impacts of an intended change, and for making the change. Better understanding can be reached both by reports or queries on the software and previous test runs as well as by monitoring and controlling of program execution. Graphical representations are also very useful for better understanding system structures. Furthermore, tools may be used to report errors and corrections, and to maintain these reports.

Static and dynamic debugging may be distinguished:

- Static debugging is debugging of documents, such as requirements specification, design specification, and source code.
- Dynamic debugging is debugging of program execution.

Debugging of descriptions has problems and constraints similar to validation. Both depend on the representation means. Formal descriptions can be analysed automatically. Static analysis tools often indicate not a symptom of an error, but already its original source. There are even tools (today in most cases compiler functions) that try to correct a recognized error automatically. The question remains if this is the right direction to go. It seems to be more appropriate to restrict tools to making suggestions for corrections, but to leave the decision to the developer.

Static debugging by humans can be supported by similar means as validation by inspection, namely by compiler functions such as cross reference listers, and by special static analysis tools such as data flow analysers and interface checkers. A simple list of identifiers already allows recognition of spelling errors. Special tools can detect such errors as uninitialized variables, variables or output parameters not used, input parameters never used, mismatches between formal and actual parameters, and unreachable code segments. Even simple listings of external module references allow to check consistency between modules. Furthermore information systems are very helpful, because information about the software is available throughout its lifetime and query facilities allow analysing software without reading large listings or reports. The answer to a question "Where is function f referenced?" may be used to trace the implications of changing this function.

The support for static debugging is also useful for dynamic debugging. But dynamic debugging can be further supported. Print statements in the original program text have often been used in the past to get information about program behaviour during execution. Even with selected trace statements, much information is produced that is not needed. The last resort has often been a post mortem dump. Then special functions have been introduced into compilers, such as snapshot facilities and test assignments for tracing control and data flow.

Nowadays special support tools improve the efficiency of debugging. Debuggers allow monitoring and controlling program execution: Tracepoints may be defined, where selected variables are printed, changing of variable values may be watched, and reports and statistics on program behavior may be generated. Interactive debuggers allow defining breakpoints in the program, that is, locations, where execution is suspended and control is passed to the terminal. Execution may be slowed down from normal mode to a step-by-step mode. The variables may be displayed. They may even been changed, thus changing program behavior. But debuggers are still too frequently based on machine instructions rather than on the language level, thus hindering the person doing the debugging from thinking in his or her terms. A symbolic interactive debugging system makes debugging more efficient and more effective.

The features of a dynamic debugging support tool may be demonstrated by an example: ALADDIN is a very powerful and in some sense unusual interactive tool, supporting debugging and testing of assembly language programs [Fair79]. ALADDIN is not a traditional debugger: It allows specification of breakpoint assertions rather than breakpoint locations. The assertions describe relations between program states, such as "(variable) x < 0" or "x + y < z". If an assertion becomes false during program execution the execution is suspended and control passed to the terminal. The user can concentrate on the original reason or condition of an error, instead of tracing its symptoms along the control flow. The program is executed under control of ALADDIN. ALADDIN is controlled by a command language.

The example of ALADDIN may also demonstrate the similar functions of testing and debugging tools: The assertions may also be used for testing, then for specifying behavior conditions that must not be violated.

Combining ideas of verification, namely assertions, with the needs of testing and debugging (also realized for example in PET by dynamic assertions [Stuc75]) seems to be a very interesting direction for future software validation.

3 DEBUGGING IN THE CONTEXT OF SUPPORT SYSTEMS

In the context of programming and software engineering environments the tools supporting debugging are connected:

- The task (and corresponding) tools can be tailored to each other, avoiding for instance duplication of work. One task knows what the others have already done. One knows if certain types of errors have been prevented or removed, i.e. if static analysis tools prevent data flow errors during runtime.
- Tools and results of other tasks may be used, i.e. cross reference lists and test path reports.

Debugging is not emphasized in current SE²s, but it is in some programming environments. In most cases the tools used for analysis, documentation and testing may be used for debugging also. In the following two sections the support for debugging in PEs and in SE²s is analysed. The differentiation seems to be useful because of the different styles of software development in both environments.

3.1 Debugging in the Context of PEs

Modern programming environments (PEs) such as INTERLISP, the Cornell Program Synthesizer, and Smalltalk are highly interactive. They are based on a programming language and provide the classical tool sets, (consisting of editor, compiler or interpreter, linker, loader, runtime system and debugger) each of the tools knowing the language. So syntactical errors can be prevented on entry, and the user can interact with the system on the level of language concepts. The most important feature perhaps is, that there is no need for context switching, when switching for example between editing, compiling, testing, and debugging.

A PE, such as INTERLISP, allows the bookkeeping of changes (file package), as well as the tracing of impacts of intended changes (Masterscope). Instead of only printing analysis reports, such as cross reference listings, Masterscope maintains a data base, allowing query facilities. In the case of an uninterpretable situation, such as a misspelled and therefore unknown function name, another tool of INTERLISP, DWIM (Do What I Mean), attempts to guess what the user might have wanted and tries to correct the function name automatically.

A significant change in the programmer's situation, and thereby also in debugging, may already be seen today. It is caused by new hardware technology: The advent of powerful small computers allows working stations to be tailored to particular tasks, and high-resolution display terminals provide windowing and mouse techniques. This makes it easy for the user to manage several simultaneous tasks, such as editing, testing, and drawing pictures. The Cornell Program Synthesizer and Smalltalk demonstrate these changes:

- The Cornell Program Synthesizer provides programming, testing and debugging. Programming and testing or debugging can be interleaved. If an unexpanded piece of the program is entered, execution is suspended. It may be resumed after the expansion. The diagnostic features employ the advantages of windowing: The screen is separated into several windows. A flow trace shows the execution output of the program on one side of the screen, while the source code being executed is presented on the other. A pace and single step facility allows slowing the execution to a step-by-step mode on language statement level. A variable monitoring feature allows watching the values of selected variables.
- The developers of Smalltalk claim that "The machine that best matches Smalltalk's strength is a personal computer with a high resolution display, a keyboard, and a pointing device such as a mouse or graphic tablet." User commands are mostly given by pointing on the screen with the mouse. While switching from testing to debugging, the state may be frozen and execution at the suspended state may be resumed after debugging. The overlapping windows allow easily switching between editing, compiling, testing and debugging. For debugging, statements may be executed in a single step mode by pointing to them, or breakpoints may be set. Furthermore an undo operation allows going back to a previous state, thereby annulling the effects of the intermediate actions.

3.2 Debugging in the Context of SE²s

Examples for SE²s are APSE, SDS, SDEM/SDSS and SOFTORG. APSE is not a particular implementation, but a specification for a class of SE²s, intended to support software development for Ada. APSE is a layer- (or onion-) approach. The functions of the two lower layers are specified: the lowest layer, K(ernel)APSE, provides data base functions, communication functions and runtime functions; the next layer, M(inimal)APSE, defines a PE for Ada. On this base the highest layer, APSE, may be built, consisting of application and method systems. A discussion of debugging within APSE may be found in [Fair80].

Most of the current SE²s are backed by information systems, so that information about the software and the previous development tasks are available throughout its life time. The availability of this information is a significant advantage for debugging: The understanding of the software is supported and impacts of intended changes can more easily be traced. Furthermore in the context of SE²s information about previous validation and debugging tasks is available.

An important difference from PEs is that SE²s are generally based on a software life cycle model, consisting of a sequence of phases, that includes at least
(1) requirements analysis and software specification,
(2) software design and module specification,
(3) implementation,
(4) testing, and
(5) usage and maintenance.
Such a life cycle model implies that the only validation is a special phase of testing after implementation. But this is not the way software is or should be developed. Errors must be found as soon as possible, because otherwise they may have expensive consequences. Most of the errors are unfortunately introduced in the early phases of software development. So the constructive tasks must be accompanied by validation tasks. The result of each phase must be validated. This may of course lead to the recognition of a bug and to a debugging activity.

If the requirements are specified by formal means it can be checked automatically whether they are complete and consistent. But requirements are mostly described nowadays in natural language. This is true also in the context of SE²s. Validation and debugging of informal documents cannot be done by automated tools but only by humans. Possible support is provided by data dictonaries. The design of a system is more often described by formal means, thus allowing for automated checking of module interfaces in particular instances. Up to now the best support for validation and debugging exists for program execution and source code analysis. Most widely used today are debugging systems and compiler functions, such as data flow analysis and cross reference listers.

The support for debugging by SE²s may be demonstrated by the example of SOFTORG. The emphasis of SOFTORG is on documentation and validation, both based on a software data base. There are no special tools for debugging. SOFTORG will consist of the following subsystems:
- SOFSPEC is a specification system, based on the entity-relation model.
- SOFTCON will support program design and evaluation, based on the method of Parnas.
- SOFTGEN may be used to generate program frames from program designs.
- SOFTDOC is a system for static analysis, documenting program structures.
- SOFTEST is a test system.
- SOFTINT will integrate software components into a version of the software system and will maintain it.
- SOFTMAN will support project management.

SOFSPEC, SOFTDOC and SOFTEST are already in use.

Let us assume that we have to debug a PL/I- or COBOL-program, written by someone else. What support can we get by SOFTORG?

SOFTDOC, the system for static analysis and documentation, produces some reports, useful for identifying dangerous constructs, for evaluating and for reasoning about modifications. The information reported includes for example:
- a module tree, describing the call structure,
- a module HIPO diagram in which the referenced and set variables are listed for each section of the module,
- a module interface description, identifying the module relations, i.e. calls, SVCs, input- and output-statements,
- a pseudocode of the source, describing control flow,
- a table of data usage (for each module section) ordered according to predicates, inputs, and outputs, and
- a listing of unsafe constructs, such as constants within expressions and jumps into if statements.

Similar reports may be generated for programs, consisting of several modules, and for the whole system, consisting of programs.

On the other hand this information is stored in the data base, thus allowing query facilities in order to check where a variable or a function is used and where it is defined. So it is possible to trace the impacts of a change of this variable or function without analysing the large amount of data of reports and listings.

The test system SOFTEST instruments the testobject, generates a testbed, stores the collected information, and finally evaluates this information. SOFTEST generates output that is useful for debugging:
- Outputs of test preparation are:
  - A report on the test data, which have been generated on the base of assertions. These assertions specify the values of all data pertainung to interfaces, such as subroutine parameters and external data.
  - A report on the instrumentation of the testobject.
  - A report on the generated testbed.
- Output from the test run is a testlog, recording the test data, the calls of stubs and the simulation of interfaces, and errors pertaining to the assertions.
- Outputs from test run evaluation are:
  - A report on assertion violations.
  - A report on the test paths executed.

- A report on data flow (changes of variables).
- A report on test coverage (on the basis of branches).

The report on discrepancies between the real and the asserted values of variables is especially helpful not only for testing but also for debugging, because it helps to trace a faulty end result to its original source. Furthermore the information about previous test runs is available and may be used for debugging purposes.

4 OUTLOOK

There are some important influences on the further evolution of software development:
- personal computers,
- knowledge engineering, and
- predicative programming.

The influence of small personal computers has already been mentioned in the context of PEs (vide section 6). The connection of personal computers and distributed systems could allow the advantages of personal environments, such as programming environments, to be employed also for the development of large software systems by large teams. So future SE^2s may consist of personal working stations for designers, programmers and managers, connected within a net to each other and to a data base (central or distributed).

SE^2s may evolve into expert systems, i.e. into systems based on knowledge engineering. This would of course influence debugging also. Such expert systems may not only detect errors, but also propose modifications. Or they may - based on known errors - help the developer or debugger by suggesting possible sources of an error.

An example of the influence of knowledge engineering is PHENARETE, a programming environment for LISP [Wert82]. The system is based on pragmatic rules, describing the construction and correction of LISP constructs, and on specalists, describing syntax and semantics of the standard LISP functions. "The system can use its understanding of the program to detect errors in it, to eliminate them, and eventually to justify its proposed modifications."

The most significant change may come when knowledge engineering and functional, relational or predicative programming replace present software engineering techniques based on procedural programming. This is discussed in the context of the Fifth Generation Computer Systems [Moto82]. Today's programming would become obsolete, because system specifications could be executed. Knowledge engineering may be added to requirements analysis. For this kind of software development new methods and tools would be needed. Validation and debugging would focus on specification rather than on programs. Dynamic analysis and debugging would only check the remaining problems with respect to program behavior on a machine, i.e. overflows, caused by the fact that algebraic rules are not always valid on real machines.

Acknowledgements: The author wish to thank her colleagues H.L. Hausen for stimulating discussions and critical remarks and J.L. Darlington for his careful editorial reading of the text.

5 REFERENCES

Alfo81 Alford, M.W.: SDS : EXPERIENCE WITH THE SOFTWARE DEVELOPMENT SYSTEM. in [Hünk81]

Buxt81 Buxton, J.N.; Druffel, L.E.: REQUIREMENTS FOR AN ADA PROGRAMMING SUPPORT ENVIRONMENT: Rationale for Stoneman. in [Hünk81]

BYTE81 BYTE publication: A series on Smalltalk. BYTE: the small computer, Vol.6, No.8, August 1981

Davi78 Davis, C.G.; Vick, C.R.: THE SOFTWARE DEVELOPMENT SYSTEM: STATUS AND EVOLUTION. in IEEE COMPSAC 1978

Fair79 Fairley, R.E.: ALADDIN: AN ASSEMBLY LANGUAGE ASSERTION DRIVEN DEBUGGING INTERPRETER. in IEEE Transactions on Software Engineering, Vol. SE-5, No.4, July 1979, pp.426-428

Fair80 Fairley, R.E.: ADA DEBUGGING AND TESTING SUPPORT ENVIRONMENT. in ACM SIGPLAN Notices Vol.15, No.11, Nov.1980, pp.16-25

Haus81a Hausen, H.L.; Müllerburg, M.; Riddle, W.E.: SOFTWARE ENGINEERING ENVIRONMENTS : A BIBLIOGRAPHY. in [Hünk81]

Haus81b Hausen, H.L.; Müllerburg, M.: CONSPECTUS OF SOFTWARE ENGINEERING ENVIRONMENTS. in Proc. 5th Int. Conf. on Software Engineering, IEEE

Haus82a Hausen, H.L.; Müllerburg, M.: COMBINATION OF SOFTWARE VALIDATION TECHNIQUES (in German). in: Int. Congress for Data Processing and Information Technology, VDE-Verlag, Berlin, 1982

Haus82b Hausen, H.L.; Müllerburg, M.: SOFTWARE ENGINEERING ENVIRONMENTS: STATE OF THE ART, PROBLEMS AND PERSPECTIVES. in IEEE COMPSAC 1982

Hünk81 Hünke, H. (ed.): SOFTWARE ENGINEERING ENVIRONMENTS. Proceedings of the Symposium; North-Holland Pub. Co., Amsterdam, The Netherlands, 1981
John82 Johnson, M.S.: A SOFTWARE DEBUGGING GLOSSARY. in ACM SIGPLAN Notices Vol.17, No.2, February 1982, pp.53-70
Myer79 Myers, G.J.: THE ART OF SOFTWARE TESTING. John Wiley & Sons, London, 1979
Moto82 Moto-oka, T. (ed.): FIFTH GENERATION COMPUTER SYSTEMS. North Holland Pub. Co., Amsterdam, 1982
Mura81 Murakami, N.; Miyanari, I.; Yabuta, K.: SDEM AND SDSS: OVERALL APPROACH TO IMPROVEMENT OF THE SOFTWARE DEVELOPMENT ENVIRONMENT. in [Hünk81]
Phil82 Phillips, J.; Green, C.; Pressburger, T.: CHI: A SELF-DESCRIBED PROGRAMMING ENVIRONMENT. II Software Technology Seminar: Software Factory Experiences, 1982, SRI International, Menlo Park, CA, USA
Pren81 Prentice, D.: AN ANALYSIS OF SOFTWARE DEVELOPMENT ENVIRONMENTS. in Software Engineering Notes, Vol.6, No.5, Oct. 1981
SEN 81 Software Engineering Notes, Vol.6, No.4 (August 1981): NBS Workshop Report on Programming Environments.
Silv81 Silverberg, B.A.: AN OVERVIEW OF THE HIERARCHICAL DEVELOPMENT METHODOLOGY. in [Hünk81]
Snee82 Sneed, H.M.; Merey, A.: AUTOMATED SOFTWARE QUALITY ASSURANCE. in IEEE COMPSAC 1982
Stuc75 Stucki, L.G.; Foshee, G.L.: NEW ASSERTION CONCEPTS FOR SELF-METRIC SOFTWARE VALIDATION. in ACM SIGPLAN Notices, Vol.10, No.6, June 1975
Teit81a Teitelman, W; Masinter, L.: THE INTERLISP PROGRAMMING ENVIRONMENT. in COMPUTER, April 1981
Teit81b Teitelbaum, T.; Reps, T.; Horwitz, S.: THE WHY AND WHEREFORE OF THE CORNELL PROGRAM SYNTHESIZER. in ACM SIGPLAN Notices Vol.16, No.6, June 1981, pp.8-16
Wass81 Wasserman, A.I: TUTORIAL: SOFTWARE DEVELOPMENT ENVIRONMENTS. IEEE Cat. No. EHO 187-5
Wert82 Wertz, H.: STEREOTYPED PROGRAM DEBUGGING: AN AID FOR NOVICE PROGRAMMERS. in International Journal of Man-Machine Studies (1982) 16, pp.379-392

6 APPENDIX: CHARACTERIZATION OF THE SUPPORT SYSTEMS MENTIONED

In this appendix some systems supporting software development are characterized. Discussions of many such systems may be found in [Haus81b, Haus82b]. The systems characterized here are: APSE, CHI, Cornell Program Synthesizer, HDM, INTERLISP, PHENARETE, SDEM/SDSS, SDS, Smalltalk und SOFTORG.

System: APSE: Ada Programming Support Environment

Developers: Dep. of Defense et al., USA
Application: Ada programs
Status: Specification
References: [Buxt81]

Short Description:
APSE is a specification for en entire class of SE^2s, intended to support software development for Ada. APSE is a layer- (or onion-) approach. The functions of the two lower layers are specified: the lowest layer, K(ernel)APSE, provides data base functions, communication functions and runtime functions; the next layer, M(inimal)APSE, is the programming environment for Ada. On this base the highest layer, APSE, may be built, consisting of application and method systems.

System: CHI

Developers: Kestrel Institute, Palo Alto, CA, USA
Application: Experiments
Status: Partly in use
References: [Phil82]

Short Description:
CHI is a result of research on program synthesis and machine intelligence. It is an experimental knowledge-based programming environment, built around a very-high-level language, called V. This language is used for specifying the program as well as for specifying program knowledge.

System: Cornell Program Synthesizer

Developers: Dep. of Computer Science, Cornell University, Ithaca, NY
Application: Programming courses at the university.
Status: In use (about 3000 students at 4 universities); implemented on PDP11, UNIX.
References: [Teit81b]

Short Description:
The system is an interactive programming environment. It supports creating and syntax-directed editing, as well as executing and debugging programs, written in a dialect of PL/I. Syntactical errors are prevented on entry. Program development and testing may be interleaved, because programs are directly translated into interpretable code. Execution is suspended when an unexpanded part of the program is entered, it may be resumed after expansion. The system provides runtime debugging, depending on window technique. Diagnostic features are flow tracing (execution output), syntax-directed pace and single-step, and variable monitoring. Detected errors are announced, not corrected.

System: HDM: Hierarchical Development Methodology

Developers: SRI International, Menlo Park, CA, USA
Application: Operating systems, system programs
Status: In use
References: [Silv81]

Short Description:
HDM is an integrated set of languages and tools based on common software engineering concepts. The aspect of specification and verification is emphasized.

System: INTERLISP

Developers: XEROX Palo Alto Research Center, Palo Alto, CA, USA
Application: Experimental programming
Status: In use (about 300 users at 20 places, mostly universities); implemented on DEC PDP-10.
References: [Teit81a]

Short Description:
The system is an interactive programming environment for LISP programmers. The tools are implemented in LISP. The system is used for experimental rather than production programming, especially in the artificial intelligence community. A program evolves as a series of experiments, in which the results of each step suggests the direction of the next.

System: PHENARETE

Developers: C.N.R.S. and Dep. d'Informatique, University Paris, Paris, France
Application: LISP programs written by students
Status: In use (about 600 students; implemented in VLISP on DEC PDP-10
References: [Wert82]

Short Description:
The system understands and improves incompletely defined LISP programs. The system has a library with pragmatic rules, describing the construction and correction of general LISP constructs. It has also so called specialists, that describe the syntax and semantics of the standard LISP functions. The system can use its understanding of the program to detect errors, to eliminate them, and eventually to justify its proposed modifications.

System: SDEM/SDSS: Software Development Engineering Methodology/ Software Development Support System

Developers: Fujitsu Limited, Tokyo, Japan
Application: General purpose software
Status: In use
References: [Mura81]

Short Description:
SDEM/SDSS is management oriented. SDEM defines the production process (life cycle) and the use of tools. SDSS offers tools for the phases from program design to testing. For the early phases, requirements analysis and system design, PSL/PSA is adopted.

System: SDS: System Development System

Developers: BMDATC et.al., Huntsville, AL, USA
Application: Military embedded systems
Status: Partly in use
References: [Alfo81, Davi78]

Short Description:
SDS is an ambitious approach to the support of software production in military areas (especially: embedded systems). Many firms are involved under the head of the BMDATC (Ballistic Missile Defense Advanced Technology Center). Special emphasis is on requirements analysis and specification which is handled by SREM (of TRW). Software developed with SDS is applied within technical processes only; thus SDS does not handle the problems of software usage by humans.

System: Smalltalk

Developers: Learning Research Group, XEROX Palo Alto Research Center, Palo Alto, CA, USA
Application: Micro computers
Status: In use
References: [BYTE81]

Short Description:
The system supports programming on small (personal) computers. It employes window- and mouse-technique, and provides graphics, thus offering a friendly user interface. User commands are given by moving the cursor around the screen (mouse technique). It is built around the language Smalltalk, which is based on the model of communicating objects. The system has no operating system in the usual sense, because "an operating system is a collection of things that don't fit into a language. There shouldn't be one."

System: SOFTORG: Software Engineering System

Developers: Software Engineering Service (SES), München, F.R. Germany and
SZAMALK, Budapest, Hungary
Application: General purpose software, written in PL/I, COBOL or Assembler
Status: Partly in use
implemented on IBM 370, SIEMENS
References: [Snee82]

Short Description:
SOFTORG (formerly SOFTING) is a phase oriented development support system. It will consist of several sub systems, each supporting a phase of software development. Emphasized are documentation and validation. Explicitly not covered are requirements analysis and maintenance. The former because of still missing practical solutions, the latter because of being supported by the other tools. The sub systems are:
- a specification system, based on the entity relation model,
- a design system, based on the method of Parnas,
- a system generating program frames from program design,
- a system for analysing program source code and documenting program structures,
- a test system,
- an integration system, and
- a system supporting project management.