# Language Oriented Programming

M. P. Ward
Computer Science Department
Science Labs, South Rd
Durham, DH1 3LE

January 6, 1995

**Abstract**

This paper describes the concept of *language oriented programming* which is a novel way of organising the development of a large software system, leading to a different structure for the finished product. The approach starts by developing a formally specified, domain-oriented, very high-level language which is designed to be well-suited to developing "this kind of program". The development process then splits into two independent stages: (1) Implement the system using this "middle level" language, and (2) Implement a compiler or translator or interpreter for the language, using existing technology. The approach is claimed to have advantages for domain analysis, rapid prototyping, maintenance, portability, user-enhanceable systems, reuse of development work, while also providing high development productivity. We give an example where the method has been used very successfully (in conjunction with rapid prototyping) in the development of a large software system: the FermaT reverse engineering tool. A major benefit of this approach to software development, as compared to the usual sequential "waterfall model" is the speed with which products can be brought to market. This is due to "concurrent engineering": the effective overlap of development stages. Finally, the "middle out" development style is compared and contrasted with the more usual "top down", "bottom up" and "outside in" development methods.

KEYWORDS: Language Oriented Programming, Very High Level Languages, Domain Oriented Languages, Rapid Prototyping, User-Enhanceable Systems, Reuse.

## 1 Introduction

The problems of designing and developing large-scale software systems are well documented, yet much of the research in program development methods has been confined to toy programs and small systems. F. P. Brooks in [2] notes the following properties of large software systems which cause problems as these systems are developed and maintained using traditional methods:

**Complexity:** This is an essential property of all large pieces of software, "essential" in that it cannot be abstracted away from. This leads to several problems:

- There are often communication difficulties among a large team of developers and these can lead to product flaws, cost overruns and schedule delays;
- It may be difficult or impossible to visualise all the states of the system, and this makes it impossible to understand the system completely. The unvisualised states can lead to security loopholes, or unforseen side-effects when extending or modifying the system;
- It is difficult to get an overview of the system, so maintaining conceptual integrity becomes increasingly difficult;
- It is hard to ensure that all loose ends are accounted for;
- There is a steep learning curve for new personnel.

**Conformity:** Many systems are constrained by the need to conform to complex human institutions and systems, for example a wages system is greatly complicated by the need to conform to

current tax regulations.

**Change:** Any successful system will be subject to change as it is used:

- By being modified to enhance its capabilities, or even apply it beyond the original domain;
- Surviving beyond the normal life of the machine it runs on;
- Being ported to other machines and environments.

**Invisibility:** With complex mechanical or electronic machines or large buildings the designers and constructors have blueprints and floorplans which provide an accurate overview and geometric representation of the structure. For complex software systems there is no such geometric representation. There are several distinct but interacting graphs of links between parts of the system to be considered; including control flow, data flow, dependency, time sequence etc. One way to simplify these, in an attempt to control the complexity, is to cut links until the graphs become hierarchical structures [26]. However, even an accurate model or abstraction of the system may become unreliable as the system is enhanced and modified over a period of time.

In the mid 1980's a survey of 19 software development projects by Bill Curtis and his colleagues [8] produced two main findings:

- There is a thin spread of domain knowledge among software developers in most projects;
- Customer requirements are extremely volatile.

More recently, a survey of 23 software development projects in the narrower area of requirements definition [19] produced these specific findings:

- Requirements were invented, not elicited. In about two-thirds of the projects, there was a potential market but no customer. The "requirements" were actually preferences which were prioritised so that the low priority "requirements" could be abandoned if the schedule slipped;
- Most development is maintenance. System evolution is so common, that a development from scratch is the exception rather than the rule;
- Most specification is incremental. The customer is rarely able to provide a complete specification at any stage of the project;
- Domain knowledge is important;
- There is a gulf between developer and user. Few developers had adequate knowledge about the user's work. This led to major misunderstandings about the system's purpose;
- User interface requirements continually change.

In this paper we describe a novel way of organising the development of a large software system, leading to a different structure for the finished product. We use the term "language oriented programming" to describe this approach, since the first stage in this development method is the design of a formally specified, domain-oriented, very high level programming language. It should be stressed that one of the aims of the language design is to capture domain knowledge in a form in which it can be readily used by the programmers. Our thesis is that a suitable language is a good way to make domain knowledge available, and the effect of developing in such a language as the first stage in the development process, is to dramatically reduce the development effort required while increasing maintainability and enabling reuse.

We give some examples of successful system developments, where the final system structure involves such a "middle level" language. Even though these systems were not necessarily developed in a "middle out order" (i.e. designing the language first before proceeding with the development of the system and implementation of the language), the "language oriented" nature of the system has contributed to its success. We also describe one major development project (the FermaT tool) which used an explicit middle out development approach, in conjunction with rapid prototyping,

to achieve a highly successful result. Finally, we compare and contrast middle out development with "top down", "bottom up" and "outside in" development methods.

## 2 Language Oriented Programming

In the history of computer science, the greatest single gain in software productivity has been achieved through the development of high-level languages with suitable compilers and interpreters. The use of a high level language often allows a program to be implemented with an order of magnitude fewer lines of code than if everything was written in Assembler. In addition, these lines of code will typically be easier to read, analyse, understand and modify. Our experience with developing the FermaT program transformation system (see Section 2.4) suggests that there is another large factor of productivity gain to be achieved by developing a suitable *problem oriented* very high level programming language, and using this language to implement the software system. In the case of FermaT, the domain oriented language is $\mathcal{META}$WSL.

In addition to the benefits of smaller code size and increased readability, another benefit of high-level languages is that they encapsulate a great deal of *programming knowledge* in an easily usable form. For example, the programmer can let the compiler deal with subroutine call and return linking, procedure arguments, simple optimisations, and so on. The second aim of Language Oriented programming (in addition to the reduction in total code size) is for the domain oriented language to form a repository of *domain knowledge* in a form which is readily useable by programmers working in that domain. Common objects in the domain will also appear in the language, common operations in the domain will be readily available as language constructs, even though the implementation of these operations may be large and complicated. In the case of FermaT, the **foreach**, **ifmatch** and **fill** constructs in $\mathcal{META}$WSL enable a programmer to write complex program transformations in a few lines of code, leaving the system to deal with most of the details and the tricky special cases.

This approach (representing domain knowledge in the form of a programming language) should be compared to the IKBS (Intelligent Knowledge-Based System) of representing domain knowledge in the form of a rule-based system. Using a rule-bases system as the repository of domain knowledge gives rise to two problems: (1) The *knowledge elicitation problem*: transferring knowledge from the brains of domain experts into a collection of rules suitable for implementing in a rule-based system; and (2) Enabling programmers to extract and make use of the information in the repository. Much work has been done on the first problem, with some notable successes in the area of medical diagnosis and hardware and software fault diagnosis. These are areas where the human knowledge is readily expressable in the form of interacting rules, and where the software system under development makes direct use of the rule base to achieve its functions. In other areas, for example the development of transformation systems, it is difficult to see how the programmers could make use of a rule-based representation of domain knowledge, while a very high-level domain-specific language can certainly be used, and re-used, in large software development projects in that domain.

The first stage in a language oriented development is therefore a *language design*, providing a formal syntax and semantics for this language. A language consists of a set of primitive operations together with language constructs and specialised abstract data types.

Having completed the language design, the development of the system breaks down into two largely independent stages (which can be carried out in any order, or even in parallel):

1. Implement the software system in the new language;
2. Implement the language in some existing computer language, i.e. write a compiler or interpreter or translator for the language.

Either or both of these stages may benefit from a recursive application of the method. Such a recursive development will result in a series of "language layers" with lower-level languages at the bottom, and very high-level domain-specific languages at the top. Each level is implemented in

terms of the next lower level by a process of interpretation, compilation or translation (which may be a formal semantic-preserving transformation). Each interpretation, translation or compilation stage may involve optimisation at both the "source" and "target" language levels.

It is essential that all of the middle level languages should be formally specified; since it is the availability of a formal specification which allows the system development and language implementation to be carried out independently. It is also important that the languages should be conceptually simple, easy to parse (by humans and computers) and should benefit from the latest developments in programming language design and implementation.

## 2.1 Advantages of Language Oriented Programming

### *2.1.1 Separation of Concerns*

The method provides a complete separation of concerns between design issues, which are addresses in a domain specific language, and implementation issues, which are addressed in the implementation of the language, and are separated from the design of the system. In addition, by making use of recent research in programming language design and implementation, it should be possible to keep the language design simple yet powerful and expressive. This will greatly reduce the complexity of the system implementation.

### *2.1.2 High Development Productivity*

Our experience with FermaT, and the experiences from other projects, indicate that a system implemented using the language oriented method, as a series of language levels, ends up much smaller than an bottom up or top down implementation of the same system. This is due to the fact that with a problem-specific very high level language, a few lines of code are sufficient to implement highly complex functions. The implementation of the language is also kept small since only those features which are relevant to the particular problem domain need to be implemented.

The small size of the final system means that the total amount of development work required is reduced, without increasing the complexity of the system, and for the same or higher functionality. This leads to improved maintainability, fewer bugs, and improved adaptability.

The very high level language means that a small amount of code in this language can achieve a great deal of work. This has already been noted for general purpose high-level languages, where an order of magnitude increase in productivity has been recorded. So called "4GLs" (forth generation languages) were an attempt to achieve a similar increase in productivity by the development of general-purpose very high-level languages. These were less successful than anticipated, partly due to a lack of formal specification of syntax and semantics, and partly because they tried to be general purpose languages. One large financial organisation is currently planning to abandon the 4GL altogether, and attempt to maintain the 40 million lines of machine-generated COBOL instead!

Our experience shows that by restricting the language to a specialised domain, the hoped-for gains in productivity *can* be achieved.

### *2.1.3 Highly Maintainable Design*

Studies have shown that the most important factor affecting maintainability is the size of the software system: more lines of code will generally require more maintenance effort [17,30]. The small total size of a system produced by the language oriented approach will implies that it will be therefore be highly maintainable. In addition, major functions of the system are implemented as a few lines of code in an appropriate language: this means that bug fixing and enhancements are easy, and there is a reduced chance of an unexpected interaction with other parts of the system.

With traditional programming methods, many design decisions (such as the representation of a data object, the file structure, the algorithms used to implement high-level operations etc.) are

"spread out" through the code. It becomes very difficult for maintainers to determine all the impacts of a particular design decision, or conversely, to determine which design decisions led to this particular piece of code being written in this way. With language oriented development, the effects of a design decision will usually be localised to one part of the system. For example, the decision to use a particular algorithm will be localised to one procedure: the algorithm will be written in an appropriate language and will therefore be short and easy to understand. Similarly, the decision to implement or represent a data structure in a particular way would normally have repercussions throughout the code, while in this case the effects would be localised to one part of the interpreter or translator. Advocates of "modular design" make these same arguments and suggest that the solution is to localise each design decision to a single module [26]. But the more fundamental design decisions cannot always be captured in a module.

### *2.1.4 Highly Portable Design*

Porting to a new operating system or programming language becomes greatly simplified: only the middle language needs to be re-implemented on the new machine, the implementation of the system (written in that language) can then be copied across without change. This is especially the case where a hierarchy of language levels has been developed, starting with a middle level language, implementing it in terms of lower-level language(s), and using it as the basis for implementing higher-level, more domain and problem-specific language(s). In this case only the lowest level language will need to be ported to a different machine or operating system, and this will be a simple task. This is one reason for the high portability of the TEX program: once the WEB-to-PASCAL or WEB-to-C programs have been ported, the one megabyte `tex.web` source file can be copied across and compiled without change.

In the case of the FermaT tool, the lowest level translator and support library consists of 2–3,000 lines of LISP code. This translates from low-level $\mathcal{META}$WSL to LISP, all the rest of the system is written in $\mathcal{META}$WSL. To port the system to a new version of LISP, or even to a new base language such as C, only requires rewriting the lowest level translator: and this is a comparatively small task–in fact, the first version of the translator was written in less than three man days. The FermaT system is currently being ported from a Unix environment to a PC environment, using C rather than LISP as the implementation language.

The DataFlex database language has also been ported to many machines and operating systems, again this is possible because much of the DataFlex system is written in DataFlex which is a macro language built on a small core of primitive database functions and user interaction functions.

The advantage of portability is not exclusive to language-oriented programming, a similar advantage (for similar reasons) can be claimed for bottom-up development, and for developments using tools such as class libraries. However, it could be argued that a class library is in fact an example of a domain-specific language (albeit with a highly restricted syntax).

### *2.1.5 Opportunities for Reuse*

There is a great potential for re-use of the middle level languages for similar development projects. The languages encapsulate a great deal of "domain knowledge": including knowledge of which data types, operations and execution methods are important in this domain, and what are the best ways to implement them. This kind of knowledge is extremely useful for requirements elicitation for new systems [19], new system development [8] and program comprehension of the existing system [3]. Hence there will be good opportunities for reuse within the project and beyond, including reuse of the language for other similar projects. A well-designed language is generally much more reusable than a collection of functions, abstract data types or objects: to understand this fact, imagine writing a typical C program in Assembler, where the C compiler has been replaced by a large library of Assembler routines! One of the main advantages of a well-designed domain-specific language is the new programming constructs which can be combined, more-or-less orthogonally in

various ways. In the case of perl, the language provides a convenient notation for regular expression searches and associative array handling, while the programmer is freed from worrying about memory allocation and freeing. The **foreach** construct in $\mathcal{META}$WSL captures the intricate details of which components of a statement are "terminal statements", in such a way that the programmer can write programs which manipulate "terminal statements" without needing to know how precisely how to calculate them.

One area in which these opportunities for reuse are particularly valuable is that of sector-specific companies serving niche markets. These companies produce a range of software within a specific domain; with their competitive advantage coming from specialised knowledge of the domain, a speedy response to new product opportunities and a rapid turnaround of users' requests for enhancements.

A project which has recognised the value of capturing domain knowledge in the form of a language is the Draco project [10,25]. This aims to encourage the reuse of design information in future program development projects by the use of *domain languages* together with the recorded results of a domain analysis. The system under development is written in a number of different domain languages, these programs are refined into the languages of other domains, and ultimately into executable code. In contrast to the Draco approach, our approach uses a single domain for several related development projects, rather than several small domains for each project. Our contention is that the best representation of domain knowledge (for programming purposes) is the design and implementation of a domain-specific programming language. Since our domain languages are implemented programming languages, there is no need for refinement to an existing programming language.

#### *2.1.6 User Enhancable System*

A system built using a hierarchy of language levels will have a "top level" language which is highly domain-specific, very high level, and formally specified. This language will almost certainly be interpreted rather than compiled: a few lines of code in this language is sufficient to implement each of the operations of the software system, so the interpretation overhead is negligible. With a suitable interface, the user could be provided with a high degree of control over the functionality of the system: calling up and editing the code for specific functions and using the top level language as a powerful "macro language" which has access to every function of the system. Note that the implemented functions would be written in this language, and would therefore provide "templates" for the user to modify and enhance—to provide their own functions or extend the existing ones. At the lowest level, this provides a macro language for the user. A major problem with most "macro" and "query" languages is that they are horrible languages (according to Hoare's "Basic principles of language design", see Section 2.3). They are not formally specified, were not designed from scratch to be a full programming language, and usually were not designed by people trained in language design, or familiar with other languages. Another problem is that it is rare to find that *all* of the systems functions are available via the macro language. In contrast, the language provided by the language oriented development is actually used to implement the whole system: it will be a well-designed, fully tested language, and all the systems facilities are guaranteed to be available in the language. Customisation of the system will be trivial.

It is easy to give the user the power to enhance the system in various ways, writing their own functions in the top level language, or modifying the ones provided (cf QED, FermaT, emacs, see Section 2.5). The user can be given access to the source code for the whole system: this will be a "small" program or collection of small programs in a highly domain specific language. In the case of QED, the "source code" is about 3000 lines of interpreted q code to which I have added about 650 lines of code to implement my personal functions. The sparc executable for the interpreter of this very high-level language is nearly one megabyte of code!

The FermaT transformation system includes around one hundred transformations, each imple-

mented in METAWSL, and ranging in size from a few lines to a few pages of code. The users can construct their own transformations by composing existing transformations (such transformations are automatically guaranteed to be correct), or by writing new METAWSL procedures. Since FermaT is itself a program manipulation system, it is possible in this case to use the system to maintain and enhance its own source code.

The LaTeX typesetting system is written as a collection of TeX macros. The user is able to extend the system by writing their own macros or (more commonly) using "style files" of macros written by other people. In addition, knowledgeable users can modify the standard style files to achieve their own effects.

One danger with giving users full access to the source code of a system is that their "enhancements" may actually degrade or damage the system over a period of time. If serious damage has been caused, the user can always go back to a previous working version. More importantly though, even after many enhancements the total size of the system will still be small, so if a complete redevelopment (or reverse-engineering) of the system turns out to be necessary, this will be a fairly small task.

The highest level language can be made "safe" in the sense that any code written by the user will do something meaningful in terms of the problem domain (rather than crashing the system with an unhelpful error code! See Section 2.3). This is a reasonable requirement because the top-level language is restricted to a small problem domain and uses concepts and operations in that domain. For example, the q code interpreter used in QED will, by default, terminate any loop which executes more than 1,000 iterations. This is ample for the vast majority of operations, and the limit can be raised or removed if necessary. The result is that the user cannot hang up the editor by inadvertantly writing an endless loop. In the FermaT system, the user is able to construct new program transformations (meta-programs) by combining existing transformations, tests and so on. Provided all the editing operations invoked by the meta-program are carried out by existing transformations, the meta-program will itself be a valid transformation.

### 2.2 Problems and their Alleviation

The main problem with introducing this development method is that good language design is *hard*. It is a highly skilled task, requiring a good grasp of the problem domain, the system requirements, and the available options in terms of computer science technology. However, the benefits from a good design are enormous, and there are ways to alleviate the problems (see below).

It should be emphasised that the aim is not to "de-skill" the programming task, but rather the opposite: to enhance the abilities of a skilled designer and domain expert. The aim is to capture a useful body of domain knowledge in the form of a domain-specific language which can be used by a skilled programmer to develop a powerful and useable system in a highly productive manner. With a user-enhancable system, the skill level required of the "knowledgeable user" is then *reduced*, since the implementation language uses familiar domain concepts, and the implementation itself provides "templates" which show how to use the language to achieve various results.

A potential problem is that writing in a very high-level language can "distance" the programmer from certain efficiency constraints: the programmer needs to understand the efficiency implications of various constructs in the language.

There may be management issues to address, for this development method to be introduced successfully. For example many programmers want to start designing and coding the system and are unfamiliar with the kind of high-level abstract thinking about the problem domain, which is required to produce a good middle level language design.

In mathematics, finding a good notation can take you half way to a good solution; for example tensor notation and calculus notation have enabled mathematicians to solve problems which would be extremely difficult in the older notations. Similarly, in computer science it is a great advantage

to have a suitable notation in which to express certain classes of algorithms, rather then writing yards of source code. An example is Bird's notation for functions acting on sequences [1].

Good language design breeds better language design: once the appropriate skills have been built up, they can be applied to many other developments.

Another potential problem is that by designing your own programming language, you will have to design all the tools needed to support programming and debugging in that language. This problem can be greatly alleviated with the use of suitable tools: there are several public domain and commercial packages available to assist with building language parsers, compilers and interpreters. A simple interpreted can provide a good debugging environment, while a translator to an existing language (such as C) will make available the debugging tools for that language—provided the structure of the translator output roughly matches the structure of its input. Low-level optimisations, constant propagation, loop unrolling, procedure inlining, constant subexpression elimination etc., can be safely left to the C compiler.

### 2.3 The Middle Layer

The middle layer language design may be imperative, functional, object-oriented, lazy-evaluated or whatever is most appropriate to the particular problem domain. But it is essential that the language should be precisely specified in both syntax and semantics; preferably in some formal notation such as set theory and logic. It should also be noted that Hoare's four "Basic principles of language design" [12] are as valid today as they were in the early 1960's:

1. Security: Every syntactically incorrect program should be rejected by the compiler, interpreter or translator, and executing any syntactically correct program should produce a result or an error message expressed in terms of the source code. In other words there should be no core dumps or "Segmentation Violations". This implies, for instance, that every array subscript access is checked at every occurrence. It also implies some sensible restrictions on the use of pointers. Hoare comments: *'In any respectable branch of engineering, failure to observe such elementary precautions would have long been against the law'*;
2. Brevity of object code and compactness of run time working data: Although the cost of high-speed memory has fallen dramatically since the early 1960's, it is still true that processors are cheap in comparison with the amount of main store they can address, and backing store is still many orders of magnitude slower. The programmer can nearly always take advantage of "spare" capacity to increase the program's quality, simplicity, ruggedness and reliability. For the case of a system constructed as a series of language layers, this principle is even more vital since the profligacy of each layer is compounded with the others;
3. Entry and exit codes for procedures and functions should be as compact and efficient as for tightly coded machine code subroutines: more generally, there should be no impediments to the use of convenient high-level facilities in the language. In the case of the GNU C compiler gcc [32] this point is taken a stage further with a compiler optimisation which can automatically select procedures and functions to be turned into inline code: with *no* calling overhead;
4. The language should be parsable in a single pass with a simple recursive-descent parser, and the compiler should be able to compile quickly. There are several reasons for this:
   - The language is to be read by *people*—who prefer to read something in a single pass if possible!
   - Even with modern multi-window workstations, compilation time is generally wasted time;
   - It is much easier to ensure the correctness of the compiler if the structure of the compiler closely follows the structure of the language. This is clearly the case for a recursive-descent parser.

For an interpreted language, rather than compiled or translated language, the need for rapid parsing by both human and machine is just as great. For the higher level languages, those which are highly domain specific, the efficiency of interpretation will be much less important. This is because all the *real* work is done by calling lower-level language components. However, if the language is to form the basis of a user-enhanceable system, then a simple and easily-understood syntax is essential.

Textual redundancy, for example requiring all variables to be declared, and the types of procedure and function arguments to be declared, is the best protection against programming or typing errors which can be extremely expensive to detect in a running program—and even more expensive not to![1]

Although good language design is a highly skilled task, the best languages are often produced by a single person or small team: the high skill levels do not have to be present in the whole development team. (cf the "Chief Programmer Teams" approach to programming [20]) The language oriented approach maximises the skill of the language designer and encapsulates the knowledge of the domain expert, making these available to the whole development team.

The language designers should ideally receive training in the design and implementation of as many existing programming languages as possible. They will then be able to learn from other's mistakes and stand on shoulders not toes![2] The designers should have a basic grasp of many different languages which use different paradigms: imperative, procedural, functional, higher-order functions, parallel languages, communicating processes, object-oriented languages, etc. They will also need training in program specification notations: especially set theory and first order logic. This is because a precise formal specification of the middle level language is essential to the success of the method. Finally, there needs to be training in language implementation and program analysis issues: providing a basic understanding of the implementation cost and complexity of various language features and how they interact (eg. mixing lazy evaluation with destructive update operations on lists can lead to incomprehensible programs).

The language will typically include not just procedures, functions, objects and/or ADTs but also new *constructs* which encapsulate aspects of the problem domain. For example, the $\mathcal{META}$WSL language used in FermaT includes a construct for iterating over the "reachable terminal statements" of the currently selected program. This construct enables the complex editing operations required to define certain program transformations to be written in just a few lines of code. Another example is the "EVENT" procedures in QED which are executed when certain conditions arise (for example, `EVENT_JUMP_FONT_NUMB` is called when the current editing position is moved to a character in a different font). In general, the data types, operations and constructs of the language will be a mixture of general purpose and domain specific constructs.

The LaTeX document typesetting system essentially defines a "typesetting language" with commands for section headings, mathematics, itemised and enumerated lists, and references to other parts of the document. This language encapsulates a great deal of knowledge about high quality typesetting of documents, including how to calculate the exact spacing between items, how to break a paragraph into lines, how to lay out a complex mathematical equation, and so on.

### 2.4 FermaT: A Case Study

FermaT is a program transformation system based on the theory of program refinement and equivalence developed in [35,38] and applied to software development in [28,37] and to reverse engineering

[1] An omitted comma in a FORTRAN `DO` statement caused the loss of the Mariner Venus probe when the statement was silently re-interpreted as an assignment to a new variable.

[2] There is an oft-quoted comment about programmers: *'Newton said "If I have seen farther, it is only because I have stood on the shoulders of giants". The problem with programming is that everyone's stepping on each others toes!'*

in [39,40]. The transformation system is intended as a practical tool for software maintenance, program comprehension, reverse engineering and program development.

The first prototype transformation system, called the "Maintainer's Assistant", was written in LISP [4,41]. It included a large number of transformations, but was vey much an "academic prototype" whose aim was to test the ideas rather than be a practical tool. In particular, little attention was paid to the time and space efficiency of the implementation. Despite these drawbacks, the tool proved to be highly successful and capable of reverse-engineering moderately sized assembler modules (up to 80,000 lines) into equivalent high-level language programs.

The system is based on semantic preserving transformations in a wide spectrum language (called WSL). The language includes both low-level programming constructs and high-level non-executable specifications. This means that refinement from a specification to an implementation, and reverse-engineering to determine the behaviour of an existing program can both be carried out by means of semantic-preserving transformations within a single language.

For the next version of the tool (i.e. FermaT itself) we decided to extend WSL to add domain-specific constructs, creating *a language for writing program transformations.* This was called $\mathcal{META}$WSL. The extensions include an abstract data type for representing programs as tree structures and constructs for pattern matching, pattern filling and iterating over components of a program structure. The "transformation engine" of FermaT is implemented entirely in $\mathcal{META}$WSL. The implementation of $\mathcal{META}$WSL involves a translator from $\mathcal{META}$WSL to LISP, a small LISP runtime library (for the main abstract data types) and a WSL runtime library (for the high-level $\mathcal{META}$WSL constructs such as **ifmatch**, **foreach**, **fill** etc.). One aim was so that the tool could be used to maintain its own source code: and this has already proved possible, with transformations being applied to simplify the source code for other transformations! Another aim was to test our theories on language oriented programming: we expected to see a reduction in the total amount of source code required to implement a more efficient, more powerful and more rugged system. We also anticipated noticeable improvements in maintainability and portability. These expectations have been fulfilled, and we are achieving a high degree of functionality from a small total amount of easily maintainable code: the current prototype consists of around 16,000 lines of $\mathcal{META}$WSL and LISP code, while the previous version required over 100,000 lines of LISP.

The FermaT design is based on a recursive application of language oriented programming, involving two "layers" of domain-specific languages. These are:

1. A fairly high-level, general purpose language, based on the executable constructs of WSL [35, 38] together with an abstract data type (ADT) for recording, analysing and manipulating programs and fragments of programs. This is implemented in LISP, using a WSL to LISP translator together with a "LISP runtime library" of functions and procedures to implement the ADT. The ADT includes facilities for loading and saving programs, moving around (it records the "current selection" within the current program), and editing operations. This consists of less than 2000 lines of LISP: the entire rest of the transformation engine is written in WSL and $\mathcal{META}$WSL code. Hence porting the system to another language (and a C version is currently under development) would entail writing at the most a few thousand lines of code;
2. On top of the "base" WSL language we have implemented a very high-level, domain-specific language for writing program transformations, called $\mathcal{META}$WSL. This includes high level constructs which do most of the work involved in writing transformations: see below for an example. $\mathcal{META}$WSL is implemented almost entirely in WSL; there are a few extensions to the WSL to LISP translator and a number of WSL libraries which are compiled into LISP and linked to the translated $\mathcal{META}$WSL.

$\mathcal{META}$WSL encapsulates much of the expertise developed over the last 10 years of research in program transformation theory and transformation systems. As a result, this expertise is readily available to the programmers, some of whom have only recently joined the project. Working in

$\mathcal{META}$WSL, it takes only a small amount of training before new programmers become effective at implementing transformations and enhancing the functionality of existing transformations.

### *2.4.1 The FOREACH Construct*

As an example of a high-level construct in $\mathcal{META}$WSL we will consider two variants of the **foreach** construct. A **foreach** is used to iterate over all those components of the currently selected item which satisfy certain conditions, and apply various editing operations to them. The construct takes care of all the details, when for example, components are deleted, expanded or otherwise edited. Consider the following procedure (which is the implementation of a transformation taken from the FermaT system):

```
proc @Delete_All_Skips_Code(Data) ≡
  foreach Statement do
    if @Spec_Type(@Item) = Skip
      then @Delete fi od.
```

The purpose of this transformation is to delete all occurrences of **skip** statements in the currently selected item. Since **skip** has no effect, the transformation is clearly valid. However, there are various syntactic considerations as shown by the following examples:

| Before | After |
|---|---|
| **while B do skip od** | $\{\neg$**B**$\}$ |
| **if B then skip**<br>**else** x := 0 **fi** | **if** ¬**B then** x := 0 **fi** |
| **do skip od** | **abort** |
| **var** x := 0 :<br>**if B then skip fi end**;<br>y := 0 | y := 0 |

Another variant of the **foreach** construct iterates over all the *simple terminal statements*. These are the components of a statement which when executed will cause termination of the statement. See [35,36,38] for a detailed definition. In the following program, the three simple terminal statements are marked with an asterisk (∗):

```
last := " "; line := " "; i := 1;
line := item[i] ++ " " ++ number[i];
if i ⩾ n
  then do do last := item[i];
            i := i + 1;
            if i = n + 1 then write(line); exit(2) * fi;
            if item[i] ≠ last
              then write(line); exit(1) *;
                   if i = j then exit(2) fi
              else line := line ++ ", " ++ number[i] fi od;
       line := item[i] ++ " " ++ number[i] od
  else skip * fi
```

Note that the second occurrence of **exit**(2) is *not* considered a terminal statement because it is not reachable. This is because it occurs as part of a statement which follows an **exit** statement.

A typical transformation which involves finding all simple terminal statements is "Absorb Right". Suppose the previous program is followed by the statement:

```
if item[i] = error then exit fi
```

This statement can be "absorbed" into all the terminal positions of the preceding statement to give the following equivalent version:

```
last := " "; line := " "; i := 1;
line := item[i] ++ " " ++ number[i];
if i ⩾ n
  then do do last := item[i];
              i := i + 1;
              if i = n + 1 then write(line);
                                if item[i] = error then exit(3) else exit(2) fi fi;
              if item[i] ≠ last
                then write(line);
                     if item[i] = error then exit(2) else exit fi;
                     if i = j then exit(2) fi
                 else line := line ++ ", " ++ number[i] fi od;
         line := item[i] ++ " " ++ number[i] od
  else if item[i] = error then exit fi fi
```

Note that the absorbed statement has to be "incremented" (by having its **exit** statements increased in value, and an **else** clause added with an appropriate **exit**) when it is inserted into one or more loops. The unreachable **exit**(2) has not been modified.

If the selected statement is part of an *action system*, a collection of mutually recursive parameterless procedures, [35,39], then further complexities arise. Some action calls may not return (because the action or a subsequent one called the special action Z which causes immediate termination of the whole action system). In a *regular* action system (where each action terminates with another action call), no action calls will return and nothing must be absorbed after an action call. In a *hybrid* action system, this is only true for a **call** Z.

All these complexities and special cases are dealt with using the high-level features of $\mathcal{META}$WSL: the **foreach** STS structure finds all the simple terminal statements, @Increment deals with moving statements to a lower depth, and @Gen_Improper? tests whether the insertion is actually required or not for each case.

The transformation can therefore be implemented in the following few lines of code (which again are taken directly from the FermaT system):

```
proc @Absorb_Right_Code(Data) ≡
  @Right; @Cut; @Left;
  foreach STS do
    if Depth = 0 ∨ (@Spec_Type(@Item) = Exit ∧ @Value(@Item) = Depth)
      then if @Spec_Type(@Item) = Exit ∧ Depth > 0
             then @Splice_Over(@Increment(@Buffer, AS_Type, Depth, 0))
           elsif @Gen_Improper?(@Item, AS_Type)
             then skip
           elsif @Spec_Type(@Item) = Skip ∨ @Spec_Type(@Item) = Exit
             then @Paste_Over(@Buffer)
              else @Paste_After(@Buffer) fi fi od.
```

### 2.5 Examples of a Language Oriented Programming

Although we have described LOP as a novel approach, we can find examples of several successful, large scale software development projects which use a middle language layer. These were not necessarily developed in middle out order (by designing the language first) and do not all provide formal specifications of the middle language.

### *2.5.1 Simulation Languages*

One of the first areas where the value of domain-specific languages was recognised—especially languages with domain-specific *constructs*—was Monte Carlo simulations. Montecode [13] is an interpreted language for writing Monte Carlo simulations. Its constructs include random sampling from distributions, management of queues, building histograms and event to event scanning. CSL (Control and Simulation Language) [5] was designed for use in the field of complex logical problems. It uses set operations and specialised constructs, including iteration over the elements of a set and finding an element in a set which meets various criteria. CSL is "compiled" into FORTRAN, with a ratio of CSL to FORTRAN statements of the order of 1 to 5. The ratio of time spent in writing similar programs in CSL and FORTRAN was also of the order 1 to 5 [5]. The users also reported that "several problems which had not previously been tackled due to difficulty in formulation have now been formulated with little trouble."

### *2.5.2 The QED Word Processor*

QED is a powerful mouse-based word processor which is implemented using the specially designed "q" language: an interpreted, functional high-level language which has special data types and operations for document manipulation [29]. All the function key actions, menus, mouse operations etc. are implemented as q functions with the source code supplied: generally only a few lines of code are required to implement each function. The whole system consists of less than 2,500 lines of q code together with the interpreter.

### *2.5.3 The emacs Text Editor*

The GNU text editor emacs [31] is implemented in a version of LISP, and comes with a compiler and interpreter for emacs LISP. As with QED, each function key is bound to an emacs LISP function. In this case, the system is supplied with over 150,000 lines of emacs LISP code which define "editing modes" for various types of file, emulators for other text editors, and many other facilities.

### *2.5.4 The TEX and LATEX Typesetting Programs*

Knuth's TEX typesetting program [15] was written in the WEB literate programming language [14] whose aim was to remove some of the deficiencies of PASCAL and allow source code and documentation to be intertwined in the same source file. TEX is implemented as a small set of primitive typesetting operations, together with a macro processor. The *plain* set of TEX macros were designed by Knuth to form a basic typesetting package. A more extensive set of macros, built on top of a modified *plain* package, forms the basis for the LATEX package [16]. This package allows the user to concentrate on the structure of the text rather than on formatting commands. In effect, LATEX implements a "structured typesetting language" which the author uses to implement his or her document, see Section 2.3.

The LATEX3 project, which is currently developing a new version of LATEX, is planning yet another language level, built on top of the TEX macro language. As the following quote from the development team illustrates, they are adopting an explicitly "middle out" style for this new development:

> Another important interface will be the LATEX3 programming language, used for producing enhancements and extensions: it will be an entirely new language based on data structures and operations suited to the kind of programming required by document processing applications and to the expression of visual components of the layout process. Built on this language there will be high-level generic functions that allow the straightforward expression of common layout components.

### *2.5.5 Perl*

Larry Wall's perl [34] is an interpreted language optimised for scanning arbitrary text files, extracting information from those text files, and printing reports based on that information. It's also a good language for many system management tasks. Perl encapsulates part of the domain of text file processing: including pattern searching and replacement, arbitrarily-sized strings, arrays and "associative arrays" and also provides a uniform access mechanism to various operating system functions. The effect is that the perl programmer can write highly portable, moderately efficient code, without having to worry about memory allocation and freeing, hash table implementation, optimised pattern matching, incompatibilities between operating systems, and similar details. Perl users have reported over an order of magnitude increase in productivity over C or $C^{++}$ for text and binary file processing applications and prototyping simple user interfaces.

### *2.5.6 Databases*

In the area of database programs, a "middle-level" language (SQL) has already been developed. The user can in theory purchase a database program based on SQL, to be used with an SQL "engine" produced by a different company.

The DataFlex database language is designed as a macro processor on top of a simple language. The language has some very high-level features for database operations and the interactive manipulation of on-screen data. Writing a simple multi-user interactive database requires only a few lines of code, and much of the DataFlex system software is itself written in the DataFlex language. The language has been implemented in both PASCAL and C, and has been ported to a wide range of systems.

### *2.5.7 Visual Basic for Applications*

The latest version of the Excell spreadsheet makes use of a very high level programming language called Visual Basic for Applications.

### *2.5.8 The FermaT Program Transformation System*

The FermaT tool, implemented in $\mathcal{META}$WSL, translated to LISP, (a very high level language designed for implementing program transformation systems). $\mathcal{META}$WSL has language constructs for pattern matching and filling of program fragments (schemas) and constructed for looping over all statements, terminal statements, free variables etc. See Section 2.4 for details of this project.

## 2.6 Other Potential Applications

The method thus seems to be very suitable for editors, word processors, interactive databases, and interactive systems generally. In addition, it seems likely that accounting software could benefit from a suitable domain-based language, especially where frequent changes are required to keep pace with changing tax regulations.

### *2.6.1 Spreadsheets*

In this section we are discussing the development of a large spreadsheet (for example, a financial model for a company), *not* the development of a spreadsheet package such as Excell.

A spreadsheet program can be thought of as a high level language for developing financial models, cash flow forecasts, and other applications which make use of figures arranged in rows and columns.

A typical spreadsheet program has a fairly powerful input language, with facilities for entering equations into a whole row or column, and high-level facilities for manipulating rows, columns and blocks of data. However, once a spreadsheet has been created, it is stored in the form of an array

of numbers, strings and formulae. All the "structure", which indicates which cells belong together and how groups of cells are related to each other, has been lost. Any changes have to be made at the cell level, by imposing a new structure, or trying to remember the original structure. Errors, such as updating only part of a column of formulae, can be very time-consuming to track down. The effect is as if the user had entered a "program" (for example, a large and complex financial model) in a high-level language, which the system then immediately "compiles" into the low-level language of cells and contents, throwing away the source code! All changes have to be made by "patching the object code", rather than by updating the source code and re-compiling.

There is therefore a need for one or more domain-specific languages which work at a higher level of abstraction than the typical spreadsheet: for example a language for defining and exercising financial models. A typical program in this language will take the raw accounts data and produce cash flow forecasts, balance sheet projections etc.

## 3 Comparison With Other Methods

### 3.1 Top Down Development

The "top down" program development method, also called "stepwise refinement" has been advocated by many people as a way of mastering the complexity of large programs by constructing them according to a rigid hierarchical structure. The method starts with a high-level description of the system to be developed: which may be an informal description or a formal specification. The specification is refined into a top level structure with "gaps" (which again may be formal specifications or informal descriptions). Each gap is treated in turn, its structure is elaborated and its function re-expressed in terms of operations with simpler functionality. The idea is that only a small part of the program is worked on at each stage, there are only a few details to keep track of so it is easy to ensure that each structure is correct. If each component is elaborated correctly at each stage then (in theory) the whole program will be correct by construction. This method has been used successfully with some small examples [9,42,43,44]; however, some severe problems emerge as larger programs are attempted.

If the specifications are given informally, then there is a serious danger of ambiguity and misinterpretation. For example, the user of a specification could interpret it to have one meaning, while the implementor interprets it with a different meaning. This has led to the publication of an incorrect program [11]. If a formal specification language is used at every stage in the development, then in theory, such ambiguities cannot arise and this method leads to a valuable "separation of concerns" between the user and implementor of each specification. The implementor should not need to know anything about the program in which the specification is used, and the user should not need to know anything about how the specification is implemented (although in practice there may be efficiency concerns). The "Refinement Calculus" of Morgan et al [21,22,23] is an example of this approach to software development. It has been used very successfully in the development of a number of small programs from specification to implementation [21].

The major problem with this approach is that the method itself provides no clue as to what the top level structure should look like in a particular case. This is not a problem for well-understood "toy" programs, for which the top level structure is obvious. But for large programs, choosing the wrong structures in the initial stages can have serious repercussions which will only be uncovered much later in the development. If a serious mistake is made then the whole development will have to be scrapped and repeated from that point on—and such mistakes are most likely in the early stages of development.

An example of this problem is described in [27] and [18]. Three case studies were designed to evaluate how suitable different flavours of object-oriented analysis were for different types of system. One of the case studies was a real project to develop the engagement system for the Tomahawk missile. The developers found that they could divide missiles into subclasses according to their

warhead and navigational properties. They could also be categorised according to whether they were tactical or exercise missiles. These two taxonomies were almost entirely orthogonal, and this suggested that they use multiple inheritance to make use of both categorisations. However, later on in the development it was found that the orthogonality broke down: although some types of exercise missiles can carry warheads, nuclear missiles cannot. Continuing with the original top-level design would have produced inelegant and artificial models, with the exceptional case buried in a mass of details and easy to miss.

### 3.2 Bottom Up Development

A bottom up development starts by implementing the lowest level general purpose "utility" routines. These are used to implement higher level routines, abstract data types and so on. The higher level routines and abstract data types will be increasingly domain specific and problem specific. Eventually, the top level structure of the program can be implemented.

One advantage of this method is that unit testing and (partial) integration testing can be carried out at each stage of the development. The performance of the various subsystems can be monitored throughout the development process. Also, the general-purpose and domain-specific routines might be reused between different development projects. A later development project in the same domain will presumably be able to make use of much of the previous work.

However, it may be difficult, especially in the middle stages of the development, to determine precisely what to build next in order to ensure that real progress is being made. The high-level routines may turn out to implement the wrong functions: for example, they may do "too much" or may not do what is really required. Some may not be needed at all. In each of these cases there will be wasted development work. As with top down design, these problems are greatly exacerbated when the application is "novel" and there is no previous experience to indicate what kinds of high-level routines will be useful.

As for top down development, in the absence of formal specifications, the user of a routine may have a different interpretation of its behaviour than the implementor of the routine—and this can still be the case even when the user and implementor are the same person! Providing a concise formal specification for each routine to be developed will minimise this risk and allow a top down development of the routine.

### 3.3 Outside In Development

Outside in development is combination of top down and bottom up developments working in parallel: a team of designers start with the high level specification, which is refined towards code. At the same time another team of implementors (or, more commonly, the same team) start implementing useful utility routines and domain specific abstract data types. Hopefully the two developments will meet in the middle!

With this method, designers and implementers can start work straight away and work in parallel: this may be an advantage where there are a lot of people to be kept busy! The implementation team may be able to re-use work from previous projects.

Unfortunately, this design method appears to combine all the problems of top down and bottom up design (the worst of both worlds). There is the additional problem of getting the top down design and bottom up implementation to meet: there may be significant "gaps" between the two teams products, as well as "overlap" where both teams implement similar functions. For a large development project, there is no *a priori* reason to suppose that the set of "to be implemented" functions for a particular stage in the top down design, will be a particularly close match to the functions actually provided by a bottom up implementation.

### 3.4 Middle Out Development

The previous three sections have covered top down, bottom up and outside in development methods and suggested that they all have severe problems when scaled up to large software system developments. Top down decomposition does not work well until the analyst has an almost complete concept of the system. Bottom-up development is unlikely to work unless someone can tell you how the lower-level developments fit into the big picture. Outside-in development cannot be started until you have determined the boundaries of the system environment: but that is a major design decision.

The obvious missing method, in order to complete the pattern, is "Middle Out" development. But what does it mean to start in the middle of the abstraction hierarchy and work outwards to both higher and lower levels of abstraction? What is the "middle layer" of a hierarchical design?

Our claim is that this "middle layer" (which forms the starting point for middle out development) is a very high level language, or "abstract machine", specially designed to make it easy to implement the kind of software system required. "Middle Out" development is in fact another, rather less descriptive, name for language-oriented programming.

## 4 Rapid Prototyping

The "rapid prototyping" approach to requirements elicitation and software development can be combined with any of the methods discussed above. It is based on two observations:

1. It is much easier to design a large complex system "properly" if you have already built a similar system in the past: in effect, this is reuse of the designer's previous experience;
2. Users often have difficulties in articulating their precise requirements, but if they are given a system which does *not* do what they want, they will quickly find this out!

Rapid prototyping is a requirements elicitation and program design method in which a number of releases are planned over the development period. The initial release concentrates on implementing a few major functions, and a basic user interface, with the aim being to get something into the hands of the users as quickly as possible in order for them to start giving useful feedback. It does not matter if the first few releases are "quick hacks" since at least one later release will be a complete re-implementation from scratch (not necessarily reusing any actual code from the earlier prototypes). As the old saying has it:

*Plan to throw one away (you will anyway).*

Traditional program design has been called "slow prototyping".

### 4.1 Rapid Prototyping and Language Oriented Programming

If the language design principles discussed above are adhered to then the first language implementation will be a simple task. A first prototype of (parts of) the system can be produced very quickly with the aid of suitable tools such as user interface toolkits such as SUIT[3], a Simple User Interface Toolkit [6], and GARNET[4] [24]; and language development toolkits such as TXL[5] [7], and Eli[6] [33].

Experience with this prototype will provide valuable input to the language design stage for the next prototype. Giving the users a prototype design can be an excellent method of acquiring domain knowledge and uncovering poor design decisions (as the users gleefully point out the deficiences in the prototype!) This domain knowledge can then be captured in the design of the middle language

[3] Available via FTP from `uvacs.cs.virginia.edu` in directory `/pub/suit/distribution`

[4] From `a.gp.cs.cmu.edu` in directory `/usr/garnet/garnet`

[5] From `qusuna.qucis.queensu.ca` in directory `/txl`

[6] From `ftp.cs.colorado.edu` in directory `/pub/cs/distribs/eli`

for the next prototype implementation, and the experience gained from implementing and using a prototype is extremely valuable in the design of the middle layer language for the next prototype.

## 5 Conclusion

A combination of rapid prototyping with middle out development of each prototype would appear to be a good development approach for many large-scale software development projects. The language-oriented approach provides a handle on the problem of scale. There is a fundamental limit to complexity of any software system for it to be still manageable: if it requires more than "one brainfull" of information to understand a component of the system, then that component will *not* be understood fully. It will be extremely difficult to make enhancements or fix bugs, and each fix is likely to introduce further errors due to this incomplete knowledge. The best way to reduce the total size of a large system (and therefore to produce an understandable system) is to implement it in a very high-level language. In this way it may be possible to repeat the order of magnitude increase in productivity achieved by moving from Assembler language to high level programming languages. Current high-level programming languages "capture" knowledge about the *programming* domain: for example, the C programmer does not have to deal with the details of register allocation and subroutine parameter passing. The perl programmer does not have to deal with memory management and data type conversion. Higher level languages are required to capture *domain* knowledge: for example the LaTeX programmer does not have to deal with the complex rules concerned with laying out mathematical formulae, the $\mathcal{META}$WSL programmer does not have to determine which components of a statement are terminal statements, or how to combine the effects of many editing operations applied simultaneously to a complex program structure.

Thus the language-oriented approach to programming directly addresses the problems with large software systems, discussed by Brooks in [2] (see Section 1).

**Complexity:** The complexity of a language-oriented system is greatly reduced since the source code is divided into two independent sections:

1. The implementation of the system in a very high level, domain-oriented language;
2. A translator or interpreter for the language in (1).

The strongly domain-oriented nature of the middle level language means that complex functions of the system are implemented in just a few lines of code. The complexities of the domain concepts are encapsulated in the language in the form of abstract data types and new programming constructs. The *user* of the language can then concentrate on domain-level issues. For a large system, a "recursive" application of middle out development (involving several language layers) will reduce the overall complexity still further;

**Conformity:** The use of a domain-oriented language will make conformity easier to achieve, for example a wages system would be written in a language which uses the concepts present in the current tax regulations. This makes it easier to ensure that the system conforms to these complex requirements;

**Change:** The small size of the source code, and the domain-oriented nature of the language, means that enhancements to the system are trivial to make, and could even be carried out by the users themselves (see Section 2.1.6). In addition, porting to new operating systems, machines and even implementation languages is simplified (see Section 2.1.4);

**Invisibility:** There is still much scope for research on the visualisation of complex system designs, however, it is anticipated that a suitable high-level domain-oriented language will be able to hide much of the complexity within domain-specific objects, operations and language constructs. This should make it easier for the designer to ensure the correctness of his design by concentrating on the problem-specific aspects.

## Acknowledgements

This work was supported by the SERC (Science and Engineering Research Council) project "A Proof Theory for Program Refinement and Equivalence: Extensions". K. H. Bennett and J. N. Buxton made helpful comments on an earlier draft of the paper.

## References

[1] R. Bird, "Lectures on Constructive Functional Programming," Oxford University, Technical Monograph PRG-69, Sept., 1988.

[2] F. P. Brooks, "No Silver Bullet," *IEEE Computer* (Apr., 1987).

[3] R. Brooks, "Towards A theory of the Comprehension of Computer Programs," *International Journal of Man-Machine Studies* 18 (1983), 543–54.

[4] T. Bull, "An Introduction to the WSL Program Transformer," *Conference on Software Maintenance 26th–29th November 1990, San Diego* (Nov., 1990).

[5] J. N. Buxton & J. G. Laski, "Control and Simulation Language," *Comput. J.* 5 (1962), 194–199.

[6] M. J. Conway, *SUIT Reference Manual*, University of Virginia, 1992.

[7] J. R. Cordy & I. H. Carmichael, "The TXL Programming Language, Syntax and Informal Semantics - Version 7," Dept. of Computing and Information Science, Technical Report 93-355, Queen's University, Kingston, Canada K7L 3N6, June, 1993.

[8] B. Curtis, H. Krasner & N. Iscoe, "A Field Study of the Software Design Process for Large Systems," *Comm. ACM* 31 (Nov., 1988), 1268–1287.

[9] E. W. Dijkstra, *A Discipline of Programming*, Prentice-Hall, Englewood Cliffs, NJ, 1976.

[10] P. Freeman, "A Conceptual Analysis of the Draco Approach to Constructing Software Systems," *IEEE Trans. Software Eng.* SE-13 (July, 1987).

[11] P. Henderson & R. Snowdon, "An Experiment in Structured Programming," *BIT* 12 (1972), 38–52.

[12] C. A. R. Hoare, "The Emperor's Old Clothes: The 1980 ACM Turing Award Lecture," *Comm. ACM* 24 (Feb., 1981), 75–83.

[13] D. H. Kelly & J. N. Buxton, "Montecode—an Interpretive Program for Monte Carlo," *Comput. J.* 5 (1962), 88–93.

[14] D. E. Knuth, "Literate Programming," *The Computer Journal* 27 (1984), 97–111.

[15] Donald E. Knuth, *The TEXbook*, Addison Wesley, Reading, MA, 1984.

[16] L. Lamport, *LATEX A Document Preparation System*, Addison Wesley, Reading, MA, 1986.

[17] D. C. Littman, J. Pinto, S. Letovsky & E. Soloway, "Mental Models and Software Maintenance," in *Empirical Studies of Programmers*, E. Soloway & S. Iyengar, eds., Ablex, Norwood, NJ, 1986, 80–98.

[18] M. Lubers, C. Potts & C. Richter, "Developing Initial OOA Models," *Proc. Intl. Conf. Software Eng.*, Los Alamitos, Calif. (1992).

[19] M. Lubers, C. Potts & C. Richter, "A Review of the State of the Practice in Requirements Modeling," *Proc. Int'l Requirements Eng. Symp.*, Los Alamitos, Calif. (1993).

[20] A. Macro & J. Buxton, *The Craft of Software Engineering*, Addison Wesley, Reading, MA, 1987.

[21] C. Morgan, *Programming from Specifications*, Prentice-Hall, Englewood Cliffs, NJ, 1990.

[22] C. C. Morgan, "The Specification Statement," *Trans. Programming Lang. and Syst.* 10 (1988), 403–419.

[23] C. C. Morgan, K. Robinson & Paul Gardiner, "On the Refinement Calculus," Oxford University, Technical Monograph PRG-70, Oct., 1988.

[24] B. A. Myers, D. A. Giuse, R. B. Dannenberg, B. V. Zanden, D. S. Kosbie, E. Pervin, A. Mickish & P. Marchal, "Garnet; Comprehensive Support for Graphical, Highly-Interactive User Interfaces," *IEEE Computer* 23 (Nov., 1990), 71–85.

[25] J. M. Neighbors, "The Draco Approach to Constructing Software from Reusable Components," *IEEE Trans. Software Eng.* SE-10 (Sept., 1984).

[26] D. L. Parnas, "Designing Software," *IEEE Trans. Software Eng.* 5 (Mar., 1979).

[27] C. Potts, "Software-Engineering Research Revisited," *IEEE Software* (Sept., 1993).

[28] H. A. Priestley & M. Ward, "A Multipurpose Backtracking Algorithm," *J. Symb. Comput.* 18 (1994), 1–40.

[29] M. Raskovsky, *Qed Users's Guide*, Atelier de Software Ltd., Rookery Farm, Nye, Hewish, Avon, 1989.

[30] E. Soloway & K. Erlich, "Empirical Studies of Programming Knowledge," *IEEE Trans. Software Eng.* SE 10 (1984), 595–609.

[31] R. M. Stallman, *The EMACS Reference Manuel*, Free Software Foundation, Inc., 1988.

[32] R. M. Stallman, *Using and Porting GNU CC*, Free Software Foundation, Inc., Sept., 1989.

[33] W. M. Waite, "Beyond LEX and YACC: How to Generate the Whole Compiler," University of Colorado, Technical Report, Boulder, Colorado, 1993.

[34] L. Wall & R. L. Schwartz, *Programming Perl*, O'Reilly & Associates, Inc., Cambridge, MA, 1992.

[35] M. Ward, "Proving Program Refinements and Transformations," Oxford University, DPhil Thesis, 1989.

[36] M. Ward, "A Recursion Removal Theorem - Proof and Applications," Durham University, Technical Report, 1991.

[37] M. Ward, "Derivation of Data Intensive Algorithms by Formal Transformation," Submitted to IEEE Trans. Software Eng., 1993.

[38] M. Ward, "Foundations for a Practical Theory of Program Refinement and Transformation," Durham University, Technical Report, 1994.

[39] M. Ward, "Abstracting a Specification from Code," *J. Software Maintenance: Research and Practice* 5 (June, 1993), 101–122.

[40] M. Ward & K. H. Bennett, "A Practical Program Transformation System For Reverse Engineering," *Working Conference on Reverse Engineering, May 21–23, 1993*, Baltimore MA (1993).

[41] M. Ward, F. W. Calliss & M. Munro, "The Maintainer's Assistant," *Conference on Software Maintenance 16th–19th October 1989, Miami Florida* (1989).

[42] N. Wirth, "Program Development by Stepwise Refinement," *Comm. ACM* 14 (1971), 221–227.

[43] N. Wirth, *Systematic Programming: An Introduction*, Series in Automatic Computation, Prentice-Hall, Englewood Cliffs, NJ, 1973.

[44] N. Wirth, *Algorithms and Data Structures*, Prentice-Hall, Englewood Cliffs, NJ, 1986.