Introduction

In contrast to the recent research interest in development environments which support programming-in-the-large and programming-in-the-many, this report is interested in environments which can maximize the productivity of the individual developer.

The goal of this report is to identify the most useful techniques for the machine-assisted generation of software programs in arbitrary domains, under the guidance of a single competent individual who is assumed to have a solid understanding of the program requirements. In other words, this report attempts to answer the question

> What techniques hold the most promise for maximizing the productivity of the knowledgeable, motivated, *individual* software developer?

To answer the question, this report surveys machine-assisted software development approaches which are actively concerned with the generation and manipulation of compilable source code. Software development approaches which deal exclusively with early lifecycle activities such as requirements specification have been excluded, as have environments which exclusively emphasize project management activities. All environments discussed in this report are capable of producing source code.

The body of the report contains four major sections. The first section introduces a set of evaluation criteria to serve as a basis for comparison, the second section discusses candidate environments, the third develops an idealized environment for the individual developer, and the fourth section summarizes the results, draws conclusions, and suggests future research directions.

Chapter 1

Evaluation Dimensions

In a survey of any research area, a set of appropriate domain characteristics must be chosen to serve as a framework of discourse. Usually the chosen characteristics are selected to answer or investigate a corresponding set of interesting questions.

In this paper, interesting questions for machine assisted software development approaches can be roughly divided into three categories concerned with (1) the conceptual approach, (2) its realistic implementation, and (3) the human interface to the final system. A list of such questions is shown below.

- What is the conceptual software development model used by the research approach?
- What languages and tools characterize the approach?
- Over what domain of situations can the method be applied?
- How much software has actually been produced by implementations of the approach?
- How complex are the systems which implement the approach?
- What range and types of problems can realistic implementations of the approach address?
- What are the hardware and software requirements of a realistic implementation?
- Are realistic implementations economically viable?
- How complex is the human interface?
- What educational levels are required of human users?
- How usable and extensible is the system?

The following sections of this chapter provide detailed explanations of the evaluation criteria suggested by the questions above.

1.1 The Research Approach

1.1.1 Languages Used

The languages used by a system play a key role in the complexity, power, and human interface of that system. Three major groups of languages can be distinguished according to the type of programming activities which they support (traditional, experimental, or automatic programming).

Traditional programming languages such as Assembler, Fortran, Cobol, ADA, and C are clearly the most prevalent type of language in the programming industry. These languages are characterized by a wide and general domain of application, and, as a group, have the ability to represent very low level detail through reasonably high level abstractions. If anything, these languages are shorter on the high-level abstraction end than on the low level "bit manipulation" end of the programming language spectrum. Due to the relative simplicity and imperative nature of these languages, they are relatively easy to understand and work with, and are computationally efficient in their compiled forms.

Modern high level languages such as Lisp, Smalltalk, Prolog, and other symbolic languages are generally more capable of representing high level abstractions, but are correspondingly more difficult to use for low level manipulations. These languages, because they are less procedurally oriented than traditional languages, require their users to think in a more abstract manner; this, in turn, often requires more education of the user. Because they are normally interpreted rather than compiled, most of these languages are somewhat less efficient than the traditional languages, which have enjoyed a long history of optimization efforts. This is not to say that all such languages are inefficient; many modern dialects of Lisp are standardly compiled into more efficient forms, and in at least one application field (editor construction [28]), can be significantly more efficient than traditional procedural languages. Many languages in this group are interpreted in *environments*, a feature which adds a significant amount flexibility and productivity to the development process.

Very high level abstraction and specification languages such as those used in the automatic programming approach to software development can represent concepts which are far beyond the limits of traditional and modern high level languages (eg. historical or descriptive reference). They are, in fact, very powerful expressive tools. Unfortunately, the formality of such languages appears to preclude notations which are easy to read even by those who designed them; as a consequence, systems based on these languages are often forced to use a mechanical paraphraser to translate the language back into something understandable. Moreover, the languages are *so* high level that it is very difficult to translate them into lower level languages which are within the range of automatic compilation systems. As a result, most of these languages are limited to some form of (usually inefficient) simulation or symbolic interpretation in their normal environments. The likelihood of widespread use of these languages in their current forms is slight for these reasons.

Wide spectrum languages are languages which span a wide spectrum of abstraction [22] [1] over the entire software lifecycle. This means that while they support the very high level,

abstract constructs needed for program specification applications in the early lifecycle, they can also be transformed into low level compilable constructs in later lifecycle phases. While the uniform syntax offered by these languages across the lifecycle is a boon to the tools which manipulate them, the languages are still difficult to construct, read, and understand. They are usually only found in research applications and research automatic programming systems.

Narrow spectrum languages, in contrast, have a limited range of abstraction expressiveness, and are usually designed for a specific problem domain [22] or specific lifecycle phase (eg. specification, design, or execution). Traditional languages such as Assembler, Fortran, and Pascal are clearly narrow spectrum languages because of their narrow lifecycle application and range of abstraction expressiveness.

1.1.2 Tools Used

The tools which characterize a research approach play an important role in the productivity which can be offered by the final implementation. Two general classes of tools can be distinguished based on their reason for existence: tools for managing complexity and tools for improving production. Classifying the tools of a research approach into these two categories can give a better understanding of the conceptual and clerical complexities of the approach.

Tools for managing complexity tend to characterize research systems, since they are generally concerned more with prototyping concepts and ideas than with achieving maximum computational efficiency. Some examples of these tools are interpreters of special languages, paraphrasers for complex specification languages, and inference engines which drive program transformations.

Tools for improving production tend to characterize traditional software development environments. Optimizing compilers, extensible editors, 'make' utilities, advanced debuggers, code libraries, version control systems, and application generators all aim at improving the efficiency, stability, and speed of software production. This is typically the largest of the two classes of tools, even in research environments.

1.1.3 Domain Limitations

Many special purpose research systems based on narrow application domains will fail when the bounds of the domain are exceeded by any significant amount. Such systems are called *brittle* because of their sudden (and often total) breakdown at their domain boundaries. It is an important challenge in software research to develop an approach which exhibits not only a wide domain of application, but which also degrades gracefully as its domain boundaries are exceeded. Unfortunately, such an approach has not yet been produced.

Experience tends to indicate that the productivity support which tools can offer is largely dependent upon the formality, concreteness, and stability of the conceptual models on which they are based. For example, language compilers offer huge productivity gains in the well understood and *formal* translation activity, yet tools which manipulate natural languages

are still in their infancy because of the *informality* of natural languages. Similarly, toolsets which offer full support across the whole software development lifecycle have not yet been developed because of the myriad different (and equally valid) ways to develop software; the lack of a formal and well understood development model is clearly a limiting factor in this regard.

It seems reasonable to conjecture that the only good research candidates for producing non-brittle development systems which will gracefully degrade near their boundaries are those approaches which include *both* a wide spectrum conceptual software development model and solid tool support for that model.

1.1.4 Lifecycle Support Offered

The practical lifecycle support offered by an approach is arguably the most important factor in the eventual acceptance or rejection of the approach by the software development community. In particular, supports for *programming-in-the-large* and *programming-in-the-many* activities are of increasing interest as both industry and research develop larger and more complex software systems. Considering a candidate approach in the context of *early*, *middle*, and *late lifecycle support* can help to give a more balanced understanding of its capabilities.

Two examples of lifecycle support beyond the traditional coding phase include source code version and access control systems. Experience has shown that such supports are of significant utility, as are product configuration control systems. Interest is also developing in the project management activities of task identification and planning systems.

1.1.5 Opportunity for Automated Support

The opportunity for automated support offered by a software development approach can be defined as the portion of *its own* conceptual lifecycle model which is easily supported by automated tools. In general, approaches which support more automation are of more interest because of their implied lower costs and higher productivity.

One large research problem concerning conceptual development models hinges around the human component of a development system. Some approaches concentrate on supporting and optimizing existing the existing human input, while others try to remove as much of the human component as possible in hopes of increasing the speed and quality of production.

For example, the traditional approach seeks to improve the productivity of the traditional human component of the system by providing better tools for specific jobs. Conventional development tools offer some form of machine assistance for a relatively high portion of the traditional lifecycle.

Other research approaches attempt to reduce the human component in the system by offering new conceptual development models such as those associated with application generators, automatic programming systems, and intelligent programming assistants. Unfortunately, these new approaches have had limited success in removing the human component from the development process. Research indicates that the human effort is usually shifted

to different activities rather than being totally removed from the system. For example, the human effort in automatic programming systems tends to be focussed on debugging the abstract specification and transformation sequence of a program rather than on the traditional activities of designing, coding, and debugging its implementation.

While the list of *possible* productivity opportunities is a long one, research has demonstrated only a few of them. The application of wide spectrum automated support to *any* software lifecycle is not a simple problem.

1.1.6 Demonstrated Production Ability

The amount of real software produced by a research approach is one indication of its maturity and viability as a candidate for a good machine assisted software development system. Many research approaches, often for a variety of legitimate reasons, have not sponsored any significant amount of realistic software development. However, several have made progress in this regard -- notably those which focus on augmenting or optimizing conventional development models [13] [24]. The most successful research approaches, of course, achieve widespread use in their field of application [30].

Research approaches can generally be divided into three groups based upon the amount of software produced under the approach:

1. *Experimental systems* usually deal with a few test cases over a limited period of chronological time (eg. a PhD residency term). Because these systems are often in their infancy and the rate of learning about the problem domain and the approach is so high, test cases are often abandoned once the learning potential of the problem has been exhausted. In other instances, test cases turn out to be more difficult than expected and funding constraints terminate the line of research. For whatever reason, systems in this group do not generate enough interest to be developed into significant trial production systems, even though they might sustain the interests of researchers for many years.

2. *Trial production systems* are major systems produced by approaches which have enough merit to be taken beyond experimental problems into a multi-user development testbed. In such environments, a significant group of researchers attempt to implement meaningful pieces of software using the approach — typically improved versions of the experimental implementation upon which the approach itself is based. Examples of such systems are [1] and [8]. There are a wealth of research systems in this class which have been developed enough to be used in academic computer science environments for teaching purposes.

3. *Full production systems* are full implementations of mature research concepts which enjoy widespread use. Exploratory Lisp environments [15] [33] are an example of a research approach in this stage, as are structured oriented environments such as [24] and [13]. Many traditional development tools can also be classified into this group (procedural languages, editors, compilers, interpreters, debuggers).

1.2 The Implementation

Even though the conceptual basis of a research approach may be straightforward, the corresponding implementation might not be. A realistic implementation is often impractical because it requires too many computing resources, is economically unfeasible, or is simply out of reach of current technology. It follows that by examining a research approach in the light of its implementation characteristics, one can gain perspective on the maturity of the associated implementation technologies and on the potential impact of the research approach as a significant productivity force in the near future.

As with the evaluation dimensions concerning the research approach, the criteria here were chosen to investigate the following implementation questions:

- Does the implementation add more or less complexity to the problem domain?
- Can realistic implementations handle significant domain problems, or are they limited to smaller research test cases? Does the approach scale up well to realistic problems?
- What magnitude of computational resources does the implementation require?
- Can the implementation be made generally portable across the machines which characterize current hardware technology?
- Are implementations of the research approach economically viable in the current or in the near term future software development industry?

1.2.1 Complexity Reduction

Desirable implementations of new research approaches should reduce the external, visible complexity of the problem domain by providing better conceptual abstractions and by taking increased responsibility for managing problem details. If possible, this should be done without adding intractable amounts of implementation complexity inside the user interface. It generally holds true that more complex implementations are more expensive, less popular, and thus less effective in industry than they might otherwise be.

Tools associated with traditional development methods and languages have usually achieved reasonable levels of external and internal complexity. The user interfaces to these tools are usually simple (command lines with control arguments), and the internal complexity is such that many tool manufacturers can find personnel with sufficient expertise to implement the tools. A standard undergraduate education in computer science is usually a sufficient base from which to start implementing such tools, although development experience obviously plays a large role in the efficiency of the final product.

In contrast, tools associated with new development models are often very complex both externally and internally. For example, the formal languages used for new development approaches such as automatic programming are highly complex externally, often hard to

read, and difficult to automatically translate (even with human input) into a form which can be understood by current compiler technology.

It seems unlikely that implementations of radically new software development models will achieve favorable levels of internal and external complexity in the near future.

1.2.2 Problem Size Capacity

The maximum problem size which can be handled by an implementation can significantly affect its reception in the software community. If an implementation is unable to handle realistic problems of the day, or if it can only solve problems which already have better solutions, then it is unlikely to gain wide acceptance because it offers no real value beyond other methods. Conversely, if an implementation clearly demonstrates a mix of useful capabilities which cannot be found elsewhere, it will likely gain popularity at an increased rate.

One frequent limitation observed in research is that of problem size. While a research approach may not have any conceptual limits on problem size, an implementation which can handle realistic problems may not be possible for a variety of reasons.

While many implementations can demonstrate the ability to solve reasonable research problems, many of them cannot easily be "scaled up" to handle realistic industrial problems. As a consequence, such implementations can become less interesting to both publication oriented researchers and economically oriented industry users as time goes by.

1.2.3 Computational Efficacy and Hardware Availability

Reasonable implementations of a development approach must be widely available in order to exert a significant impact on the software development industry. This usually requires that both the research approach and its implementation be computationally efficacious enough to handle realistic problems on realistic computing hardware.

It seems reasonable to conjecture that approaches and implementations based on widely available "workstation class" minicomputers and microcomputers will enjoy more success than those which require extensive (and expensive) computing resources.

1.2.4 Economic Viability

The economic viability of a research approach and its implementation is essentially based on a tradeoff between problem solving capacity and cost of implementation. A research approach might clearly demonstrate research success and productivity gains, but if its implementation requires costs in excess of those gains, the economic viability of the approach will be low until a more favorable balance is achieved. This tradeoff has acted against many research implementations in the past, but will no doubt be mitigated by further improvements in hardware and software technology.

1.3 The Human Interface

1.3.1 Complexity Reduction

The human interface offered by a research approach can significantly affect the popularity of the implementation when more favorable competing implementations exist. However, within relatively narrow limitations, most implementations which share a common development model will probably also share similar user interfaces.

The complexity of human interface required by the approach itself is the fundamental issue, not the complexity added by the implementation. For example, if the approach itself requires significant amounts of complex information from the user, the human interface is considered to be complex, even if the implementation uses user-friendly pop-up menus and a graphical interface. Other things being equal, the favored interface is one which is complex enough to address the full problem while still being easy to use.

A comparison of traditional development approaches and tools with current research approaches and their tools indicates that the complexity of the human interfaces in the newer research approaches is quite high. In particular, automatic programming research [1] [16] [10] indicates that approaches which use a very high level abstraction language have a significant amount of complexity in both the visible human interface as well as in the underlying implementation. The underlying implementation of knowledge based approaches [31] [32] [25] can also be very complex.

It is clear that the development of new software production technologies which can present low-complexity human interfaces is a very difficult problem.

1.3.2 User Education Required

The educational level required by a software development approach is a key factor in its widespread acceptance in the software community. Since user training is one of the major costs associated with the introduction of new technology, research approaches which require extensive educational upgrades of their users will not gain acceptance quickly. Only if the approach can be supported by standard levels of education will it be a candidate for rapid widespread use.

1.3.3 Speed, Usability, Extensibility

The more common aspects of a human interface also have an effect on the general acceptance of a research approach. The speed of operations in the new implementation (as compared with equivalent traditional techniques) is of great significance to the user, and has a heavy bearing on the economic viability of the approach. Similarly, the complexity, utility and expressiveness of the notation used by the implementation may also affect system acceptance rates.

Some past research [28] has indicated that the *extensibility* of a system plays a critical role in the usefulness of a development tool. If users can easily extend an implementation

to address their specific local interests or problems, the implementation will enjoy a wider audience than other non-extensible versions.

One problem with extensible systems which are based on a formal approach is that it is difficult to extend the formal foundation to user-written extensions [8]. However, since full formal support for even static development systems is still not yet a realistic expectation, this problem does not count heavily against the notion of extensibility.

1.4 Summary

The evaluation dimensions above have been chosen to investigate the particular software development models and systems discussed in the second part of this paper, with emphasis on determining the utility of the approaches in actually lowering the cost or improving the quality of software produced with these systems.

Of course, the chosen criteria are only one way of evaluating the various approaches to machine assisted software development; other dimensions may be more or less important from different viewpoints.

Chapter 2

Research Approaches and Implemented Systems

The following sections are organized according to a research approach taxonomy similar to that used in [9], but with additions and deletions considered relevant for the purpose of this paper (systems which actually generate code). In particular, *method based environments* oriented purely toward early lifecycle or project management activities have been omitted from this paper. See [9] for an overview of research in this area.

While the systems mentioned in the following section have been associated with the approach which best characterizes them, it is important to note that some of them exhibit characteristics from more than one conceptual approach. It follows that the classifications below are somewhat arbitrary and not mutually exclusive. The following approaches are discussed in the remainder of this chapter:

- Traditional General Methods
- Application Generators
- Structure Oriented Environments
- Exploratory Programming Systems
- Automatic Programming Systems
- Knowledge Based Systems
- Software Reuse Systems

2.1 Traditional General Methods

Traditional software development approaches usually use some form of the Waterfall [4] or Spiral [3] lifecycle development models. In particular, conventional lifecycle models almost always contain the development activities of software design, implementation, and testing.

Opportunities for automated support of this model usually involve specialized tools for particular activities and lifecycle phases, leading to reasonably effective wide spectrum support of the whole lifecycle, and to very competent support for the core phases of implementation and debugging. Fully integrated wide spectrum supports for this lifecycle model have not yet been solidly conceptualized or implemented.

The conventional development model, though often chastised in the literature, is a highly accepted and heavily used software development approach.

2.1.1 The Research Approach

- *Languages:* This approach uses traditional procedural programming languages such as Fortran, Cobol, Assembler, Pascal, PL/1, C, and Ada. These languages are generally considered to be narrow spectrum languages because of their limited abstraction mechanisms (as opposed to a special domain language such as used by the Draco system [22].)

- *Tools:* This approach has a wealth of specific, production oriented tools for each step of the core lifecycle. The tools, as a group, successfully reduce the visible complexity of the problem while maintaining reasonable amounts of complexity in the internal implementation.

- *Domain Limitations:* The traditional software development method has no significant domain restrictions. It is clearly not the method of choice in some domains (logic programming, exploratory Lisp programming), but has been used to implement software systems in the widest variety of domains of any known approach. The conventional development approach is not brittle.

- *Lifecycle Support Offered:* The conventional lifecycle model, because it treats each step as a separate entity, allows each lifecycle activity to be supported by a specific individual tool. As a consequence, virtually every step in the conventional lifecycle has access to machine assisted support in the form of software tools. In addition, tools which support particular project management models such as source code access and version controls, configuration management, and activities management are also available.

- *Opportunity for Automated Support:* Several steps in the traditional development method are already heavily optimized and automated (eg. compilation), and so offer little opportunity for productivity gains. As a consequence, the less optimized areas of the traditional lifecycle have been receiving more attention in the last decade. In particular, early lifecycle activities (requirements specification and validation, prototyping, system design) are now being addressed by a wide variety of so-called CASE tools. The late lifecycle maintenance activity is still relatively unsupported.

- *Completed Applications:* The conventional development method has been used to create applications of virtually every size.

2.1.2 The Implementation

The software implementation of the traditional development approach is best characterized by the many tools which address the various specific steps in the development process.

The core set of tools found in most implementations includes an editor, compiler, linker, debugger, a code library, a program building utility (such as make), and a batch file mechanism for automating repetitive groups of task oriented commands. While seemingly very basic in today's world of high resolution displays, pull down menus, powerful workstations, and modern communications, this basic tool set is still the engine of the software world. Recent high-tech additions to the basic toolset, despite their glossy appeal, have added little productivity in comparison to the gains which were recorded by the development of external compilation, linkable code libraries, and third generation languages.

- *Complexity Reduction:* Most traditional tools have been able to achieve reasonable levels of external (problem) and internal (implementation) complexity. Their success may be related to the fact that each tool only addresses a small and independent portion of the lifecycle.

- *Problem Size Capacity:* Traditional tools are generally competent at handling realistic problems within their intended problem area. Even though poor implementations have problems dealing with abnormally large files and data sets, as a group traditional tools are quite competent at handling common industry problems.

- *Computational Efficacy and Hardware Availability:* Perhaps because of their narrow problem scope more than other reasons, traditional tools generally have a reasonably high level of computational efficacy. Similarly, because most of the tools are written in traditional procedural languages, they are portable to a wide range of traditional computing hardware.

- *Economic Viability:* The economic viability of tools which support the traditional software development model is clearly demonstrated by the wealth of implementations available in the industry. In particular, the microcomputer software industry is an outstanding example of the economic viability of traditional software tools which can run effectively on widely available and inexpensive computing hardware.

2.1.3 The Human Interface

- *Complexity Reduction:* The complexity of the human interface on most traditional software tools is quite low. While some complex problems might require dozens of invocation options, most traditional software tools need far fewer. The current trend toward full screen and menu driven interfaces is reducing the interface complexity even more. For example, the MacIntosh interface is highly uniform across almost every application, requiring less of the user. IBM's SAA Presentation Manager indicates a similar trend in the mainframe world.

- *Education Required:* The mental concepts required by most traditional tools are easily supported by standard educational levels and/or competent on-the-job training. This relatively low educational requirement is probably a consequence of several factors which include the narrow problem scope of traditional tools, the easily understood languages, and the simple interfaces. The low level of abstraction and high level of concreteness found throughout the traditional lifecycle is also probably a significant factor.

- *Speed, Usability, Extensibility:* Traditional tools have a level of optimization unmatched by any other research approach. The obvious reasons for this include strong economic motives and long term development of construction expertise. As a result, traditional tools are relatively fast compared to research implementations of new development models.

  While most traditional tools have no extension capabilities because of their narrow problem scope and high level of optimization, text editors are regularly extended in both research and industrial environments. Two particularly motivating factors are (1) that extensions clearly increase productivity on very specific local problems, and (2) there are strong economic and aesthetic incentives to improve the interface of such a heavily used application program.

2.2 Application Generators

The application generator approach to software development is based upon the notion of reusing existing software solutions to solve recurring, (often standard) problems. Interesting aspects of the software solution are captured in the form of input parameters which the user can modify to fit the current problem situation.

This approach admits little opportunity for increased levels of automated support because of its already highly automated nature; early lifecycle phases (such as parameter design and selection) probably offer the most significant opportunity.

The application generator approach to software development is an accepted method in domains which can be effectively and economically treated with the approach. The approach is particularly effective in generating the data entry and reporting systems associated with database applications.

2.2.1 The Research Approach

- *Languages:* Application generators have been written in many languages for many problem domains. While they are mostly written to generate output in traditional programming languages because of the productivity gains possible in the conventional lifecycle, they can be used with any language and application where standard solutions to recurring problems are required. Parser generators are good examples of application generators, as are database generators. The application oriented languages used as input by application generators are narrow spectrum languages, as are (usually) the output languages.

- *Tools:* The tools associated with the application generator approach are primarily concerned with supporting the main generator itself. While this generator is most often viewed as a tool for improving production, it undeniably manages clerical and conceptual complexity too (eg. in a parser generator).

- *Domain Limitations:* Application generators are usually *brittle* (if at all functional) outside their intended domains. However, brittleness is not always a problem if the generator does a comprehensive job of covering a well defined domain — in such cases, the local problem situation will rarely exceed the range of the generator.

- *Lifecycle Support Offered:* The application generator approach, like automatic programming systems, can be said to support a large portion of the lifecycle. It supports some of the specification phase in that tentative generation specifications can be quickly validated through prototyping, clearly supports the implementation phase, and aids maintenance needs which can be treated by changing generation parameters and regenerating.

  The application generator approach is designed to support the individual programmer (programming-in-the-single), and can support the generation of applications of any

size. Programming-in-the-many supports such as source code access, version, and configuration controls, however, must be obtained from the conventional approach.

- *Opportunity for Automated Support:* The application generator approach arguably offers the best demonstrated productivity opportunity in current software development research -- so much so that implementations of this approach are often referred to as "fourth generation languages". However, the approach does have its problems. Application generators have a very limited range of flexibility in the products which they generate, do not always fare well against efficient third generation implementations [21], and still do not offer the generality or scope of a wide spectrum development model.

- *Completed Applications:* Many applications have been generated with application generators in a variety of problem domains.

2.2.2 The Implementation

- *Complexity Reduction:* Application generators do a reasonable job of reducing the external complexity of a problem situation by removing information which is not germane to the parameterized solution. In addition, the structure and organization of the required parametric information is often improved as a result of the implementation's human interface design process.

  The internal complexity of an application generator is not usually significantly greater than the complexity of the solution software itself, since the conceptual generation task is basically one of module selection and parameter substitution. The added complexity is partly related to identifying and representing the generic problem structures in a computationally efficient form, identifying and integrating the appropriate generation options into the generation system, and providing access to all relevant information in a suitable user interface.

- *Problem Size Capacity:* Since application generators have a demonstrated ability to deal with realistic problems, it is clear that the problem size limits of the conceptual approach are not a major issue. Problem size limits are more likely to be a property of particular implementations or target application domains. For example, the generic solution methods which a generator might use on small problems may be inappropriate for large problems which are better treated with special algorithms and data structures.

- *Computational Efficacy and Hardware Availability:* The computational efficacy of generator programs is demonstrated by their ability to produce satisfactory results on current hardware technology in a variety of application domains. However, the efficiency of the generated application is not always as satisfactory. In at least one instance [21], the programs produced by application generators did not save significant amounts of labor and were not as efficient as those produced using manual third generation techniques.

- *Economic Viability:* Application generators have demonstrated a strong economic viability over many years of existence, with market needs, tractable application domains, and the the costs of alternative development all playing a major role in the success of the approach. Traditional general development techniques will have to become very significantly more productive in order to challenge the viability of this well established this approach.

2.2.3 The Human Interface

- *Complexity Reduction:* The human interface complexity of application generators is not significantly more complex than that required by traditional development methods. The definition of generation parameters sometimes involves a new terminology and organization of data, but since the conceptual complexity of the input model is a function of the domain more than of the generator, the complexity required of the user by the model is relatively constant.

- *Education Required:* Since application generators are designed to improve productivity in standard applications, the standard terminology and domain concepts used by the generator will not likely require a level of user education beyond the normal level for that domain. A key point is that the application generator optimizes an existing conceptual model rather than introducing a new model which requires significantly more education.

- *Speed, Usability, Extensibility:* While application generators have demonstrated capability in the areas of generation speed and usability, they have very little extensibility outside their intended domain. As before however, this is not a large problem in generators which comprehensively cover a particular domain.

2.3 Structure Oriented Environments

The basis of structure oriented approaches to software development is the utilization of structural and semantic knowledge to better manipulate objects in the environment. This additional knowledge is usually captured in a formal grammar upon which some of the environment's tools are based; it allows structure oriented approaches to offer extended support where structural knowledge can offer an advantage. With the exception of the option of structural manipulation, the software lifecycle of structure oriented approaches is identical to the conventional model. As a consequence, opportunities for automated development support in structure oriented approaches are essentially the same as those available in the conventional model. It is only in the structure oriented manipulation of lifecycle objects that structure oriented approaches offer some advantage.

Despite having the ability to manipulate objects according to their structural relationships, the overall advantages of structure oriented environments have not promoted widespread acceptance within the software industry.

2.3.1 The Research Approach

- *Languages:* Structure oriented environments can be automatically generated for any language which can be expressed in the form of a suitable formal grammar. Most such environments are based on popular procedural programming languages (narrow spectrum), but environments have been generated for specification languages, modeling languages, and other symbolic languages for various purposes (all possibly wide spectrum).

- *Tools:* The mainstay of most structure oriented environments is a syntax directed editing facility which parses concrete program representations into internal abstract syntax trees. The resulting internal form can then be redisplayed in the concrete representation (that is, the language syntax) of the user's choice. This tool is primarily a production tool which helps the developer manipulate structured objects.

  As a group, syntax directed editors are capable of incremental parsing, incremental code generation, and other advanced processing based on specific "action functions" (which walk the abstract syntax tree) [20] or formal attribute grammars (whose values are computed on an incremental basis after each editing change) [23].

  One very important tool in such environments is the generator program which generates structure oriented tools from an input grammar specification. This makes it possible to generate environments for many different languages with a minimum of effort. While early structure oriented environments were dedicated to one particular grammar because of their manual construction, modern implementations can work with several different grammars at once.

  The most often cited problem with syntax directed editors is their non-intuitive user interface, which requires users to think in terms of the syntactic structure of the supported language (eg. blocks and trees) instead of viewing the program text as a linear sequence of text tokens. In particular, since some editors will not let the user create (even temporarily) a syntactically incorrect program (this implies a broken or inconsistent internal abstract tree), the user may be prevented from using familiar intuitive editing patterns. Non-intuitive cursor motions may also contribute to the problem. Most implementations try to mitigate this problem by allowing some degree of plain text editing in math expressions, under manual control, or by maintaining a completely separate (but consistent) pure text copy of the buffer contents for the user's convenience.

- *Domain Limitations:* The structure oriented approach to software development is limited by the representational power of its supported grammar. That is, it can only offer support where the concepts, objects, and operations in the chosen domain can be adequately represented by a formal grammar.

  A simple example concerns the unstructured program comments found in most software programs. Since most grammars do not treat comments as formal grammatical objects (they are usually stripped out by the lexical analyzer before they reach the parser), structure oriented editors must make special provisions (outside the grammatical model) to handle them.

  Even though a good deal of the software development process deals with unstructured text and relationships, thus preventing the benefits of the structure oriented approach from being realized with them, the resulting limitations are not very significant. Structure oriented systems are not brittle, and one can revert to conventional editing methods at any time.

- *Lifecycle Support Offered:* Structure oriented systems, as a group, normally do not provide broad lifecycle support. Rather, they have traditionally concentrated on a central structure oriented editing facility. As a consequence, any broad lifecycle support which accompanies the approach is usually provided by tools designed to support the conventional lifecycle model.

  The Gandalf system [13] is a notable exception to the statements above — this structure oriented environment provides support for both programming-in-the-large and programming-in-the-many activities.

- *Opportunity for Automated Support:* Probably the most often cited productivity advantages of structure oriented environments is that they can preclude (or detect) the creation of syntactically incorrect program modules, often using incremental processing techniques, and can provide various abstract views of an object based upon its structure. However, it is not clear that these features improve overall productivity when unintuitive interfaces and thinking patterns are also considered. While it is reasonable to assume that the syntax checking advantage will become less important as compilers and hardware resources become faster, it is also clear that the abstract structural view advantage will become *more* important as systems become more complex.

Excluding the editing task, the software lifecycle in the structure oriented approach is essentially the same as in the traditional approach, and so offers similar potential for productivity improvements through development of specific, task oriented tools.

- *Completed Applications:* Many structure oriented environments have been successfully constructed. While most are experimental in nature, a few are clearly full production systems in educational and research environments [24] [13]. The approach has not received wide acceptance in industrial situations.

2.3.2 The Implementation

- *Complexity Reduction:* The structure oriented approach generally adds complexity to the external implementation interface by requiring the user to deal with the problem and the implementation through grammatical or structural manipulations. In some instances this can be an advantage (particularly where structural manipulations are intuitive), but in general the conceptual complexity of a grammar is greater than that needed to deal with unstructured text or concrete objects.

  The internal complexity of tools in the structure oriented approach (compilers excepted) is also greater than corresponding tools in the traditional approach, again because of the need to deal with a grammar instead of unstructured objects. For example, a syntax directed editor contains much more internal complexity than a normal text editor of comparable power (structure manipulation excepted).

- *Problem Size Capacity:* The structure oriented approach is generally free to handle problems of any size, since grammatical models tend to scale up well. However, current implementations of this approach have had difficulties in handling the larger structures[1] and multiple users characteristic of programming-in-the-large (and many) situations [9]. These problems will no doubt be reduced as implementation techniques improve.

- *Computational Efficacy and Hardware Availability:* The computational efficacy of the structure oriented approach has been demonstrated many times on widely available hardware. Any reasonable implementation could be expected to be successful in this regard.

- *Economic Viability:* The economic viability of structure oriented implementations is demonstrated by their existence in the current marketplace, even though they have not been widely accepted in the software industry.

2.3.3 The Human Interface

- *Complexity Reduction:* The human interface to structure oriented tools is clearly more complex than that of corresponding traditional tools because of the need to represent and manipulate structural relationships in the implementation. However, even though the added complexity is easily understood by developers who understand grammatical concepts, the frequent literature references to unintuitive editing operations indicates that the structural complexity has a significant impact.

- *Education Required:* It is reasonable to assume that the structure oriented approach can be adequately supported by standard educational levels, since a developer whose can program in a particular programming language will likely be able to understand the concepts of syntax directed processing.

- *Speed, Usability, Extensibility:* The speed of tools in the structure oriented approach is generally comparable to that of traditional tools, implying a reasonable degree of computational efficiency. However, their ease of use seems in doubt (unintuitive).

  Perhaps the largest barrier to extensibility is the the need for extensions to share the fundamental (and often complex) internal data structures which underly the main editing facility [9].

[1] For example, it becomes difficult (if not impossible) to load complete syntax trees for very large programs into a single editor buffer.

2.4 Exploratory Programming Systems

The essence of the exploratory programming approach to software development is the *incremental, evolutionary* development of complex software systems. In contrast to the heavy development guidance (or constraints) provided by the conventional specification, validation, and design review processes, exploratory development often involves incremental and tentative implementation explorations rather than being completely specified and designed prior to implementation. Though seemingly uncontrolled, the method is clearly a valid one when the form of the final system is unknown [28].

Opportunities for automated development support are plentiful in most exploratory programming environments because of the high information coupling which is possible between the underlying host environment, development tools, and the application programs. The wealth of contextual information which is available in such environments can often be utilized in the construction of more intelligent development tools.

The exploratory approach to software development is an accepted method with decades of history. Many significant software concepts and tools were originally developed in such environments.

2.4.1 The Research Approach

- *Languages:* Exploratory programming systems (also known as *language oriented systems* [9]) have traditionally used wide spectrum symbolic processing languages which are interpreted rather than being compiled (Lisp, Smalltalk, Prolog, others). Interpreted languages can generally provide faster feedback in the program development process because the developer works in an easily changeable and storable *environment* rather than having to rebuild a new runtime environment with every execution of a program. The use of interpreted languages and environments thus adds both flexibility and productivity to the development process.

- *Tools:* The tools found in exploratory programming environments are arguably the most advanced tools found in the software development field (highly extensible editors [30] [28] [33], advising capabilities, Masterscope, Dwim [15], and KBEmacs [32]). One factor in the development of this situation is that many researchers in the exploratory programming field are both developers and users of the systems at the same time — this dual focus encourages the creation of advanced (research oriented) but useful (user oriented) tools. Many standard features of current programming environments have their roots in the exploratory environments of the past two decades (windows, menus, mice, objects, extensible editors).

  Both tools for managing complexity and tools for increasing production can be found in these environments. For example, the Masterscope facility of Interlisp is one example of a tool which *provides information* about complex systems. Masterscope provides answers to questions such as "Which functions use the parameter X?" and "Where

is the variable Y modified?". The Defsystem tool of the Lisp Machine is an example of a tool which *manipulates* complex systems. It allows large systems to be defined declaratively and manipulated on the basis of a symbolic definition of the system. This method separates knowledge of system structure from knowledge of how to process a system for specific tasks (eg. compilation), allowing one specification of how to process a system to be used with many symbolic system definitions. In contrast, the corresponding tool from the traditional programming approach, the Unix *make* utility, is considerably less capable.

  Probably the most unique characteristic of exploratory environments is the use of one programming environment to simultaneously hold system facilities, tools, and applications. This seamless interface between components makes all resources fully accessible to all components -- a feature which aids development productivity, but which sometimes decreases modularity and portability.

- *Domain Limitations:* The exploratory programming approach has no significant domain limitations. Entire and practical computing facilities (Smalltalk, Interlisp, Lisp Machine) have been created using this model of software development.

- *Lifecycle Support Offered:* Most exploratory environments offer little in the way of widespread lifecycle support because they have traditionally focussed on the development of small or medium sized projects written by individual developers. Despite this focus however, such systems often have advanced late lifecycle maintenance facilities (eg. the Cedar system model, the Lisp Machine Defsystem model) which are still beyond the conventional approaches exemplified by the Unix *make* utility.

- *Opportunity for Automated Support:* The exploratory programming approach admits both more and less opportunity for automated development support than does the traditional development approach. It admits more opportunities in that the development environment and language and the application environment and language are usually one and the same — this feature provides an information channel which can support huge amounts of information transfer and operational manipulations between the application, the development tools, the environment, and the developer. Yet the exploratory approach also provides less opportunity in that its development lifecycle tends to contain reduced requirements and design activities. Because development proceeds in small, manageable increments, the approach does not have the same need for automated support that other approaches do.

- *Completed Applications:* A great deal of software has been developed with these full production systems.

2.4.2 The Implementation

- *Complexity Reduction:* The external, visible complexity of exploratory programming implementations is usually greater than that found in conventional environments, pri-

marily a result of the languages and advanced tools which are available in the typical exploratory implementation. For example, the notions of environments, advising functions, functional languages, object oriented programming, and manifest data types are not usually associated with the conventional development approach.

The internal complexity of such implementations is also clearly greater than that of traditional systems, again mostly a result of the languages and the advanced *environment* implementations. One probable cause of increased internal complexity is the high degree of coupling which can occur between the application, tools, and the environment in the form of application dependencies on environment and tool facilities. The fact that all programs reside in one large environment and that all facilities are available to all components can lead to high levels of coupling (and lower levels of modularity and portability) between components of the system.

- *Problem Size Capacity:* Exploratory programming implementations are not constrained to simple problems of a given size unless the problem, not the implementation, requires excessive computational resources. It takes a significant problem to exhaust the computational power of most exploratory environments.

- *Computational Efficacy and Hardware Availability:* The interpreted nature of the exploratory programming approach necessarily results in implementations which are somewhat slower than traditional environments when considered in contexts where the traditional approach is strong (eg. math operations, string processing). Yet when considered in the domains for which they were designed (and even in ones for which they were not), exploratory environments are clearly computationally efficacious.

  The hardware support required by (or maybe just typical of) exploratory environments makes them somewhat more expensive and less popular than comparable traditional computing facilities. Their hardware support normally includes large quantities of dedicated computing power, virtual memory, disk space, and display monitor surface area, with comparable software support — advanced editors, software inspectors, debugging facilities, and program maintenance tools. It should be noted, however, that any differences between "exploratory hardware" and "traditional hardware" are rapidly disappearing with advances in hardware technology. For example, it is currently possible to run a full Interlisp system (with very respectable performance) on an advanced (Intel 80386) personal computer.

- *Economic Viability:* The economic viability of exploratory programming implementations in research environments has been demonstrated by their continued survival over several several decades. However, the viability of such implementations in the non-research industry is in some doubt, as they have received less acceptance where the features of the approach are not a prerequisite.

2.4.3 The Human Interface

- *Complexity Reduction:* The external complexity of a full exploratory implementation is clearly greater than that of a comparable traditional computing facility. However, the complexity can clearly be overcome by normal developers in a reasonable length of time. Competence can usually be developed after approximately six months, but it can take years to develop expert familiarity with such systems.

- *Education Required:* The exploratory approach does not require more education of the user so much as it requires a different (more abstract and symbolic) way of thinking. Some users adapt quickly, and some never adapt at all. Formal instruction in the concepts behind the approach and its implementation would probably benefit most users, as strong foundations in the approach are usually best instilled through a formal course of instruction (on or off the job).

- *Speed, Usability, Extensibility:* There is little doubt that the hardware, software, and development speeds associated with exploratory programming environments are very acceptable. In addition, their extensibility is unsurpassed. (However, as stated above, this ease of extension can lead to portability problems if applications depend heavily on extensions peculiar to the original development environment.)

2.5 Automatic Programming Systems

The essential development method in automatic programming systems which are based on the *transformational approach* involves the automatic translation of an initial high level specification into one or more executable programs through the application of *transformation rules*. The transformational process can be fully automatic [17], partially automatic [10] [22], or non-automatic [8] [1].

The automated programming approach precludes many opportunities for automated support because its conceptual lifecycle model involves only two major steps: develop a requirements specification and translate it into an implementation. And since the primary opportunity for automated support in such approaches, transformation of the specification, is a central research issue in the approach, there are few other candidate activities which can be automated.

The automatic programming approach to software development is still a research issue for a variety of reasons, and so has not gained wide acceptance in the software community.

2.5.1 The Research Approach

- *Languages:* The languages used in this approach are generally wide spectrum specification languages which can support high level data and procedural abstractions. They are more difficult to read and to work with than traditional programming languages, but they have an expressiveness and power well beyond most other languages of any kind.

  A second kind of language found in many transformational systems is one which describes sequences of transformations which can be applied to a subject program specification. These narrow spectrum languages (eg. Paddle [1]) can be interpreted to "replay" a transformation history automatically, saving the effort of manual application during the iterative cycles of the specification validation phase.

- *Tools:* The tool set found in transformational environments generally includes at least a library of transformation rules and an interpreter which applies the rules. In addition, several environments have some sort of interpreter which can symbolically execute specifications written in the high level specification language and a paraphraser to translate the specification language into something more readable. As a general rule these tools are designed to manage complexity (transformations, complex specification languages). A complement of regular production-oriented programming tools (editors, compilers, etc) usually accompanies the special purpose collection of research tools.

- *Domain Limitations:* The conceptual automatic programming approach itself has no domain limitations. However, all prototype systems which have been developed have been limited to a relatively narrow application domain for a variety of reasons, including (1) a general transformation rule collection would be massive, (2) the number of possible implementations of a program (the search tree size) gets exponentially larger as problem size increases, and (3) the very large domain knowledge required for a general programming system is beyond current technology. These factors alone make a wide-domain implementation unrealistic with current technology, even if the brittleness of the approach is ignored.

  Notwithstanding the above, some trial production systems based on this approach have been constructed [1] [22] [8].

- *Lifecycle Support Offered:* The automated programming approach can be said to offer machine assistance to all lifecycle phases of the conventional waterfall model. It offers validation support for the specification phase, transformation support for the implementation phase, and "replay" support for the maintenance phase. However, in practice each one of these phases may require human input comparable to conventional methods.

  Automatic programming systems currently support *programming in the small* and *programming in the single* activities.

- *Opportunity for Automated Support:* The automatic programming approach provides few opportunities for more automated support because its shortened lifecycle (specify and transform) is already supported with automated tools. It is interesting to note that even though transformation systems clearly have the ability to implement a program from a specification, experience indicates that human effort saved in coding and testing the low level implementation is often consumed in directing the low level transformation of the high level specification. As a consequence, significant gains in productivity have not been realized [10].

  While it can be argued that specification-based automatic programming systems can provide many more opportunities for improvement in the areas of software reuse, reliability, productivity, and standardization, little research evidence exists to support the claim. Application generators are often brittle, automatic programming systems are plagued with unreadable languages and the need for significant amounts of very competent human input throughout the shortened lifecycle, and intelligent programming apprentices are currently limited by the computational inefficiency and conceptual complexity of their approaches. It is unlikely that these problems will be resolved in the near future.

- *Completed Applications:* Automatic programming implementations are still research systems, though some have reached the trial production system stage [1] [22] [8].

2.5.2 The Implementation

- *Complexity Reduction:* Automatic programming systems have a much higher level of external and internal complexity than other environments, primarily because of the high level specification languages upon which they are based, but also because of the

many transformations (hundreds sometimes) which must often be applied to generate a compilable representation of the specification.

- *Problem Size Capacity:* Fully automatic implementations of this approach [17] tend to be slow because of the computational costs involved in interpreting the specification and selecting a transformation. Simple one page programs can consume tens of minutes of computer time, even when many reasonable heuristics are used to reduce the search space used by the system. Semi-automatic and non-automatic implementations do not seem to fare much better, since the human effort involved seems excessive. It is reasonable to conjecture that such systems will be limited to smaller problems pending advances in modeling, heuristic pruning, and transformation selection techniques.

- *Computational Efficacy and Hardware Availability:* It has been shown that computationally efficacious transformational systems can be run on widely available hardware [8] when human input to guide the transformation process is available. However, it is not clear that fully automatic transformational systems fair as well in this regard. Their increased need for memory and computation, especially on problems with large search trees, makes them less attractive on low or medium powered computer systems.

- *Economic Viability:* The economic viability of the automatic programming approach is essentially untried because only research systems have been developed. However, until such systems can demonstrate an ability to handle realistic problems with standard hardware resources and human educational levels it is reasonable to assume that their economic viability will be very low.

2.5.3 The Human Interface

- *Complexity Reduction:* The visible complexity of automatic programming systems is clearly much higher than that required for traditional systems, largely because of the different languages and activities introduced by the different conceptual development model.

- *Education Required:* Effective use of automatic programming systems will require significant educational upgrading for most potential users, and may never be a realistic expectation [8]. The new conceptual development model and associated languages of this approach are so different from those taught in standard educational environments that the standard educational system provides little relevant support.

- *Speed, Usability, Extensibility:* The speed and usability of systems based on this approach seem less favorable in comparison to other development methods. While some implementations of the approach can be extended with effort ([8], [22]), most implementations cannot be extended at all.

2.6 Knowledge Based Systems

The overall approach of knowledge based software development systems is to use a data base of software knowledge to improve some aspect of the development process. Implementations of the knowledge based approach include automatic programming systems (Libra [17], Glitter [10]), "programmer apprentice" systems (KBEmacs [31]), and approaches based on the reuse of software libraries (RSL [6]). The wide variety of approaches makes any generalization in a survey paper such as this one very difficult, if possible at all.

Since the automatic programming approach was discussed earlier, and since the MIT Programmer's Apprentice Project [27] epitomizes the goal of the knowledge based approach. the following discussion is heavily oriented toward KBEmacs alone.

The software development model used by KBEmacs is similar to the traditional software lifecycle. The essential difference is that KBEmacs is intended to act as an *active* assistant throughout the lifecycle, advising, commenting, implementing, and generally interacting with the human developer throughout the whole development process. In addition, the development process makes heavy use of pre-defined software models (*cliches*).

The KBEmacs implementation is still a research system, and so has not been exposed to the general software community.

2.6.1 The Research Approach

- *Languages:* The languages used by most knowledge based *automatic programming* systems usually include a high level abstract language for the specification, and a lower level language to implement the analysis, synthesis, and reasoning engines of the knowledge based system.

  Since the operational model of KBEmacs involves translation of the edited language into a symbolic internal language and back again, the external, user visible language used by KBEmacs can be any language, wide or narrow spectrum, which is supported by the translation modules. A narrow spectrum *plan calculus* language is used internally to represent the data and control flows of a program in a language independent manner. The editor and knowledge based reasoning components are implemented in Lisp.

- *Tools:* Even though KBEmacs resides in a production oriented extensible editor tool (the Lisp Machine Emacs editor), the "knowledge based" part of KBEmacs is better characterized as a tool for managing the complexity of the software development knowledge stored in the implementation. The main software components which support KBEmacs are (1) an analysis module for analyzing existing programs and translating them into the internal symbolic *plan calculus*, (2) a synthesis module which can construct source code in the target language from the symbolic plan, and (3) a library of standard coding *cliches* (models) in which the symbolic program plan can be expressed and manipulated. The program's source code in the regular Emacs editor buffer is the

primary communication medium between the developer and KBEmacs—both parties "speak" to each other by changing the source code in the editor buffer. Of course, both the developer and KBEmacs can also interact through the editor's minibuffer facilities.

- *Domain Limitations:* The KBEmacs approach to software development does not have any domain limitations because, like the conventional approach, it does not make any limiting domain-specific assumptions. The KBEmacs implementation, however, is clearly limited by the domain of knowledge which is currently contained in its extensible knowledge base. KBEmacs is not a brittle system, as manual development techniques can augment (or entirely replace) knowledge based operations as domain boundaries are exceeded.

- *Lifecycle Support Offered:* The intended goal of the Programmer's Apprentice Project is to support all phases of the lifecycle [27] [7]. However, the current KBEmacs implementation only supports the middle lifecycle phases (implementation), and can be best characterized as a programming-in-the-small, programming-in-the-single system.

- *Opportunity for Automated Support:* As with the traditional lifecycle, the knowledge based KBEmacs development model offers many opportunities for automated support outside the implementation phase of the lifecycle. The chief support provided by the current implementation *inside* the implementation phase is automated code generation and consistency enforcement. For example, KBEmacs can implement many standard code fragments given the appropriate parametric information. In addition, if a developer makes an implementation decision which he later violates with a conflicting order to the assistant, KBEmacs can sometimes detect the conflict and bring it to the developer's attention.

- *Completed Applications:* The KBEmacs implementation and the Programmer's Apprentice project are still subjects of active research, and so are best characterized as experimental systems.

2.6.2 The Implementation

- *Complexity Reduction:* The external complexity of the KBEmacs implementation is quite low due to the simple nature of issuing knowledge-based commands [32] to the editor. However, the internal complexity of the system is massive, largely a result of the internal *plan calculus* representation and the modules which manipulate it during analysis and synthesis. To the novice user, the calculus alone seems formidable enough to preclude extending the plan library in any significant way.

- *Problem Size Capacity:* The problem size capacity of the current system appears to be limited to programs of a few hundred lines [32] because of the heavy computational requirements of the system.

- *Computational Efficacy and Hardware Availability:* The computational efficacy of KBEmacs is considerably lower than it might be because it does much more full analysis of the source program than is necessary — each time the user modifies the buffer contents in any way, a complete reanalysis of the entire buffer is done. Incremental analysis, once the research problems are solved, will save a great deal of computation.

  As implemented, KBEmacs runs on a dedicated Lisp Machine workstation. Hardware of this type is not as readily available as standard computational resources. It is unlikely that KBEmacs will run on widely available hardware in the near future.

- *Economic Viability:* The economic viability of the KBEmacs approach (in its current form) is low because of the problem size constraints, hardware expense, and poor computational efficacy. When these problems have been removed, the viability of the implementation will no doubt improve.

2.6.3 The Human Interface

- *Complexity Reduction:* The external human interface to KBEmacs is comparable to that of standard development tools. However, the internal complexity of the plan calculus library is very high. This may preclude a significant number of users of the system from extending the plan library in any meaningful way.

- *Education Required:* It seems reasonable to conjecture that effective use of the system could be supported with standard educational levels and on the job training. However, effective manipulation and extension of the plan library would require significantly more knowledge and training of the user.

- *Speed, Usability, Extensibility:* The speed of the current implementation is very slow, requiring several minutes even for simple operations. The system usability is high because it can be operated within the standard editor command model on source code in standard programming languages. Finally, while the editor itself can be easily extended to an arbitrary degree (KBEmacs is itself an extension), extension of the plan library or the functionality of KBEmacs is not easy.

2.7 Software Reuse Systems

The general concept of the reuse approach to software development is to build up a database of software modules which can be reused in future applications to save development costs. Research problems include the construction of reusable software, the classification and assignment of module attributes as they are stored in the database, and the identification and selection of suitable modules to retrieve from the database.

The approach admits little opportunity for increased levels of automated support because it does not actually generate new software; better candidate module selection, rating, and manipulation tools probably offer the best opportunity for increased automation.

The concepts of the software reuse approach to software development are widely accepted in the software industry, although practical implementations are less prevalent for a variety of economic, technical, and political reasons [19].

2.7.1 The Research Approach

- *Languages:* The reuse approach is language independent with respect to the language of the stored software modules, and so the only languages that users typically must deal with are the database languages used to query and manipulate the stored reusable modules. In practice, search queries are not complex because of the limited number of descriptive attributes and simple (independent) relationships characteristic of the stored data.

- *Tools:* The tools associated with reuse approaches are generally concerned with manipulating the database and the reusable software components. Examples of tools from [6], a representative reuse system, include library maintenance tools, software component retrival and evaluation tools, and an integrated library-search/software-design tool. Generally speaking, tools associated with this approach are conventional database application tools.

- *Domain Limitations:* Since it can store and reuse any component, the reuse approach has no theoretical domain limitations.

- *Lifecycle Support Offered:* The reuse approach indirectly supports a large part of the conventional lifecycle because it can mitigate the need for detailed design, implementation, unit testing, and documentation of components which are reused. Early lifecycle phases have not been supported by most reuse systems because of their focus on the reuse of executable code versus specifications. (There is no reason that the technology could not be applied to specifications instead of code.)

  The reuse approach can support single or multiple developers on both large and small projects.

- *Opportunity for Automated Support:* The opportunities for automated support of the reuse approach are generally limited to the installation, selection, and retrieval of candidate software modules from the software database. Some success has been achieved with semi-automatic selection tools [6], although they are crucially dependent on uniform assignment of reuse attributes at the time modules are installed into the database.

- *Completed Applications:* The software community has surely reused an astronomical number of modules by now, with no end in sight.

2.7.2 The Implementation

- *Complexity Reduction:* Reuse databases reduce the external complexity of searching for candidate reusable software modules by providing an appropriate attribute organization and a structured searching facility.

  The internal complexity of reuse databases is comparable to that of simple database applications.

- *Problem Size Capacity:* The reuse approach has no theoretical limits on problem size. In principle, arbitrary numbers of software modules and systems of any size can be stored in the database. Problem size limits are more likely to be a property of particular database implementations.

- *Computational Efficacy and Hardware Availability:* The computational efficacy of reuse databases is demonstrated by their ability to produce satisfactory results using current hardware technology.

- *Economic Viability:* The economic viability of reusable code is demonstrated by its ubiqutous presence in both large and small software development organizations, even though few of them have a formal facility to support the reuse approach. The economic costs and unclear benefits of a reuse system have been deterrents to many organizations.

2.7.3 The Human Interface

- *Complexity Reduction:* The human interface complexity of the software database is comparable to standard database applications (relatively simple). However, as [6] points out, the activities of classifying reusable software modules and assigning reuse attribute values are best done by an experienced expert developer to ensure that the selection tools (which are heavily dependent on the attributes) have a uniform information base to work with.

- *Education Required:* The reuse approach does not require additional educational training on the part of the user. Any developer who knows the attributes of the module he is searching for can be expected to find and recognize the module if it is contained in the database.

- *Speed, Usability, Extensibility:* While the speed and usability of informal implementations of the reuse approach probably leaves something to be desired, the unbroken historical record of use of this approach speaks highly in its favor. The notion of starting with an existing solution which is almost right and modifying it to suit the purpose at hand is an old and respected method in almost every discipline [25].

  The domain of the reuse approach can, of course, be extended simply by adding new reusable modules to the database.

Chapter 3

The Productive Development System

> What kind of an ideal environment would offer the highest, most general, and most extensible level of overall productivity to the knowledgeable individual software developer?

This chapter discusses the characteristics of a software development system which could significantly improve the productivity of a single knowledgeable developer. The following sections of this chapter

1. State the assumptions under which the answer is developed,
2. List two major concepts underlying the proposed development approach,
3. Present a sequence of abstraction levels which should be supported by the proposed system, and finally,
4. Discuss the knowledge and procedural capabilities required by the preceding sections.

3.1 Assumptions

The following assumptions have been made to clarify the problem question:

- The primary goal of the system is to support software *development*, not software debugging, conversion, modification, or maintenance.
- The individual programmer knows exactly what software needs to be constructed. That is, the requirements specification is assumed to be complete. (This assumption removes the need for considering early lifecycle supports in the proposed solution system.)
- The proposed system should be designed to maximize the productivity of a knowledgeable individual developer (programming in the single). The developer is assumed to have a standard level of computing science education.

- The system should support the development of applications within the reach of a single developer's mind. Using current technology, such an application might possibly be up to 50,000 lines of code, but is more likely to be 30,000 lines of code or less (programming in the small to medium).

- The system should run satisfactorily on modern personal computer hardware (eg. Intel 80386 or Motorola 68000 class hardware). Such machines can provide graphical interfaces and adequate quantities of disk space and computational power.

3.2 Concepts of the Development Approach

"What techniques hold the most promise for maximizing the productivity of the knowledgeable, motivated, *individual* developer?"

1. The reuse of software which implements useful conceptual models is probably the best chance of improving the productivity of the individual developer because the developer can work with useful abstract functional interfaces rather than typing in procedural source code. While many other factors can impact productivity, they are insignificant when compared to this approach, which allows a single mind to grasp and reuse hundreds (if not hundreds of thousands) of man-hours of expended labor. (A key assumption here is that reusable code with the appropriate attributes of language, reliability, size, speed, and algorithm is available.)

   In support of this view, it is obvious that reusing an existing software solution is cheaper and faster than creating and debugging a new one, all else being equal. For example, application generators can generate massive amounts of functional code from only a few specifications. The correctness of this view also appears to be generally accepted by the industry: the ubiquitous complaint of "reinventing the wheel" is prevalent in the literature.

2. Where reuse of code at the whole application level is not possible (that is, in the context of "first-time" applications), the greatest chance of productivity lies in the reuse of software abstractions at the highest level allowed by the problem at hand.

   The key point is that the developer should have the ability to reuse existing conceptual models (and the code which implements them) at varying levels of abstraction, according to the needs of the problem. Developers should be experts at working with conceptual models and interfaces (a mental activity), not at writing source code (a clerical activity).

   In the Productive Development System, programming would more nearly become a *concept manipulation* occupation, and programmers would more nearly work in what Brooks [5] calls "the pure thought stuff."

3.3 Abstraction Levels of the Approach

Assuming the conceptual approaches above, a developer would use the following abstraction levels in order of preference. That is, the developer would maintain as high a level of abstraction, and thus reusability, as possible.

- The developer would use canned solutions for standard problems. This is the highest level of abstraction (at the application level), and could be supported with (1) system libraries which contained candidate solutions or with (2) application generators which could generate such solutions.

- Where the problem required some degree of customization, the developer would manually tune a completed application which was close to the needed solution. A large part of the existing program could and would be reused.

- Where major customizations or completely new implementations were required for new (but easily modeled) problems, the developer would revert to the subsystem level of abstraction. Standard conceptual models would be used for the standard abstractions of computer science: data base subsystems, report writing subsystems, user interface architecture abstractions, and other such subsystem level components.

- In cases where even subsystem level components could not be reused, the developer would use even finer levels of abstractions such as those taught in standard computer science curricula: abstract data structures and operations, procedural processing techniques and modular organization. Both exploratory and conventional programming technologies would be used for truly difficult (or original) problems.

3.4 System Knowledge Requirements and Procedural Capabilities

A competent implementation of the Productive Development System would require substantial fulfillment of the following needs:

- *A body of codified software models.* The most obvious initial need of the system is for a codified body of conceptual models which can be used together to create effective software programs. Unfortunately, no such suitable collection exists. The models presented in standard computer science data structure textbooks are too general (their models do not always fit together easily and are not always efficient), and the standard subroutine libraries of most programming languages, while they are efficient and useful, only cover the lowest of abstraction levels.

  Thus one of the first steps in creating the Productive Development System will be to conceptualize and codify a set of models which will support the needs of prototype implementations. The content of such a collection is discussed in the next paragraph.

- *Multiple layers of abstraction in the model library.* In order to effectively support a wide variety of software development domains, a mature collection of conceptual software abstractions must include the following models:
  - *Knowledge of standard application programs.* Models in this class would include complete and comprehensive implementations of popular applications such as database systems, accounting systems, and word processors, as well as for traditional software tools such as compilers, assemblers, editors, debuggers, and version control systems. Application generators would be included for popular applications which could be effectively treated with parametrization and generation techniques.
  - *Knowledge of user interface architectures.* Since all programs must use one of the few known standard user interfaces (command line, menu, mouse), this class of models should be one of the first to be codified.
  - *Knowledge of subsystem level software models.* Window and menu subsystems, database subsystems, data entry and editing subsystems, and data structure subsystems are all sample candidates for models in this category.
  - *Knowledge of module level abstractions.* Models in this class of knowledge are typically implemented with subroutines and "includable" data structure definitions, making them easy to represent and manipulate across programming languages. Most currently codified knowledge falls into this class of models.
  - *Knowledge of low level software models.* This class of models has not yet been codified effectively, perhaps because of its detail and variety in form or because models in this class are relatively easy to reinvent and reimplement on a daily basis. The models in this class of knowledge form the substance of introductory programming courses.
- *A suitable machine representation of the models.* Once a suitable collection of software abstractions has been codified in the form of appropriate models, they must be represented in a machine-manipulable form so that they can be used by the system in the service of a developer.

  The development of uniform and effective representations for the levels of abstraction suggested above is not a simple problem, and can lead to relatively complex implementations such as the *plan calculus* described in [25].
- *A suitable method of accessing the stored models.* The Productive Development System should be capable of accessing the database and instantiating a software model both interactively from the main development editor and non-interactively from a batch file (the equivalent of *replaying* a transformation sequence [1]). Access to the model library during the design phase [6] [14] would also be an asset.
- *A suitable method of joining the models.* The ability of the system to efficiently join selected models together into a working architecture is a key point in automating

the development process. Software development can be a *declarative process* (with all the associated benefits) as long as the developer is not required to manually edit the software under construction. Providing full declarative processing support for the software models in even one or two of the abstraction levels above would be a major productivity achievement.

- *Models in any language.* Finally, given the highly conceptual nature of the proposed development approach, the Productive Development System should be capable of instantiating software models in any programming language which supports the concepts used by the model itself. This too would be a major productivity accomplishment, because it would allow the reuse of conceptual models (design information) across traditional programming language barriers [14].

Chapter 4

Analysis and Conclusion

The preceding chapters of this paper have (1) presented a set of criteria for characterizing software development approaches in the context of improving the development productivity of a single developer, (2) applied the criteria to a set of candidate approaches representative of current software technology, and (3) characterized an idealized software development environment for the individual developer.

This chapter attempts to use the evaluations of Chapter 2 and the idealized model of Chapter 3 to answer the question posed in the introduction:

> What techniques hold the most promise for maximizing the productivity of the knowledgeable, motivated, *individual* software developer?

The following sections present some advantages and disadvantages of the approaches discussed in Chapter 2 and identify the most promising techniques for improving individual productivity.

4.1 Summary of Approaches

4.1.1 Conventional Development

- *Advantages:* The advantages of the conventional approach include generality, low complexity, flexibility, extensibility, and few domain limitations.
- *Disadvantages:* The primary disadvantage of this approach is the quantity of manual effort which it requires, a condition which increases software costs, lowers reliability, and decreases standardization and sharing of existing work.

4.1.2 Application Generators

- *Advantages:* Application generators are fast, specific, and effective, providing unparalleled productivity within their intended domains.

- *Disadvantages:* The obvious problem with application generators is their lack of broad domain support. However, this really doesn't count against the approach; instead it indicates that too few application generators are available to cover the domains of interest.

4.1.3 Structure Oriented Environments

- *Advantages:* The central offering of structure oriented environments is that they can allow (or enforce) manipulation of objects according to their structural relationships. It is unclear whether this is a significant advantage.
- *Disadvantages:* Experience indicates that any advantages the approach has may be offset by the sometimes unintuitive nature of structured manipulation. Thus the approach may be adding complexity without comparable benefits.

4.1.4 Exploratory Programming Systems

- *Advantages:* The main advantages of this approach are the use of symbolic languages in the context of an extensible programming *environment* and the use of a powerful incremental development approach. These features both increase developer productivity.
- *Disadvantages:* The ease of incremental development in these environments can encourage developers to "patch on" too many new features to existing program architectures, sometimes leading to systems whose final architecture is not well thought out.

4.1.5 Automatic Programming Systems

- *Advantages:* Probably the most significant productivity advantage of these systems is their ability to "replay" the development of a program, allowing similar programs to be developed with less effort. A side benefit is that they codify a potentially useful body of transformation and programming knowledge.
- *Disadvantages:* Excessive complexity appears to plague these systems in their specification languages (which often require paraphrasing) and in their transformation processes (which require significant human guidance). Commensurate improvements in developer productivity and program efficiency are not obvious.

4.1.6 Knowledge Based Approaches

- *Advantages:* KBEmacs has shown some of the potential power of an intelligent programming engine combined with a knowledge base of standard programming models. The power of this approach appears to be only surpassed by human programmers.

- *Disadvantages:* The current implementation is complex, costly, and slow.

4.1.7 Software Reuse Systems

- *Advantages:* The advantages of this approach include increased software reliability, lower lifecycle costs, and increased standardization in software interfaces and architectures. Only application generators offer better productivity than this approach.

- *Disadvantages:* The approach is currently not based on matched sets of conceptual models and software implementations, so finding a reusable module to fit the local design model has been (and still is) the central problem. The construction, classification, and retrieval of reusable software are also research issues.

4.1.8 The Productive Development System

- *Advantages:* The sliding scale of reusable abstractions, the matched implementations, and the automated tools for manipulating the models will help to promote *optimal* productivity and code reliability across a wide domain. The developer will only have to manually add code where the abstraction models are insufficient for the task.

- *Disadvantages:* The approach is more abstract than current software development models, and thus will require more education and discipline of the developer. Of course, the most obvious disadvantage of this "approach" is that it hasn't even been rigorously conceptualized as of this writing.

4.2 Promising Techniques

The most promising techniques for improving the development productivity of an individual programmer currently hinge around the development of a system which has the following three major characteristics:

1. *A database of domain knowledge.* The system must be conceptually based on compatible software abstraction models with matching implementations. The representations of both models and implementations must be suitable for machine manipulation.

2. *An intelligent model manipulation engine.* The system must have some efficient automated means of identifying, accessing, combining, replaying, and otherwise manipulating *both* the conceptual models and the matching implementations.

3. *A convenient human interface.* The system must attempt to provide a low complexity, extensible human interface which can be used to access and control the system's resources.

Portions of these two basic needs can be identified in some of the systems of Chapter 2:

- *Application generators* clearly have both a sufficient domain knowledge and intelligent synthesis engines. The combination of these two elements is the basis of the unsurpassed productivity of the application generator approach.

- *Software reuse systems* have a low level of domain knowledge which is insufficient for increasing productivity into the application generator range, yet which offers much broader support than narrow application generators. Reuse systems have no engines, and so are very limited in their synthesis capabilities.

  It is interesting to note that application generators and reuse systems share essentially the same approach; the only real difference between them is that one contains narrow, extensive domain knowledge and has a synthesis engine, and the other contains broad, shallow domain knowledge and comes without an engine.

  It follows that combining an intelligent engine with a suitable set of matched conceptual models and implementations would offer the power of the application generator approach with the breadth of the reusable software approach. This is the approach taken by the Productive Development System in the previous chapter.

- *Exploratory programming environments* based on the notion of an *environment* integrate the full power of the development system, complete with state information, into a relatively seamless user interface. The incremental development model is also a strong feature of such environments, and is arguably the method of choice when working on large complex systems.

4.3 Conclusion

This paper has attempted to determine the most promising techniques for increasing the productivity of a knowledgeable, motivated *individual* developer by characterizing several current research systems and approaches in the context of a set of interesting criteria.

In addition, the paper characterized the Productive Development System, a model of an idealized environment based on the notion of a sliding scale of matching conceptual software abstractions and their corresponding implementations.

Finally, the paper summarized the advantages and disadvantages of the research systems, drew three important system characteristics from that summary, and identified those approaches which exhibited the important characteristics in some degree.

In closing, the arguments presented above indicate that there is good potential for maximizing an individual developer's productivity in an interpretive programming environment which is based on the machine-assisted manipulation of matching conceptual and physical (implemented) software models.

Chapter 5

Bibliography Organized by Research Approach

5.1 Automatic Programming Systems Bibliography

- Balzer, Robert. *A 15 Year Perspective on Automatic Programming.* IEEE T. Software Engineering Vol 11(11):1257-1268, Nov 1985.
- Balzer, Robert. *Editorial: Program Transformations.* IEEE T. Software Engineering 7(1):1, January 1981.
- Balzer, R. *Transformational Implementation: An Example.* IEEE T. Software Engineering 7(1). 1981.
- Barstow, David R. *A Perspective on Automatic Programming.* Reprinted in *Artificial Intelligence and Software Engineering*, Rich and Waters, Morgan Kaufman, 1986.
- Barstow, David R., and Kant, Elaine. *The Refinement Paradigm: The Interaction of Coding and Efficiency Knowledge in Program Synthesis.* IEEE T. Software Engineering 7(4):458-471. September 1981.
- Barstow, David R. *Domain Specific Automatic Programming.* IEEE T. Software Engineering 11(11):1321-1336. November 1985.
- Cheatham,T., Townley, J., and Holloway, G. *A System for Program Refinement.* IEEE Fourth International Conference on Software Engineering, Munich, Germany. September 1979. pp. 53-62.
- Cheatham, Thomas E. *Reusability Through Program Transformations.* IEEE T. Software Engineering 10(5), 1984. Also reprinted in *Artificial Intelligence and Software Engineering*, Rich and Waters, Morgan Kaufman, 1986.

- Fickas, Steven F. *Automating the Transformational Development of Software.* IEEE T. Software Engineering 11(11):1268-1277, November 1985.
- Fickas, S. *Automatic Goal-directed Program Transformation.* Proc. 1st Nat. Conf. Artif. Intell., Stanford, 1980.
- Green, C. C. *The Design of the PSI Program Synthesis System.* Proceedings IEEE 2nd Int'l Conf. on Software Engineering, Long Beach, Calif. 1976.
- Kant, E. and Barstow, D. *The Refinement Paradigm: The Interaction of Coding and Efficiency Knowledge in Program Synthesis.* IEEE T. Software Engineering 7(5), 1981.
- Kant, E. *On the Efficient Synthesis of Efficient Programs.* Reprinted in *Artificial Intelligence and Software Engineering*, Rich and Waters, Morgan Kaufman, 1986.
- Kant, E. *Efficiency Considerations In Program Synthesis.* PhD Dissertation, Stanford, 1979.
- Neighbors, James M. *The Draco Approach to Constructing Software from Reusable Components.* IEEE T. Software Engineering 10(5), 1984. Also reprinted in *Artificial Intelligence and Software Engineering*, Rich and Waters, Morgan Kaufman, 1986.
- Standish T., Harriman D., Kibler D., and Neighbors, J. *The Irvine Program Transformation Catalog.* Dept. of Computer Science, UC Irvine, Calif. 1976.
- Rich, Charles and Waters, Richard C. *Automatic Programming: Myths and Prospects.* IEEE Computer, 21(8):40-51, August 1988.

5.2 Structure Oriented Systems Bibliography

- Allison, Lloyd. *Syntax Directed Program Editing.* Software Practice and Experience, Vol 13(5) 1983. pp. 453-465.
- Budinsky, Frank J. *SRE: A Syntax Recognizing Editor.* Software Practice and Experience, Vol 15(5):489-497. May 1985.
- Donzeau-Gouge, V., Huet G., Kahn G., Lang B. *Programming Environments Based On Structured Editors: The MENTOR Experience.* Interactive Programming Environments, pp. 128-140. McGraw Hill, New York, 1984. ISBN 0-07-003885-6.
- Feiler, Peter H. *A Language Oriented Interactive Programming Environment Based on Compilation Technology.* PhD Dissertation, Carnegie-Mellon University, Pittsburgh. May 1982.
- Habermann, A. Nico, and Notkin, David S. *Gandalf: Software Development Environments.* IEEE T. Software Engineering 12(12):1117-1127, December 1986.

- Hansen, Wilfred J. *Creation of Hierarchic Text With A Computer Display.* PhD Dissertation, Stanford University. June 1971.
- Medina-Mora, Raul. *Syntax Directed Editing: Towards Integrated Programming Environments.* PhD Dissertation, Carnegie-Mellon University, Pittsburgh. March 1982.
- Medina-Mora, Raul., and Notkin, David S. *ALOE Users' and Implementors' Guide.* CMU-CS-81-145, Carnegie-Mellon University. November 1981.
- Notkin, David S. *Interactive Structure Oriented Computing.* PhD Dissertation, Carnegie-Mellon University. 1984.
- Reps, Thomas W. *Generating Language Based Environments.* PhD Dissertation, Cornell University. 1983.
- Reps, Thomas W., and Teitelbaum, T. *The Cornell Program Synthesizer: A Syntax-Directed Programming Environment.* CACM 24(9):563-573, September 1981.
- Reps, T., Teitelbaum, T., and Horowitz, Susan. *The Why and Wherefore of the Cornell Program Synthesizer.* Proceedings of the ACM SIGPLAN/SIGOA Symposium on Text Manipulation. Portland, Oregon. June 1981. pp. 8-16.

5.3 Knowledge Based Systems Bibliography

- Chapman, David. *A Program Testing Assistant.* MIT AI Memo 651, November 1981.
- Rich, Charles. *A Formal Representation for Plans in The Programmer's Apprentice.* Reprinted in *Artificial Intelligence and Software Engineering*, Morgan Kaufman, Los Altos, California. 1986.
- Rich, Charles and Waters, Richard C. *Abstraction, Inspection, and Debugging in Programming.* MIT AI Memo 634. June 1981.
- Rich, C. and Shrobe, H. *Initial Report on a Lisp Programmer's Apprentice.* IEEE T. Software Engineering Vol 4(6):456-467. November 1978.
- Rich, Charles. *Inspection Methods in Programming.* MIT AI-TR-604, June 1981.
- Rich, Charles, and Waters, Richard C. *The Programmer's Apprentice Project: A Research Overview.* MIT AI Memo 1004, November 1987.
- Rich, Charles and Waters, Richard C. *Toward a Requirements Apprentice: On the Boundary Between Informal and Formal Specifications.* MIT Artificial Intelligence Laboratory, AI Memo 907, July 1986.

- Waters, Richard C. *KBEmacs: A Step Toward the Programmer's Apprentice.* PhD Dissertation, MIT AI-TR-753. May 1985.
- Waters, Richard C. *The Programmer's Apprentice: A Session with KBEmacs.* IEEE T. Software Engineering 11(11):1296-1320, November 1985.
- Waters, Richard C. *The Programmer's Apprentice: Knowledge Based Program Editing.* IEEE T. Software Engineering 8(1):1-12. January 1982.

5.4 Software Reuse Systems Bibliography

- Burton, Bruce A. et al. *The Reusable Software Library.* IEEE Software 4(4):25-33, July 1987.
- Gargaro, Anthony. and Pappas, T. L. *Reusability Issues and Ada.* IEEE Software 4(4):43-51, July 1987.
- Lenz, Manfred., Schmid, Hans A., and Wolf, Peter F. *Software Reuse through Building Blocks.* IEEE Software 4(4):34-42. July 1987.

5.5 Other Systems Bibliography

- Barstow, David R., Shrobe, Howard E., Sandewall, Erik. *Interactive Programming Environments.* McGraw Hill, New York, 1984. ISBN 0-07-003885-6.
- Basset, Paul G. *Frame Based Software Engineering.* IEEE Software 4(4):9-16, July 1987.
- Dart, Susan A., Ellison, Robert J., Feiler, Peter H., and Habermann, Nico A. *Software Development Environments.* IEEE Computer 20(11):18-28, November 1987.
- Dowson, Mark. *Integrated Project Support with IStar.* IEEE Software 4(6):6-15, November 1987.
- Harrison, William. *RPDE3: A Framework for Integrating Tool Fragments.* IEEE Software 4(6):46-56, November 1987.
- Hazel, Philip. *Development of the ZED Editor.* Software Practice and Experience, Vol 10(1):57-76. January 1980.
- Jameson, Kevin W. *SEE: A Software Engineering Environment.* University of Calgary, Dept. of Computer Science Research Report 88/316/28. 1988.
- *Multics Emacs Text Editor User's Guide.* Honeywell Inc., CH27-00. December 1983.

- *Multics Emacs Extension Writer's Guide.* Honeywell Inc., CJ52-01. July 1982.
- Peck, J.E.L. *Chef: A Versatile, Portable Text Editor.* Software Practice and Experience, Vol 11(5) 1981. pp. 467-477.
- Stallman, Richard M. *EMACS: The Extensible, Customizable, Self-Documenting Display Editor.* In *Proceedings of the ACM SIGPLAN SIGOA Symposium on Text Manipulation*, Portland, Oregon, June 1981. pp. 147-156.
- Stallman, Richard M. *GNU Emacs Manual, Sixth Edition, Version 18.* Free Software Foundation, Cambridge, Massachusetts. March 1987.
- Sterpe, Peter J. *TEMPEST: A Template Editor for Structured Text.* MSc Thesis, MIT AI-TR 843. June 1985.
- *Symbolics Text Editing and Processing Manual.* Symbolics Inc., Concord Massachusetts. July 1986.
- Teitelman, Warren, and Masinter, Larry. *The Interlisp Programming Environment.* IEEE Computer 14(4):25-34, April 1981.
- Walker, Janet H., Moon, David A., Weinreb, Daniel L., and McMahon, Mike. *The Symbolics* Genera *Programming Environment.* IEEE Software 4(6):36-45, November 1987.
- Weinreb, D., and Moon, D. *The Lisp Machine Manual.* MIT Artificial Intelligence Library, 1981.
- Wilcox, R.R., Davis, A.M., Tindall, M.H. *Design and Implementation of a Table Driven, Interactive Diagnostic Programming System.* CACM 19(11), 1976. pp. 609-616.
- Wood, S.R. *Z: The 95% Program Editor.* Proceedings of the ACM SIGPLAN/SIGOA Symposium on Text Manipulation. Portland, Oregon. June 1981. pp. 1-7.
- Yonke, Martin D. *A Knowledgeable, Language-Independent System for Program Construction and Modification.* USC ISI/RR-75-42, October 1975.

Bibliography

[1] Balzer, R. *A 15 Year Perspective on Automatic Programming.* IEEE TSE 11(11), 1985.

[2] Balzer, R. *Transformational Implementation: An Example.* IEEE T. Soft. Engg. SE-7(1). 1981.

[3] Boehm, Barry W. *A Spiral Model of Software Development and Enhancement.* IEEE Computer 21(5):61-72. May 1988.

[4] Boehm, Barry W. *Software Engineering Economics.* Prentice-Hall, Englewood Cliffs, New Jersey. 1981.

[5] Brooks, Jr., Fredrick P. *The Mythical Man-Month.* Addison Wesley, Menlo Park, California. 1979.

[6] Burton, Bruce A. et al. *The Reusable Software Library.* IEEE Software 4(4):25-33, July 1987.

[7] Chapman, David. *A Program Testing Assistant.* MIT AI Memo 651, November 1981.

[8] Cheatham, Thomas E. *Reusability Through Program Transformations.* IEEE T. Soft. Engg. Vol 10(5), 1984. Also reprinted in *Artificial Intelligence and Software Engineering*, Rich and Waters, Morgan Kaufman, 1986.

[9] Dart, Susan A., Ellison, Robert J., Feiler, Peter H., and Habermann, Nico A. *Software Development Environments.* IEEE Computer Vol 20(11), November 1987. pp. 18-28.

[10] Fickas, Steven F. *Automating the Transformational Development of Software.* IEEE T. Soft. Engg. Vol 11(11), November 1985. pp. 1268-1277.

[11] Fickas, S. *Automtic Goal-directed Program Transformation.* Proc. 1st Nat. Conf. Artif. Intell., Stanford, 1980.

[12] Green, C. C. *The Design of the PSI Program Synthesis System.* Proceedings IEEE 2nd Int'l Conf. on Soft. Engg., Long Beach, Calif. 1976.

[13] Habermann, A. Nico, and Notkin, David S. *Gandalf: Software Development Environments.* IEEE T. Soft. Engg. 12(12), December 1986, pp. 1117-1127.

[14] Jameson, Kevin W. *A Model for the Reuse of Software Design Information.* University of Calgary, Dept. of Computer Science Research Report 88/321/33. 1988.

[15] Kaisler, Steven H. *Interlisp: The Language and Its Use.* John Wiley, New York, 1986. ISBN 0-471-81644-2.

[16] Kant, E. and Barstow, D. *The Refinement Paradigm: The Interaction of Coding and Efficiency Knowledge in Program Synthesis.* IEEE TSE 7(5), 1981.

[17] Kant, E. *On the Efficient Synthesis of Efficient Programs.* Reprinted in *Artificial Intelligence and Software Engineering*, Rich and Waters, Morgan Kaufman, 1986.

[18] Kant, E. *Efficiency Considerations In Program Synthesis.* PhD Thesis, Stanford, 1979.

[19] Lenz, Manfred., Schmid, Hans., and Wolf, Peter F. *Software Reuse through Building Blocks.* IEEE Software 4(4):34-42. July 1987.

[20] Medina-Mora, Raul. *Syntax Directed Editing: Towards Integrated Programming Environments.* PhD Dissertation, Carnegie-Mellon University, Pittsburgh. March 1982.

[21] Misra, Santosh K., and Jalics, Paul J. *Third Generation versus Fourth Generation Software Development.* IEEE Software 5(4):8-14. July 1988.

[22] Neighbors, James M. *The Draco Approach to Constructing Software from Reusable Components.* IEEE T. Soft. Engg. Vol 10(5), 1984. Also reprinted in *Artificial Intelligence and Software Engineering*, Rich and Waters, Morgan Kaufman, 1986.

[23] Reps, Thomas W. *Generating Language Based Environments.* PhD Dissertation, Cornell University. 1983.

[24] Reps, Thomas W., and Teitelbaum, T. *The Cornell Program Synthesizer: A Syntax-Directed Programming Environment.* CACM 24(9), September 1981. pp. 563-573.

[25] Rich, Charles. *Inspection Methods in Programming.* MIT AI-TR-604, June 1981.

[26] Rich, Charles and Waters, Richard C. *Toward a Requirements Apprentice: On the Boundary Between Informal and Formal Specifications.* MIT Artificial Intelligence Laboratory, AI Memo 907, July 1986.

[27] Rich, Charles, and Waters, Richard C. *The Programmer's Apprentice Project: A Research Overview.* MIT AI Memo 1004, November 1987.

[28] Stallman, Richard M. *EMACS: The Extensible, Customizable, Self-Documenting Display Editor.* In *Proceedings of the ACM SIGPLAN SIGOA Symposium on Text Manipulation*, Portland, Oregon, June 1981. pp. 147-156.

[29] Standish T., Harriman D., Kibler D., and Neighbors, J. *The Irvine Program Transformation Catalog.* Dept. of Computer Science, UC Irvine, Calif. 1976.

[30] Teitelman, Warren, and Masinter, Larry. *The Interlisp Programming Environment.* IEEE Computer 14(4):25-34, April 1981.

[31] Waters, Richard C. *The Programmer's Apprentice: Knowledge Based Program Editing.* IEEE T. Software Engineering 8(1):1-12. January 1982.

[32] Waters, Richard C. *The Programmer's Apprentice: A Session with KBEmacs.* IEEE T. Software Engineering 11(11):1296-1320, November 1985.

[33] Weinreb, D., and Moon, D. *The Lisp Machine Manual.* MIT Artificial Intelligence Library, 1981.