# SPARE Parts: A C++ Toolkit for String PAttern REcognition

Bruce W. Watson

Ribbit Software Systems Inc.
(IST Technologies Research Group)
Box 24040, 297 Bernard Ave.
Kelowna, B.C., V1Y 9P9, Canada

e-mail: watson@RibbitSoft.com

**Abstract.** In this paper, we consider the design and implementation of SPARE Parts, a C++ toolkit for pattern matching. SPARE Parts is the second generation string pattern matching toolkit from the Ribbit Software Systems Inc. and the Eindhoven University of Technology. The toolkit contains implementations of the well-known Knuth-Morris-Pratt, Boyer-Moore, Aho-Corasick and Commentz-Walter algorithms (and their variants).

The toolkit is publicly available (though it is not in the public domain and it may not be redistributed) for noncommercial use. A totally re-implemented toolkit, known as SPARE Parts II, is available for commercial licensing from Ribbit Software Systems Inc. In addition to the functionality of SPARE Parts, it contains approximate pattern matchers, regular pattern matchers, multi-dimensional pattern matchers, and a highly tuned set of foundation classes.

**Key words:** keyword pattern matching, C++ toolkits, C++ frameworks, generic algorithms, foundation classes, taxonomies.

## 1 Introduction and related work

In this paper, we outline the design, design rationale, and use of SPARE Parts, a C++ toolkit for string pattern matching. The algorithms implemented in the toolkit include:

- The Knuth-Morris-Pratt [KMP77] single keyword pattern matching algorithm.
- Several variants of the Boyer-Moore [BM77] single keyword pattern matching algorithms.
- Several variants of the Aho-Corasick [AC75] multiple keyword pattern matching algorithms.
- Several variants of the Commentz-Walter [Com79a, Com79b] multiple keyword pattern matching algorithm.

In addition to the original papers, all of these algorithms are extensively treated (with full correctness arguments) in a taxonomy presented in [Wats95, Chapter 4].

SPARE Parts is the second generation string pattern matching toolkit from the Ribbit Software Systems Inc. and the Eindhoven University of Technology. The toolkit is publicly available (though not in the public domain and it may not be redistributed) for noncommercial use. A re-implemented (and greatly extended) toolkit, known as SPARE Parts II, is available for commercial licensing from Ribbit Software Systems Inc. In addition to the functionality of SPARE Parts, it contains approximate pattern matchers, regular pattern matchers, multi-dimensional pattern matchers, and a highly tuned set of foundation classes.

The first generation toolkit (called the Eindhoven Pattern Kit, written in C and described in [Wats94, Appendix A]) is a procedural library based upon the original taxonomy of pattern matching algorithms [WZ92]. Experience with the Eindhoven Pattern Kit revealed a couple of deficiencies (leading to the design of SPARE Parts), detailed as follows. The rudimentary and explicit memory management facilities in C caused numerous errors in the code and made it difficult to perform pattern matching over more than one string simultaneously (in separate threads of the program) without completely duplicating the code. While the performance of the toolkit was excellent, some of the speed was due to sacrifices made in the understandability of the client interface.

There are other existing pattern matching toolkits, notably the toolkit of Hume and Sunday [HS91]. Their toolkit consists of a number of implementations of Boyer-Moore type algorithms — organized so as to form a taxonomy of the Boyer-Moore family of algorithms. Their toolkit was primarily designed to collect performance data on the algorithms. As a result, the algorithms are implemented (in C) for speed and they sacrifice some of the safety (in terms of error checking) that would normally be expected of a general toolkit. Furthermore, the toolkit does not include any of the non-Boyer-Moore pattern matching algorithms (other than a brute-force pattern matcher) and, most noticeably, does not contain multiple keyword pattern matchers.

SPARE Parts is a completely object-oriented implementation of the algorithms appearing in [Wats95, Chapter 4]. SPARE Parts is designed to address the shortcomings of both of the toolkits described above. The following are the primary features of the class library:

- The design of SPARE Parts follows the structure of the taxonomy in [Wats95, Chapter 4] very closely. As a result, the code is easier to understand and debug. In addition, SPARE Parts includes implementations of almost all of the algorithms described in [Wats95, Chapter 4].

- The use of C++ (instead of C) for the implementation has helped to avoid many of the memory management-related bugs that were present in the original toolkit.

- The client interface to the toolkit is particularly easy to understand and use. The flexibility introduced into the interface does not reduce the performance of the code in any significant way.

- The toolkit supports multi-threaded use of a single pattern matching object.

This paper is structured as follows:

- Section 2 considers issues in the design and implementation of class libraries.
- Section 3 gives an introduction to the client interface of the toolkit. It includes some examples of programs which use SPARE Parts.
- Section 4 presents some experiences with the toolkit and the conclusions of this chapter.
- Section 5 gives some information on how to obtain and compile the toolkit.

# 2 Designing and implementing class libraries

In this section, we briefly discuss some of the issues involved in designing, implementing, and presenting class libraries (or *toolkits*). The following description of a toolkit is taken from [GHJV95, p. 26]:

> A toolkit is a set of related and reusable classes designed to provide useful, general-purpose functionality. Toolkits don't impose a particular design on your application; they just provide functionality that can help your application do its job. They are the object-oriented equivalent of subroutine libraries.

We will use the terms *class library, library,* and *toolkit* interchangeably. We will also use the term *client* to refer to a program that makes use of classes in the toolkit, or the author of such a program. The important aspects and design goals of a toolkit are:

- Toolkits do not provide a user interface. (Toolkits that do provide user interfaces should be placed in the category of 'application program'.)
- The classes in the toolkit must have a coherent design, meaning that they are designed and coded in the same style. They have a clear relationship and a logical class hierarchy.
- The client interface to the library must be easily understood, permitting clients to make use of the library with a minimum of reading.
- The efficiency of using the classes in the toolkit must be comparable to hand-coded special-purpose routines — the toolkit must be applicable to production quality software.
- To provide an educational use for the toolkits, and to allow clients to easily modify classes and member functions, the method of implementation must be clear and understandable.

The toolkit is implemented in the C++ programming language, which was chosen because of its object-oriented features and its widespread availability. Efforts were made to refrain from using obscure features of C++ (such as RTTI or name spaces),

or language features not easily found in other object-oriented programming languages (such as multiple-inheritance).

Throughout this paper, we assume that the reader is familiar with the C++ language and object-oriented terminology (especially the C++ variety).

The general process of library design will not be described here, as there is a large body of literature discussing this issue. The following books are of particular relevance:

- [GHJV95, Souk94] discuss 'design patterns' (not related to our pattern matching problem) which are used heavily in library design.
- [CE95], [Stro91, Chapter 13] and [Stro94, Chapter 8] provide a general discussion of C++ library design.
- [MeyB94] is an excellent treatment of the design of a number of loosely coupled libraries in the Eiffel programming language. Many of the concepts and techniques discussed in the book are broadly applicable to C++ as well.
- [Plau95, Teal93] discuss the design and implementation of specific C++ libraries — the standard C++ library[1] and the IOStreams (input and output) class libraries respectively.

We define the following types of object-oriented classes:

**User:** A class intended for use by a client program.

**Client:** A class defined in the client program.

**Implementation:** A class defined in the toolkit for exclusive use by the toolkit. The class is used to support the implementation of the client classes.

**Foundation:** Those implementation classes which are simple enough to be reused in other (perhaps unrelated) class libraries.

**Interface:** An abstract (pure virtual) class which is declared to force a particular public interface upon its inheritance descendants.

**Base:** An inheritance ancestor of a particular class.

**Derived:** An inheritance descendant of a particular class.

## 2.1 Motivations for writing class libraries

There are a number of motivations for creating the class libraries:

- Until now, few general purpose toolkits of pattern matchers existed. The ones that do exist are not intended for general use in production quality software.

[1] Plauger's book considers the implementation of an early, and now defunct, draft of the standard library.

- The level of coherence normally required to implement a toolkit was not previously possible. The literature on pattern matching algorithms was scattered and in some places incomplete. With the construction of the taxonomy, [Wats95, Chapter 4], all of the algorithms are described in a coherent fashion, allowing us to base the class library structures on the taxonomy structure.

- The uniformity of implementation that was possible (given the taxonomy) had two important effects:

  - Clients need not examine the source code in order to make a decision on which class to use; the quality of the implementations of each of the pattern matchers is roughly the same.
  - Uniformity gives greater confidence in the accuracy of relative performance comparing different algorithms.

- The toolkit and the taxonomy can serve as examples of implementation techniques for class libraries; in particular methods for organizing template classes[2] and class hierarchies.

- Implementing the abstract algorithm can be painless and fun, given the taxonomy presentation of the algorithms and their correctness arguments.

## 2.2 Code sharing

One of the main aims of object-oriented programming is that it permits, and even encourages, code sharing (or code reuse). The code reuse in object-oriented programming corresponds neatly with the factoring of common parts of algorithms in the taxonomy.

Although code sharing can be achieved in a number of ways, in this section we discuss four techniques which could have been used in the design of the toolkits. The first discussion centres around the use of base classes (with virtual member functions) versus templates. The second discussion concerns the use of composition versus protected inheritance.

### 2.2.1 Base classes versus templates

A number of the pattern matching objects have common functionality and it seems wasteful to duplicate the code in each of the specific types of pattern matchers.

The obvious design involves creating a new base class and factoring the common code into the base class. Each of the pattern objects would then inherit from this base and provide specific virtual member functions to obtain the desired functionality. For example, the Commentz-Walter algorithms all share a common algorithm skeleton: they each have specific shift functions. We could create a CW base class with the functionality of the skeleton and provide a virtual 'shift distance' member function to obtain the Commentz-Walter variants.

[2] We use the term *template class*, as opposed to *class template* suggested by Carroll and Ellis in [CE95]. Our choice was made to correspond to the term *generic class* used in some other object-oriented programming languages.

The advantage of this approach is its elegance. It provides a relatively easy to understand class hierarchy, that reflects the structure of the taxonomy. Furthermore, a member function which takes (as a parameter) a pointer to a CW object need not know which particular variant (of a CW object) the pointer points to, only that the CW object satisfies the general CW functionality. This solution provides code reuse at both the source language and executable image levels. The disadvantage is that it would require a virtual function call for every shift. Indeed, if the same technique was used to factor the common code from the Aho-Corasick variants, it would require a virtual function call for every character of the input string.

The other approach is to create a template class CW, which takes a 'shifter class' as its template (type) parameter. We would then provide a number of such shifter classes, for use as template parameters — each giving rise to one of the Commentz-Walter variants. The primary advantage of this approach is that it is efficient: when used to implement the Aho-Corasick algorithms, each character in the input string will require a non-virtual function call (which may be inlined, unlike virtual function calls). The disadvantages are twofold: pointers to the variants of the CW algorithms are not interchangeable and code will be generated for each of the CW variants. The code reuse is at the source level and not at the executable image level.

It is expected, for example, that few clients of the toolkits will instantiate objects of different CW classes. A programmer writing an application using pattern matching is more likely to choose a particular type of pattern matcher, as opposed to creating objects of various different types. For this reason, the advantages of the template approach are deemed to outweigh its disadvantages, and we prefer to use it over base classes in the toolkits.

### 2.2.2 Composition versus protected inheritance

Composition (sometimes called the *has-a* relationship) and protected inheritance (sometimes called the *is-a* relationship) are two additional solutions to code sharing. We illustrate the differences between these two solutions using an example. When implementing a *Set* class, we may wish to make use of an already-existing *Array* class. There are two ways to do this: composition and protected inheritance.

With protected inheritance, class *Set* inherits from *Array* in a protected way. Class *Set* still gets the required functionality from *Array*, but the protected inheritance prevents the *is-a* relation between *Set* and *Array* (that is, we cannot treat a *Set* as an *Array*). The advantage of this approach is that it is elegant and it is usually the approach taken in languages such as Smalltalk and Objective-C [Budd91]. The disadvantage is that the syntax of C++ places the inheritance clause at the beginning of the class declaration of *Set*, making it plain to all clients of *Set* that it is implemented in terms of *Array*. Furthermore, protected inheritance (and indeed private inheritance) is one of the rarely-used corners of C++, and it is unlikely that the average programmer is familiar with it [MeyS92, Murr93].

In a composition approach, an object of class *Set* has (in its private section) an object of class *Array*. The *Set* member functions invoke the appropriate member functions of *Array* to provide the desired functionality. The advantage of this approach is that it places all implementation details in the private section of the class definition. The disadvantage is that it deviates from the accepted practice (in some other languages) of inheriting for implementation. It is, however, the standard approach

in C++. At first glance, it would appear that composition can lead to some inefficiency: in our example, an invocation of a *Set* member function would, in turn, call an *Array* member function. These extra function calls, usually called *pass-throughs*, are frequently eliminated through inlining.

There are no efficiency-based reasons to choose one approach over the other. For this reason, we arbitrarily choose composition because of the potential readability and understandability problems with protected inheritance.

### 2.3 Coding conventions and performance issues

At this time, coding in C++ presents at least two problems: the language is not yet stable (it is still being standardized) and, correspondingly, the "standard" class libraries are not yet stable.

In designing the libraries, every effort was made to use only those language features which are well-understood, implemented by most compilers and almost certain to remain in the final language. Likewise, the use of classes from the proposed standard library, or from the Standard Template Library [SL94], was greatly restricted. A number of relatively simple classes (such as those supporting strings, arrays, and sets) were defined from scratch, in order to be free of library changes made by the standardizing committee. A future version of the toolkits will make use of the standard libraries once the International Standards Organization has approved the C++ standard.

In the object-oriented design process, it is possible to go overboard in defining classes for even the smallest of objects — such as alphabet symbols. In the interests of efficiency, we draw the line at this level and make use of integers for such basic objects.

Almost all of the classes in the toolkits have a corresponding class invariant member function, which returns *TRUE* if the class is structurally correct and *FALSE* otherwise[3]. Structural invariants have proven to be particularly useful in debugging and in understanding the code (the structural invariant is frequently a good first place to look when trying to understand the code of a class). For this reason, they have been left in the released code (they can be disabled as described in the next section).

We use a slightly non-traditional way of splitting the source code into files. The public portion of a class declaration is given in a `.hpp` file, while the private parts are included from a `.ppp` file. There is a corresponding `.cpp` file containing all of the out-of-line member function definitions. A `.ipp` file contains member functions which can be inlined for performance reasons. By default the member functions in the `.ipp` file are out-of-line. The inlining can be enabled by defining the macro `INLINING`. To implement such conditional inlining, the `.ipp` file is conditionally included into the `.hpp` or the `.cpp` file. The inlining should be disabled during debugging or for smaller executable images.

## 3 Using the toolkit

In this section, we describe the client interface of the toolkit and present some examples of programs using the toolkit.

[3]This will be changed to use the new **bool** datatype once most compiler support it.

The client interface defines two types of abstract pattern matchers: one for single keyword pattern matching and one for multiple keyword pattern matching. (A future version of SPARE Parts can be expected to include classes for regular expression pattern matching — for example, an implementation of the algorithm described in [Wats95, Chapter 5].) All of the single keyword pattern matching classes have constructors which take a keyword. Likewise, the multiple keyword pattern matchers have constructors which take a set of keywords. Both types of pattern matchers make use of *call-backs* (to be explained shortly) to register matched patterns. In order to match patterns using the call-back mechanism, the client takes the following steps (using single keyword pattern matching as an example):

1. A pattern matching object is constructed (using the pattern as the argument to the constructor).

2. The client calls the pattern matching member function *PMSingle::match*, passing the input string and a pointer $f$ to a client defined function which takes an **int** and returns an **int**[4]. (This function is called the *call-back function*.)

3. As each match is discovered by the member function, the call-back function is called; the argument to the call is the index (into the input string) of the symbol immediately to the right of the match. (If there is no symbol immediately to the right, the length of the input string is used.)

4. If the client wishes to continue pattern matching, the call-back function returns the constant *TRUE*, otherwise *FALSE*.

5. When no more matches are found, or the call-back function returns *FALSE*, the member function *PMSingle::match* returns the index of the symbol immediately to the right of the last symbol processed.

We now consider an example of single keyword pattern matching.

The following program searches an input string for the keyword `hisher`, printing the locations of all matches along with the set of matched keywords:

```
#include "com-misc.hpp"
#include "pm-kmp.hpp"
#include <iostream.h>

static int report( int index ) {
    cout << index << '\n';
    return( TRUE );
}

int main( void ) {                                          10
    auto PMKMP Machine( "hisher" );
    Machine.match( "hishershey", &report );
    return( 0 );
}
```

[4]The integer return value is a Boolean value; recall that *TRUE* and *FALSE* have type **int** in C and C++. The new **bool** keyword is not yet supported by all compilers.

The header file `com-misc.hpp` provides a definition of constants *TRUE* and *FALSE*. Header file `pm-kmp.hpp` defines the Knuth-Morris-Pratt pattern matching class, while header file `iostream.h` defines the input and output streams, including the standard output *cout*. Function *report* is our call-back function, simply printing the index of the match (to the standard output) and returning *TRUE* to continue matching. The *main* function (the program mainline) creates a local KMP machine, with keyword `hisher`. The machine is then used to find all matches in string `hishershey`. (Recall that, in C and C++, a pointer to the beginning of the string is passed to member *match*, as opposed to the entire string.)

In addition to the KMP algorithm defined in `pm-kmp.hpp`, other single keyword pattern matchers are defined in header file `bms.hpp`, which contains suggestions for instantiating some of the Boyer-Moore variants. Additionally, a brute-force single keyword pattern matcher is defined in `pm-bfsin.hpp`.

Multiple keyword pattern matching is performed in a similar manner, as the following example shows. The following program searches an input string for the keywords `his`, `her`, and `she`, printing the locations of all matches:

```
#include "com-misc.hpp"
#include "string.hpp"
#include "set.hpp"
#include "acs.hpp"
#include <iostream.h>

static int report( int index, const Set<String>& M ) {
    cout << index << M << '\n';
    return( TRUE );
}

int main( void ) {
    auto Set<String> P( "his" );
    P.add( "her" ); P.add( "she" );
    auto PMACOpt Machine( P );
    Machine.match( "hishershey", &report );
    return( 0 );
}
```

Header file `string.hpp` defines a string class, while `set.hpp` defines a template class for sets of objects. Header file `acs.hpp` defines the Aho-Corasick pattern matching classes. Function *report* is our call-back function, simply printing the index of the match (to the standard output) and the set of keywords matching, and returning *TRUE* to continue matching. Note that the call-back function has a different signature for multiple keyword pattern matching: it takes the index of the symbol to the right of the match and the set of keywords matching with *index* as their right end-point.

The *main* function (the program mainline) creates a local AC machine from the keyword set. The machine is then used to find all matches in string `hishershey`. In the following two sections, we consider ways to use SPARE Parts more efficiently in certain application domains.

## 3.1 Multi-threaded pattern matching

One important design feature (as a result of the call-back client interface) of SPARE Parts is that it supports multi-threading. This can lead to high performance in applications hosted on multi-threading operating systems. For example, consider an implementation of a keyword `grep` application in which 1000 files are to be searched for occurrences of a given keyword. The following are three potential solutions:

- In a sequential solution, a single pattern matching object is constructed and each of the 1000 files are scanned (in turn) for matches.
- In a naïve multi-threaded solution, 1000 threads are created (each corresponding to one of the input files). Each of the threads construct a pattern matching object, which is then used to search the file.
- An efficient solution is to create a single matching object, with 1000 threads sharing the single object. Each of the threads proceeds to search its file, using its own invocation of member function *PMSingle::match*.

The last (most efficient) solution would not have been possible without the call-back client interface.

## 3.2 Alternative alphabets

The default structure in SPARE Parts is to make use of the entire ASCII character set as the alphabet. This can be particularly inefficient and wasteful in cases where only a subset of these letters are used. For example, in genetic sequence searching, only the letters $a$, $c$, $g$, and $t$ are used. SPARE Parts facilitates the use of smaller alphabets through the use of *normalization*. Header file `alphabet.hpp` defines a constant *ALPHABETSIZE* (which, by default is *CHAR_MAX*). The alphabet which SPARE Parts uses is the range $[0, ALPHABETSIZE)$. An alternative alphabet can be used by redefining *ALPHABETSIZE* and mapping the alternative alphabet in the required range. The mapping is performed by functions *alphabetNormalize* and *alphabetDenormalize*, both declared in `alphabet.hpp` (by default, these functions are the identity functions). The only requirement is that the functions map 0 to 0 (this is used to identify the end of strings). In the genetic sequence example, we would make use of the following version of header `alphabet.hpp`:

```
#include <assert.h>
#define ALPHABETSIZE 5

inline char alphabetNormalize( const char a ) {
    switch( a ) {
    case 0:     return( 0 );
    case 'a':   return( 1 );
    case 'c':   return( 2 );
    case 'g':   return( 3 );
    case 't':   return( 4 );                                    10
    default:    assert( !"Non-genetic character" );
    }
```

```
}

inline char alphabetDenormalize( const char a ) {
    switch( a ) {
    case 0:     return( 0 );
    case 1:     return( 'a' );
    case 2:     return( 'c' );
    case 3:     return( 'g' );
    case 4:     return( 't' );
    default:    assert( !"Non-genetic image" );
    }
}
```

# 4 Experiences and conclusions

Designing and coding SPARE Parts lead to a number of interesting experiences in class library design. In particular:

- SPARE Parts comprises 5787 lines of code in 59 `.hpp`, 32 `.cpp`, 43 `.ppp`, and 49 `.ipp` files.
- Compiling the files, with the WATCOM C++32 Version 9.5b compiler, shows that the size of the object code varies very little for the various types of pattern matchers.
- The taxonomy presented in [Wats95, Chapter 4] was critical to correctly implementing the many complex precomputation algorithms.
- Designing and structuring generic software (reusable software such as class libraries) is much more difficult than designing software for a single application. The general structure of the taxonomy proved to be helpful in guiding the structure of SPARE Parts.
- In [Wats95, Chapter 4], we consider the relative performance of the algorithms implemented in SPARE Parts. It is also helpful to consider how the implementations in SPARE Parts fare against commercially available tools such as the `fgrep` program. Four `fgrep`-type programs were implemented (using SPARE Parts), corresponding to the Knuth-Morris-Pratt, Aho-Corasick, Boyer-Moore and Commentz-Walter algorithms. The four tools were benchmarked informally against the `fgrep` implementation which is sold as part of the MKS toolkit for MS-DOS. The resulting times (to process a 984149 byte text file, searching for a single keyword) are:

| `fgrep` *variant* | MKS | KMP | BM | AC | CW |
|---|---|---|---|---|---|
| *Time (sec)* | 3.9 | 5.1 | 4.2 | 4.7 | 4.0 |

  These results indicate that using a general toolkit such as SPARE Parts will result in performance which is similar to carefully tuned C code (such as MKS `fgrep`). Much more extensive test results are reported in [Wats95, Wats96].

Detailed records were kept on the time required for designing, typing, compiling (and fixing syntax errors) and debugging the toolkit. The time required to implement the toolkit is broken down as follows (an explanation of each of the tasks is given below):

| *Task* | Design | Typing | Compile/Syntax | Debug | Total |
|---|---|---|---|---|---|
| *Time (hrs:min)* | 6:00 | 13:40 | 10:05 | 5:15 | 35:00 |

Most of these times are quite short compared to what a software engineer could expect to spend on a project of comparable size. The following paragraphs explain exactly what each of the tasks entailed:

- The *design* phase involved the creation of the inheritance hierarchy and the declaration (on paper) of all of the classes in the toolkit. (A C++ declaration provides names and signatures of functions, types and variables, whereas a definition provides the implementation of these items.) The design phase proceeded exceptionally smoothly thanks to a number of things:
  - The inheritance hierarchy followed directly from the structure of the taxonomy.
  - The decisions on the use of templates (instead of virtual functions) and call-backs were made on the basis of experience gained with FIRE Engine, Ribbit's finite automata and transducer toolkit. These decisions were also somewhat forced by the efficiency requirements for the toolkit as well as the need for multi-threading.
  - Representation issues, such as the selection of data structures, were resolved using experience gained with the earlier Eindhoven Pattern Kit.
- Once the foundation classes were declared and defined, typing the code amounted to a simple translation of guarded commands (appearing in [Wats95]) to C++.
- The times required for compilation and syntax checking were minimized by using a very fast integrated environment (BORLAND C++) for initial development. Only the final few compilations were done using the (slower, but more thoroughly optimizing) WATCOM C++ compiler. The advantages of using a fast development environment on a single user personal computer should not be underestimated.
- Since the C++ code in the toolkit was implemented directly from the abstract algorithms (for which correctness arguments are given), the only (detected) bugs were those involving typing errors (such as the use of the wrong variable, etc.). Correspondingly, little time needed to be spent on debugging the toolkit.

New research in pattern matching requires that tools such as SPARE Parts evolve. The following are some of the upcoming changes:

- The toolkit will use a version of the C++ Standard Template Library.
- In light of the success and widespread applicability of the commercialized version, SPARE Parts II, SPARE Parts will be template parameterized to support input strings and patterns over arbitrary alphabets (as opposed to the rudimentary alphabet support now provided).

- Efforts will begin to integrate approximate pattern matching algorithms into the toolkit.

# 5 Obtaining and compiling the toolkit

SPARE Parts is available via `www.RibbitSoft.com/research/watson/`. The toolkit and some associated documentation are combined into a `tar` file.

SPARE Parts has been successfully compiled with BORLAND C++ Versions 3.1 and 4.0, and WATCOM C++32 Version 9.5b on MS-DOS and MICROSOFT WINDOWS 95 platforms. Since the WATCOM compiler is also a cross-compiler, there is every reason to believe that the code will compile for WINDOWS NT or for IBM OS/2. The implementation of the toolkit makes use of only the most basic features of C++ and it should be compilable using any of the template-supporting UNIX based C++ compilers.

A version of SPARE Parts will remain freely available (though not in the public domain). Contributions to the toolkit, in the form of new algorithms or alternative implementations, are welcome.

# References

[AC75] AHO, A.V. and M.J. CORASICK. Efficient string matching: an aid to bibliographic search, *Comm. ACM,* **18**(6) (1975) 333–340.

[BM77] BOYER, R.S. and J.S. MOORE. A fast string searching algorithm, *Comm. ACM,* **20**(10) (1977) 62–72.

[Budd91] BUDD, T.A. *An introduction to object-oriented programming.* (Addison-Wesley, Reading, MA, 1991).

[CE95] CARROLL, M.D. and M.A. ELLIS. *Designing and coding reusable C++.* (Addison-Wesley, Reading, MA, 1995).

[Com79a] COMMENTZ-WALTER, B. A string matching algorithm fast on the average, in: H.A. Maurer, ed., *Proc. 6th Internat. Coll. on Automata, Languages and Programming* (Springer-Verlag, Berlin, 1979) 118–132.

[Com79b] COMMENTZ-WALTER, B. A string matching algorithm fast on the average, Technical Report TR 79.09.007, IBM Germany, Heidelberg Scientific Center, 1979.

[GHJV95] GAMMA, E., R. HELM, R. JOHNSON, and J. VLISSIDES. *Design Patterns: Elements of Reusable Object-Oriented Software.* (Addison-Wesley, Reading, MA, 1995).

[HS91] HUME, S.C. and D. SUNDAY. Fast string searching, *Software—Practice and Experience* **21**(11) (1991) 1221–1248.

[KMP77] KNUTH, D.E., J.H. MORRIS and V.R. PRATT. Fast pattern matching in strings, *SIAM J. Comput.* **6**(2) (1977) 323–350.

[MeyB94] MEYER, B. *Reusable Software: The Base Object-Oriented Component Libraries.* (Prentice Hall, Englewood Cliffs, NJ, 1994).

[MeyS92] MEYERS, S. *Effective C++: 50 specific ways to improve your programs.* (Addison-Wesley, Reading, MA, 1992).

[Murr93] MURRAY, R.B. *C++ strategies and tactics.* (Addison-Wesley, Reading, MA, 1993).

[Plau95] PLAUGER, P.J. *The Draft Standard C++ Library.* (Prentice Hall, New Jersey, 1995).

[SL94] STEPANOV, A. and M. LEE. Standard Template Library, Computer Science Report, Hewlett-Packard Laboratories, 1994.

[Souk94] SOUKUP, J. *Taming C++: Pattern Classes and Persistence for Large Projects.* (Addison-Wesley, Reading, MA, 1994).

[Stro91] STROUSTRUP, B. *The C++ programming language.* (Addison-Wesley, Reading, MA, 2nd edition, 1991).

[Stro94] STROUSTRUP, B. *The Design and Evolution of C++.* (Addison-Wesley, Reading, MA, 1994).

[Teal93] TEALE, S. *C++ IOStreams Handbook.* (Addison-Wesley, Reading, MA, 1993).

[Wats94] WATSON, B.W. The performance of single-keyword and multiple-keyword pattern matching algorithms, Computing Science Report 94/19, Eindhoven University of Technology, The Netherlands, 1994.

[Wats95] WATSON, B.W. *Taxonomies and Toolkits of Regular Language Algorithms.* (Ph.D dissertation, Eindhoven University of Technology, The Netherlands, 1995).
For availability, see `www.RibbitSoft.com/research/watson/`.

[Wats96] WATSON, B.W. The performance of single and multiple keyword pattern matching algorithms, *Workshop on String Processing* (Recife, Brazil, 1996). Available via `www.RibbitSoft.com/research/watson/`.

[WZ92] WATSON, B.W. and G. ZWAAN. A taxonomy of keyword pattern matching algorithms, Computing Science Report 92/27, Eindhoven University of Technology, The Netherlands, 1992.

[WZ95] WATSON, B.W. and G. ZWAAN. A taxonomy of sublinear keyword pattern matching algorithms, Computing Science Report 95/13, Eindhoven University of Technology, The Netherlands, 1995.