Deploying Artificial Intelligence Techniques In Software Engineering

Jonathan Onowakpo Goddey Ebbah
Department of Computer Science
University of Ibadan
Ibadan, Nigeria

Received March 8, 2002 ***Accepted March 18, 2002***

ABSTRACT

Software development is a very complex process that, at present, is primarily a human activity. Programming, in software development, requires the use of different types of knowledge: about the problem domain and the programming domain. It also requires many different steps in combining these types of knowledge into one final solution. This paper intends to review the techniques developed in artificial intelligence (AI) from the standpoint of their application in software engineering. In particular, it focuses on techniques developed (or that are being developed) in artificial intelligence that can be deployed in solving problems associated with software engineering processes.

I. INTRODUCTION

Three fundamental questions are likely to run through your mind as you read this paper.

- What is software engineering?
- What is artificial intelligence (AI)?
- What are the aspects of software engineering that makes it amenable to concepts and techniques in artificial intelligence?

Software engineering is the act of adopting engineering principles in software development. In this act, the principles of analysis and synthesis are observed [1-4]. Analysis is the process of breaking something into pieces or components with a view to understanding the individual components [5]. Synthesis, the reverse of analysis, is the putting together of a large structure from small building blocks [5]. Thus, any problem-solving technique must have two parts: analyzing the problem to determine its nature and then synthesizing a solution based on the analysis. The diagrams below illustrate these two concepts (Figure 1).

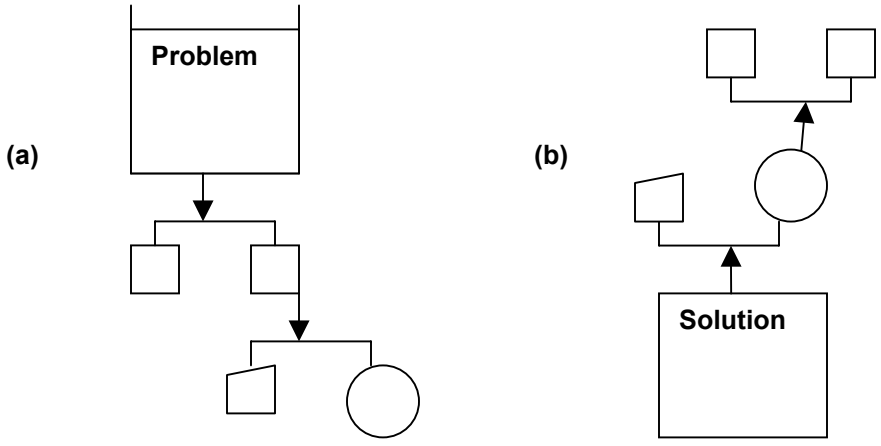

Figure 1. Any problem-solving technique must have two parts: (a) An analysis of the problem to determine its nature, and (b) a synthesis of a solution based on the analysis.

Artificial intelligence on the other hand is a domain of computer science that attempts to make machines perform tasks that hitherto done by human beings [4]. Its focus is on creating machines that can engage in behaviors that humans consider intelligent. This ability to create intelligent machines has intrigued humans since ancient times, and today with the advent of computers and after more than fifty years of research into AI programming techniques, the dream of smart machines has become a reality. This domain of computer science uses a combination of knowledge from different domains of study. Such domains include psychology, philosophy, physiology and computer science. Human beings have been writing software for a long time and no machine can yet do better. Therefore, software engineering is an appropriate topic for AI application and techniques [1-3].

II. SOFTWARE ENGINEERING AND ARTIFICIAL INTELLIGENCE.

Software engineering requires two kinds of knowledge: programming knowledge (e.g. data structure construction, control structure, programming language syntax, and how to combine and choose them); and domain knowledge (e.g. concepts, theories and equations characterizing the particular domain). Each domain has its own terms and properties, which must be linked to the programming language in which the software is being developed. Thus, it requires that there be a conversion from one knowledge set to another (e.g. from domain to programming language and vice versa). This obviously requires some method of conversion which requires some techniques in AI. The bottom line in this conversion is that both languages are formal thus making it particularly suitable for AI. There is a similar mapping taking place when going from the requirements specification stage to the design stage, though the initial specification is an informal description of the problem. Thus, the conversion is from an informal statement to a formal one.

Software engineering also involves modeling, analysis, and the generation of alternate designs, which entails problem decomposition [6]. Techniques are needed to analyze and evaluate these decompositions in order to choose one design over the others. The kind of decision making needed at this stage is amendable to AI.

First, in the translation from the informal description of requirements into formal descriptions, natural language processing, a sub field in AI, may be used. The task is not so much of direct translation from the natural language to a formal one, but rather to provide help to users since there is no completely automated process that can convert natural language representations to a formal form. However, it can generate questions and elaborations of what the user is writing in natural language. It is the response to these questions that eventuality forms what the system generates. In database management systems, this technique has been adopted in that the naïve user now has a natural English view through which he can write query statements in pure natural English. This sub field of AI encompasses:

- Voice Communication: for person-to-computer interaction through vocal inputs with microphone;
- Speech Synthesis: for computer-to-person interaction through sound generated by a synthesizer;
- Language Comprehension: for person-computer interaction through symbols such as text and words.

Adopting this feature in software engineering can go a long way to solve the problem of extensive documentation.

The traditional view of software development process begins at the requirements specification and ends at testing the software. At each of these stages, different kinds of knowledge (design knowledge at design stage and programming and domain knowledge at the coding stage) are required. At each of the two stages: design and coding, exist a cycle: error recognition and correction. Experience shows that errors can occur at any stage of development (Figure 2). Errors due to coding may occur because of faulty design. Such errors are usually expensive to correct.

Knowledge-based techniques in AI can be used to modify this traditional approach. One strategy is to automatically translate from the requirements specification to program testing. Taking the whole problem as a continuum and making

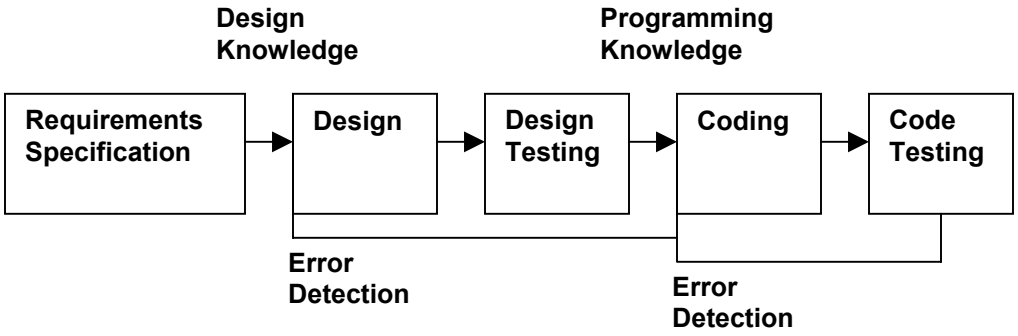

Figure 2: The traditional software development process.

changes at the specification stage accomplishes this. The user only provides the requirements and the machine does the translation into codes. This technique is advantageous in that:

- If done correctly, it reduces cost.
- Errors detected in coding will be isolated in the requirements stage.
- Changes need be made only at the requirements stage.

This isolation however is difficult to achieve and has not proven very efficient for large programs.

To generate a system in software engineering, one might find another system with similar requirements. The design of the first system would then be modified until it becomes a reasonable design for the given problem. Although this process looks feasible, it has not been demonstrated in software engineering to any great extent [4-7]. On the other hand, AI offers a technique called constraint propagation technique which gives rise to a truth maintenance system that can be used for planning. Decisions are made at one place of the planning process and carried to the next level. Then, using analogical reasoning—an offshoot of logical reasoning and problem solving—is used to compare a problem for which a solution is known with another problem to be solved.

A basic problem of software engineering is the long delay between the requirements specification and product delivery [7]. This long development cycle causes requirements to change before product arrival. It therefore becomes imperative to have an automated system which can take the requirements and drive all the way through multiple levels of translation to codes.

In addition, there is the problem of phase independence of requirements, design and codes. Phase independence means that any decision made at one level becomes fixed for the next level. Thus, the coding team is forced to recode whenever there is change in design. However, the AI technique that handles this problem is automated programming which results in reusable code. Thus, when a change is made in the design, that part of the design that does not change remains unaffected. Thus, automated tools for system redesign and reconfiguration resulting from a change in the requirements will serve a useful purpose. This technique, of course, cannot work without employing a constraint propagation technique.

III. TECHNIQUES AND TOOLS OF AUTOMATED PROGRAMMING

Because of the evolutionary nature of software products, by the time coding is completed, requirements would have changed (because of the long processes and stages of development required in software engineering): a situation that results in delay between requirement specification and product delivery. There is therefore a need for design by experimentation, the feasibility of which lies in automated programming. Some of the techniques and tools that have been successfully demonstrated in automated programming environments include:

- Language Feature: this technique adopts the concept of late binding (i.e. making data structures very flexible). In late binding, data structures are not finalized into particular implementation structures. Thus, quick prototypes are created which result in efficient codes that can be easily changed. Another important language feature is the packaging of data and procedures together in an object, thus giving rise to object-oriented programming: a notion that has been found useful in environments where codes, data structures and concepts are constantly changing. Lisp provides these facilities.
- Meta Programming: this concept is developed in natural language processing (a sub field of AI). It uses automated parser generators and interpreters to generate executable lisp codes. Its use lies in the modeling of transition sequences, user interfaces and data transformations.
- Program Browsers: these look at different portions of a code that are still being developed or analyzed, possibly to make changes, thus obviating the need for an ordinary text editor. The browser understands the structures and declarations of the program and can focus on the portion of the program that is of interest.
- Automated Data Structuring: this means going from a high-level specification of data structures to a particular implementation structure.

When systematic changes need to be made throughout a code, it is more efficient and controllable to do it through another program (*i.e.*, program update manager) than through a manual txt editor. For instance, a change in program X may be required whenever h is being updated by b-1 under the condition that b is less than C. Assume that a program W makes a systematic change in all such places. If another program makes a change in W, then any program changed by W also must be updated. Thus, program update managers propagate changes. Because of this ability, program update managers are useful when prototypes need to be developed quickly.

IV. CONCLUSION

Although a very formal theory has been discussed, automated programming still has its own limitations and is sometimes impractical. With the concepts well illustrated, the problem is in the synthesis of big programs. Thus, special cases will need to be identified where these processes are practical [5].

Besides, the basic concepts for automating program development must be language independent. One of the enhanced language features is object-oriented programming which shortens the requirements-to-code cycle. This object-oriented concept has now been fully developed and is been embedded in a number of object-oriented languages such as C++, Visual C++. Data and procedure packaging (or data binding), another language feature, has also been implanted in visual basic, C++, Visual dbase, Delphi, among others.

ACKNOWLEDGEMENTS

I would like to appreciate Dr. B.A Oluwade for his help with this paper. His encouragement and long hours with me in this effort are greatly appreciated. My thanks also go to Dr. B.A Akinkunmi for his foresight and involvement in my research interest.

REFERENCES

1. Stuart Russell and Peter Norvig, *Artificial Intelligence: A Modern Approach* (Prentice Hall Publishers, Upper Saddle River, New Jersey, USA) 1994.
2. Ian Sommerville, *Software Engineering (6th Edn.)* (Addison Wesley Publishers, New York, New York, USA) 2000.
3. Roger S. Pressman, *Software Engineering: A Beginner's Guide* (McGraw Hill Higher Education Publishers, New York, New York, USA) 1988.
4. Seth Hock, *Computers and Computing* (Houghton Mifflin College Publishers, Boston, Massachusetts, USA) 1989.

5. M.L. Emrich, A. Robert Sadlowe, and F. Lloyd Arrowood (Editors), *Expert Systems And Advanced Data Processing: Proceedings of the conference on Expert Systems Technology the ADP Environment* (Elsevier-North Holland, New York, New York, USA) 1988.
6. Shari Lawrence Pfleeger, *Software Engineering: theory and Practice* (Prentice Hall Publishers, Upper Saddle River, New Jersey, USA) 1998.
7. C.S. French, *Data Processing and Information Technology (10th edition)*, (Letts Educational Publishers, London, United Kingdom) 1996.