Creating global software: Text handling and localization in Taligent's CommonPoint application system

by M. E. Davis
J. D. Grimes
D. J. Knoles

Developing software that can be used across global enterprises is one of the many challenges of today's information technology systems. Taligent's CommonPoint™ application system eases this problem by providing a foundation for fully global software, based on object-oriented frameworks and the Unicode™ character encoding standard. This paper describes Taligent's Unicode implementation and the CommonPoint text and international frameworks. It discusses how the CommonPoint system can be used to build international software and some of the advantages of object-oriented technology.

Many organizations today are faced with the challenge of implementing software that operates seamlessly across national borders. The goal is to create *global* applications, that is, applications that have a single binary form that can be used everywhere. These applications are then localized for use in a particular geographic region, usually a country, that shares a language along with other local characteristics such as a time zone, currency units, and common number and date formats. Unfortunately, it is not easy to create global applications, and often a different binary version is needed to support each country or region. In addition to the expense and inconvenience of creating and maintaining multiple versions of an application, documents created using a particular localized version of a program cannot be displayed correctly by other versions.

Taligent's CommonPoint** application system facilitates the creation of global software. The CommonPoint system comprises a set of integrated object-oriented frameworks, implemented in C++, that enable the development of modular object-oriented applications and documents. The CommonPoint system runs as a layer on existing operating systems, including the Advanced Interactive Executive* (AIX*) and Operating System/2* (OS/2*) environments. The CommonPoint system allows applications to be created with these global qualities:

- Users can enter and manipulate textual and numerical data in their native language, and can create and display multilingual text.
- The application can be completely localized without accessing its source code. The interface can be presented in any user's native language, and the same binary version can have multiple localized presentations.

- There is high potential for customization by both the user and the developer, provided by object-oriented frameworks and modular, data-driven objects. Users have more control over which resources they use, and developers can take advantage of the functionality already provided by the frameworks and focus on adding more specialized features and more localized resources.

This paper describes the support for globally distributable software provided by the CommonPoint application system. The CommonPoint system enables international software development by providing:

- An implementation of the Unicode** character encoding standard that provides a common mechanism for storing character data regardless of language. The Unicode character set provides a full set of symbols and other characters, enables the creation of text in multiple languages and scripts, and provides data integrity.
- Text handling mechanisms that facilitate the storage and manipulation of multilingual-styled text
- Character input features that allow users to enter multilingual text using today's standard input devices
- Localization services that allow localizable resources to be easily created, stored, and customized for use in a specific language or geographic region
- Powerful object-oriented frameworks and data-driven localizable objects that enable a high degree of customization and extensibility

The paper describes these mechanisms and discusses the impact of object-oriented technology on the implementation of international software.[1]

Applying the Unicode standard

Use of the Unicode standard as the sole character encoding mechanism is the foundation for the CommonPoint international feature set. Because the Unicode standard is so fundamental to the design of the CommonPoint system's text and international frameworks, it is worth summarizing the standard and some of its features here.

Developed by the Unicode Consortium,[2] the Unicode standard is a fixed-width, 16-bit character encoding system that contains codes for every character needed by the major writing systems currently in use in the modern world, along with codes for a full range of punctuation, symbols, and control characters. The Unicode standard provides, in all, codes for over 34 000 characters from the world's alphabets, ideographs, and symbol sets. The standard incorporates characters from many existing standards—for example, the first 256 characters correspond to the International Organization for Standardization (ISO) Latin-1 character set (which attempts to provide character coverage for the major Western European languages)—and is compatible with the international standard ISO/IEC (International Electrotechnical Commission) 10646.[3,4]

Along with a script or character name, the Unicode standard associates semantic information with each character that can be used to simplify text processing features. Each character can have an associated set of descriptive type properties identifying, for example:

- Punctuation marks (for example, [?] and ['])
- Diacritical marks (for example, [´] and [¨])
- Uppercased, lowercased, and uncased letters (for example, [A] and [a], respectively—uncased letters appear in languages such as Hebrew and Arabic that do not distinguish between uppercase and lowercase)
- Characters used to represent digits (for example, [0] and [5])
- Control characters (for example, a carriage return or end-of-text character)

Exclusive use of the Unicode standard for all character data in the CommonPoint system automatically eases several of the problems inherent in creating international applications on many of today's current systems by providing a simple and consistent interface, for manipulating character data, that does not vary based on the language being manipulated. Many programs on existing systems are currently based on much more limited character sets—for example, the 7-bit ASCII (American National Standard Code for Information Interchange) character standard. Several methods have been developed to help overcome the limitations of these relatively small character sets. The ISO 8859 standard, for example, provides a series of 8-bit extended character sets that use the standard ASCII character set for the first 7 bits and the eighth bit to define another 128 characters, thus extending ASCII to support a variety of additional languages.

This provides a partial solution for programs that need to support only a single language, or a set of languages whose character requirements are very similar. However, implementing the ability to produce many combinations of character sets requires an additional enhancement: the use of switching codes, or escape sequences, that are embedded in the text and indicate the character set of the following characters. This allows the creation of multilingual text, but text features become much more difficult to implement because the program must implement mechanisms that determine the character set to which any given character or range of text belongs.

Providing applications that support Japanese, or other eastern languages that cannot be supported by an 8-bit character set, is even more complicated. In these markets, double-byte and triple-byte character sets are used to define the large number of characters—often tens of thousands—that are required. Typically these languages are encoded with a combination of single- and double-byte codes, such as shift-JIS (JIS is the Japanese Industrial Standard). Programs quickly become much more complex because processing double-byte character data requires very different code than processing single-byte character data.

These are some of the problems the Unicode standard eliminates. The Unicode standard provides a built-in solution because it contains codes for virtually all of the characters needed to support all major writing systems, in any combination. Every character is encapsulated as a 16-bit unsigned integer, so there is no need to write different code to deal with both single-byte and double-byte data. Perhaps more importantly, the integrity of text data is much higher because character data are always interpreted using a single encoding standard. This means that:

- Programming errors are minimized—text-processing code does not need to examine the current font, maintain escape-sequence state, or use any other heuristic to determine the semantics, or the byte boundaries, of a character. The semantic meaning is inherent in the character code.
- Data loss is prevented—internal conversions between different code pages are not required. This prevents problems that occur today when characters cannot be represented in current code pages, and means that loss of font information or escape-sequence tags does not destroy the meaning of the text data.

Text handling

All text handling in the CommonPoint system is based on the Unicode standard described above. The CommonPoint system defines a system data type called UniChar, analogous to the C language data type char, which encapsulates individual Unicode character codes. (Note: w_char is not used because, unfortunately, the C++ standard does not guarantee that its implementation will be large enough to hold 16 bits.)

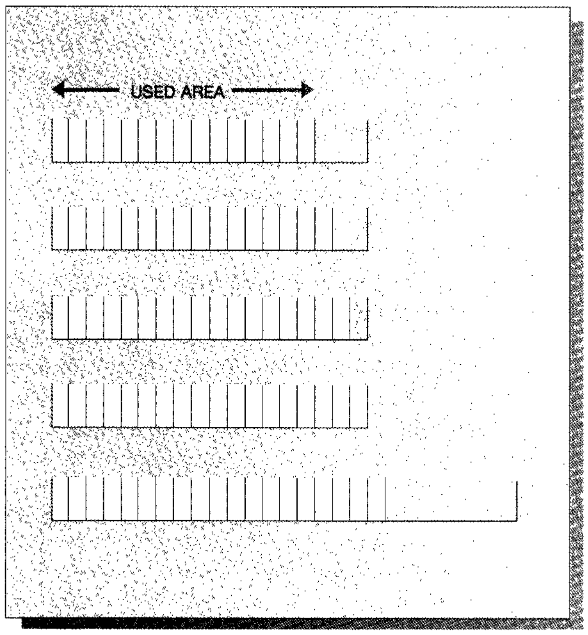
All CommonPoint character data are encapsulated using the UniChar data type. Text data from a non-Unicode system can be converted to Unicode data at the point of entry into the CommonPoint system and then used anywhere. The CommonPoint text system provides special objects called *transcoders* to handle this conversion. Each transcoder supports a specific non-Unicode character encoding standard, such as ASCII or JIS, and can handle mapping of character data both into and out of that standard. Transcoders can be built to support any character encoding.

Higher-level text handling mechanisms provided by the CommonPoint system are also based on Unicode character data. The primary text-handling mechanisms are the basic text class TText, used to encapsulate Unicode character strings, and the text editing framework, which allows users to enter and edit multilingual-styled text.

Creating text objects. TText, an abstract class, defines the interface for text objects that are usable anywhere in the CommonPoint system. Instances of TText subclasses can contain any combination of Unicode characters from any of the available scripts or symbol sets. This means that this class can be used to store and manipulate multilingual text strings without having to add any additional multilingual text-handling support. TText also allows styling information to be associated with any or all characters encapsulated by a text object. The exact implementation for character and styling information is defined by TText subclasses.

Because TText provides a single set of protocols for text strings throughout the system, mixed-style text can appear anywhere text appears, including labels, buttons, menu text, dialog fields, and spreadsheet cells. TText bundles character and style information in the same object, which means that, unlike many current systems, text can be moved between CommonPoint applications and subsystems without los-

Figure 1 Storage allocation for small strings



ing any of the styling information. The system includes a concrete subclass, TStandardText, that provides the primary mechanism for representing styled text.

Storing character data. TStandardText can be used for strings of almost any size, from only a few characters up to 2 billion characters. TStandardText objects dynamically change their storage allocation strategy to provide efficient storage at different sizes. At small sizes, each instance uses a single contiguous block of memory. For efficiency, the block is resized only when necessary. Figure 1 shows the use of memory with successive character insertions. The strategy automatically changes to use discontiguous storage for longer strings, as shown in Figure 2. This implementation provides for much better performance when inserting and deleting characters, avoiding rescoping or reallocating data unnecessarily.

Storing style data. TStandardText also implements a storage mechanism for styling information. Individual styles or groups of styles (called style sets) are created, and then applied to specific characters or ranges of characters in the text object. Styles that apply to paragraphs rather than individual characters, such as indentation or justification, are applied to entire paragraphs, as delimited by paragraph separator characters, so that specific character ranges need not be calculated. TStandardText manages all the necessary storage for these styles and ensures that styling information is stored efficiently. For example, any contiguous range of adjacent characters or paragraphs with identical styling information always shares a single style set. TStandardText objects allocate space for styling information only when the style is actually applied to the text. This means that unstyled text objects are very lightweight, yet the same class can be used to store fully styled text data. TStandardText also ensures that no contradictory styling information is applied to the same text. If a user applies a red color style to a character string, for example, any existing color style information is replaced.

Figure 3 shows the class relationships for text and style classes. The "Δ" symbol indicates an abstract class or method or an inheritance relationship (for example, TStandardText is a subclass of the abstract class TText). The "n" indicates a one-to-many relationship (for example, a TStyleSet object can contain more than one TStyle object).

Note that styles in the CommonPoint system derive from the general protocol for data attributes. In CommonPoint, text styles never affect the underlying meaning of the character data (unlike in the Macintosh system, where multiple character sets are supported by a single character encoding by applying different fonts for each character set). Text styles simply associate additional characteristics with the text, typically describing how it should be displayed. While encapsulated in a single text object, style data are actually parallel to character data and can be ignored by text operations such as searching for and sorting character strings.

Subclasses of the abstract class TStyle support a wide variety of text styles, including common styles such as font, point size, weight (for example, bold or light), posture (for example, italic or backslant), typographic style, letter-spacing, line-spacing, justification and paragraph indentation, superscripts and subscripts, color, underlining, and so on. The styling mechanism can also be used to apply arbitrary graphic transformations to text strings, such as skewing, stretching, and rotating, as shown in Figure 4.

In addition to the kinds of styles used to determine how to render (or display) text, the styling mechanism supports nonrendering styles that can be used

Figure 2 Storage allocation for longer strings

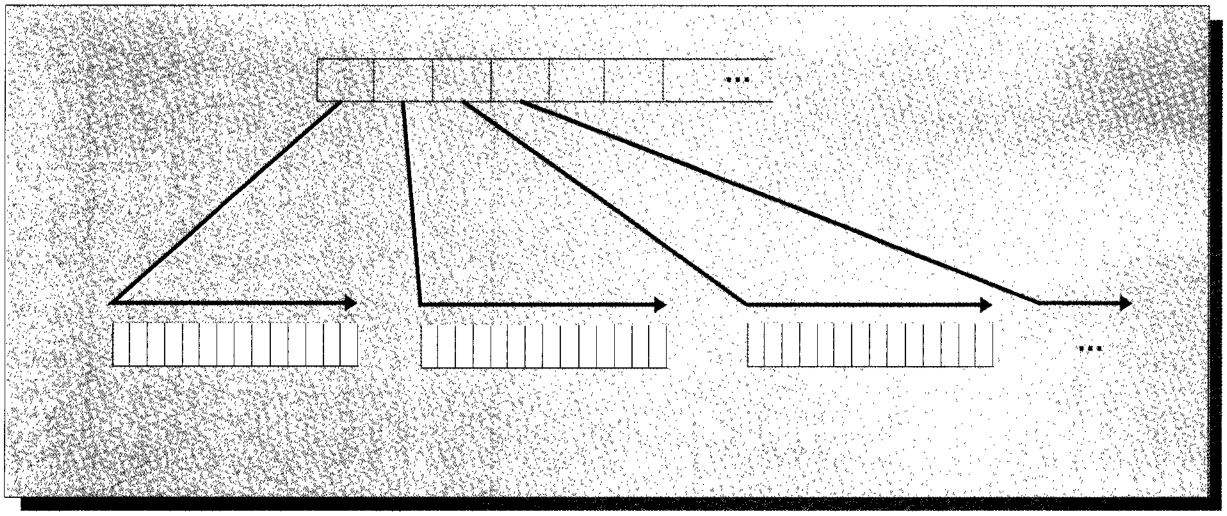
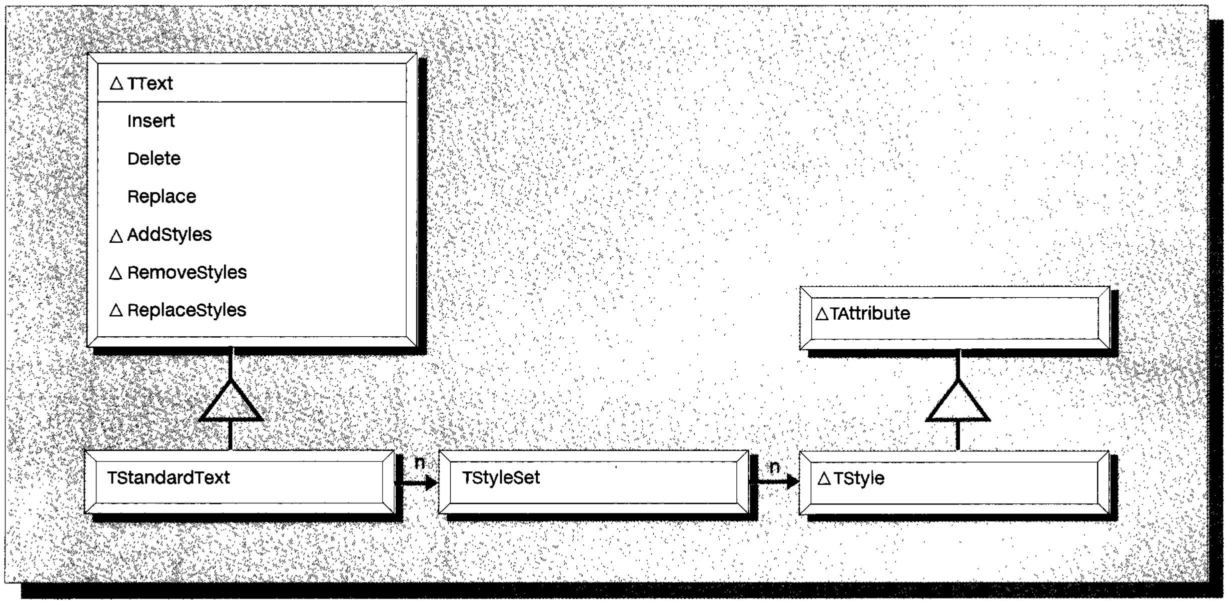
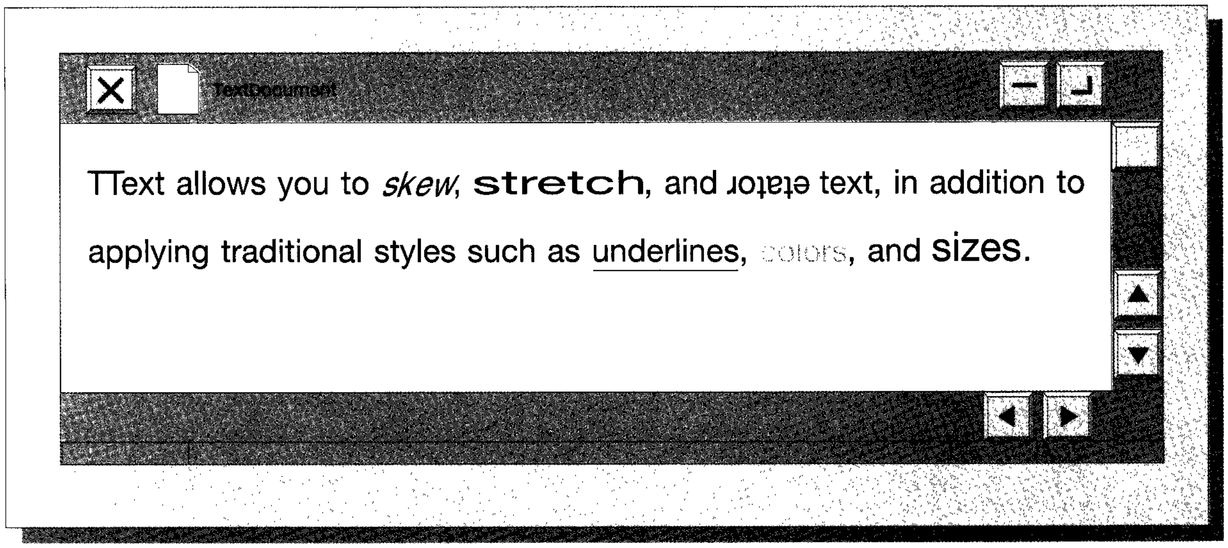


Figure 3 Class relationships for text and style classes



to encapsulate other information about characters or paragraphs. For example, the style TLanguage-Style identifies the natural language of a character string, providing information in addition to the character information inherent in the Unicode representation. This might be useful, for example, in searching a multilingual text document for text of a particular language—text of other languages could be eliminated from the search range. This information also enables the system to effectively select an appropriate font for displaying character strings if a font is not explicitly specified. Because any class

Figure 4 Text string transformations



TText allows you to *skew*, **stretch**, and ɹoʇɐʇǝ text, in addition to applying traditional styles such as underlines, colors, and sizes.

descending from TStyle can be applied to a range of characters in a TText object, TStyle subclasses can be designed to add arbitrary information to any piece of text.

An important aspect of the CommonPoint text and international features is the separation of language-sensitive facilities from basic string storage and manipulation. For example, a TTextComparator object, an object that encompasses the rules necessary for language-sensitive string comparison, is a separate object. Different comparators can be used with different strings at will—no global state is involved. Additionally, as a separate object it can be:

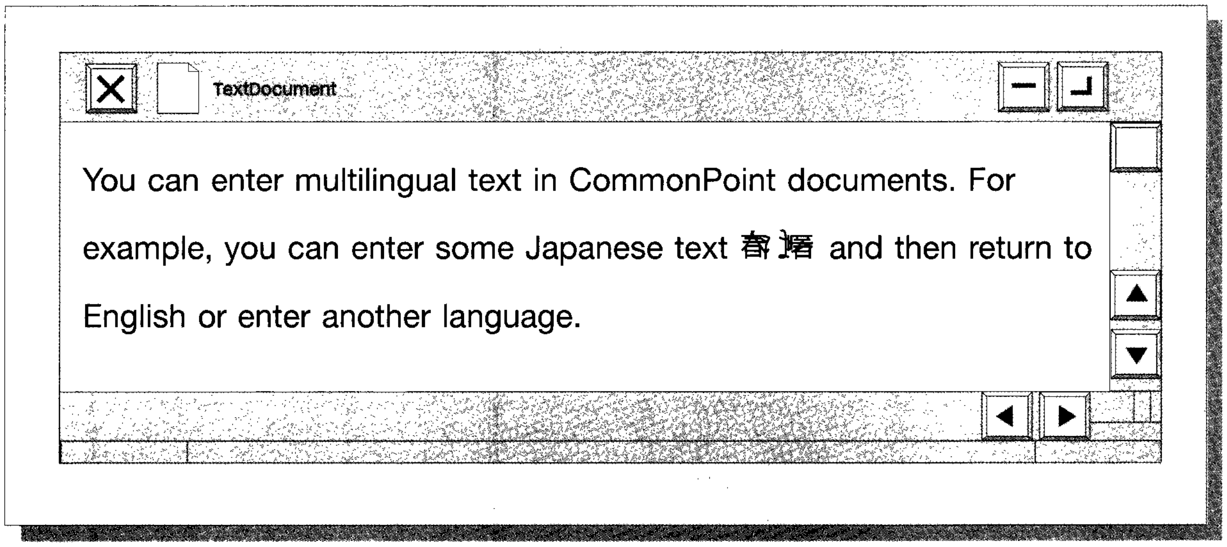
- Sent to a server for remote sorting and searching operations
- Modified (that is, rules can be added or deleted) and then applied to text

User editing features. Support for display and editing of styled text is provided by the text editing framework. Like other CommonPoint frameworks, the text editing framework is extensible and provides a foundation for continued evolution of text editing functionality.[5] The text editing framework is intended to provide a complete text editing facility that can be used by applications in which the primary focus is not text editing, for example, for e-mail or for text fields in a charting program. Word processing and desktop publishing programs should not subclass this framework, but build directly on the underlying system support. The text editing framework is integrated with the rest of the system, and is based on other application frameworks that provide features such as automatic "undo" and "redo" of user actions, collaboration, and the ability to embed other data types (for example, movies or graphics) within text.[6]

The editable text data type is based on TStandardText and provides the additional features needed to display and edit the character and style data. The text editing framework provides a fully functional interface for entering text, including preassembled menus that allow users to apply any of the character or paragraph styles described above to any selected range of characters or paragraphs. The framework also includes a set of cursor tools that provide an alternative mechanism for applying styles—the user activates the tool and uses the mouse or cursor to apply styles through direct manipulation.

The text editing framework supports entry and display of text in arbitrary combinations of languages and scripts. Input of multilingual text using the keyboard is enabled by a set of typing configurations, described in more detail in the next section. The framework provides menus that allow users to switch between any available typing configurations—including English, French, Greek, and Japanese—to en-

Figure 5 Multilingual text in a document



You can enter multilingual text in CommonPoint documents. For example, you can enter some Japanese text 書籍 and then return to English or enter another language.

ter multilingual text into a single document. The framework provides intuitive behavior for switching between configurations. For example, the framework automatically changes the font, if necessary, when the user activates a new typing configuration. When the user switches from an English keyboard configuration to a Russian keyboard configuration, for example, the framework automatically begins applying a Cyrillic font to the input text. The framework also reactivates the correct typing configuration when a user places the cursor arbitrarily into a text document. For example, if a user placed the cursor within the Kanji ideographs in the document shown in Figure 5, the Japanese typing configuration would automatically be activated.

The text editing framework also implements a font substitution mechanism to ensure that multilingual text data are always displayed meaningfully. This mechanism is automatically activated when the user enters a character that is not displayable by the current font. Instead of displaying a "box" or other meaningless glyph, the mechanism searches the available fonts for the correct glyph to display that character, using heuristics that take context into account. If it cannot find an appropriate glyph, it displays a glyph from a special font provided by the CommonPoint system that identifies the script or general category of that character. These glyphs are enclosed by a rounded rectangle so that you can identify immediately that an additional font must be installed to display the data. For example, Table 1 shows the glyphs used to represent missing glyphs from several scripts and categories. This mechanism ensures meaningful display of multilingual text and facilitates the exchange of text documents internationally.

Entering multilingual text

Multilingual character input in the CommonPoint system is enabled by typing configurations, each of which supports typing in a specific script or language. Users generally select a typing configuration as the default for their system. However, they can activate different configurations arbitrarily to enter multilingual text, as described in the previous section.

A typing configuration consists of several components:

- A virtual keyboard mapping
- One or more text modifiers (tools that map characters from one sequence into another based on context)
- Optionally, an input method for entering ideographic characters such as Kanji

As the user types, the CommonPoint input system converts keystrokes to key codes and passes them to the active typing configuration. The typing config-

Table 1 Glyphs inserted to indicate missing fonts

Glyph	Script or General Category
$\boxed{\text{L}}$	Roman
$\boxed{\text{ש}}$	Hebrew
$\boxed{\$}$	Currency Symbol
$\boxed{\text{✈}}$	Dingbat Symbol

uration creates the correct character string and inserts it into the document, as shown in Figure 6. Each of these elements takes advantage of the Unicode foundation and of the data reuse inherent in object-oriented technology to provide efficient, robust typing functionality.

Virtual keyboards. The virtual keyboard is the only required element in the typing configuration. The virtual keyboard provides a mapping between the codes issued by the physical keyboard and virtual key codes associated with specific characters. Generally a virtual keyboard mapping corresponds to a specific language, so there can be multiple keyboard mappings that support a particular script. For example, on an English keyboard, key 2 (the "Q" key on a QWERTY keyboard) maps to the character "q," while on a French keyboard it maps to the character "a." A virtual keyboard can produce any text object from a keystroke, including multiple characters and styled text. For example, a keyboard could be configured to enter the user's name or another common string with a single keystroke.

Text modifiers. After the virtual keyboard determines the correct character codes, they are processed by any text modifiers in the configuration. The typing configuration allows text modifiers to be chained together, so that the modified text produced by one text modifier is the input text to the next text modifier. Currently the CommonPoint system has defined two kinds of text modifiers: lexical tools and transliterators. Lexical tools operate on words, such as spelling or grammar checkers. Lexical tools can be added directly to the typing configuration so that these operations occur as the user types.

The CommonPoint system does not currently provide any lexical tools directly. It does, however, include a number of transliterators. Transliterators perform transformations on text input based on a specific algorithm or set of rules. Transliterators can, for example, perform the following as the user types:

- Provide accent composition, such as combining the key sequence [a][¨] or [¨][a] into the single character [ä]. Transliterators allow this to occur without needing "dead keys," that is, key combinations such as alt-U that do not create a display until the character to be accented (the "a" in this case) is typed.
- Change the case of selected letters, such as capitalizing the first letter of each word or sentence
- Create "smart quotes," that is, replace straight quotation marks with left and right quotation marks as appropriate
- Provide phonetic transcription between scripts, for example, between Latin and Greek or between Romaji and Kana

Most of the transliterators provided by the CommonPoint system are rule-based, meaning that they use a table of rules to determine how to modify the text. Each rule has up to four fields defining the input text, the output text, and, optionally, preceding and succeeding contexts that allow the rules to be context sensitive. Each rule field can contain up to 256 characters, allowing the creation of very specific rules.

Table 2 lists example fields for the very simple transliteration operation of changing straight quotation marks to left and right quotation marks. The rules are traversed in order, and once a rule is applied the transliteration is complete. "NIL" could be specified for the preceding context in the second rule in Table 2, because if the criteria for the first rule is not met the second rule must be applied.

Range variables can be specified that provide a limited amount of wildcard matching for rule input and context fields. For example, a variable might be defined to represent all uppercase letters (the range A–Z) or all vowels (the set aeiouAEIOU). Inverse rules can be created that are applied if a character in the variable range or set does not appear in the

Figure 6 Converting user keystrokes into character strings

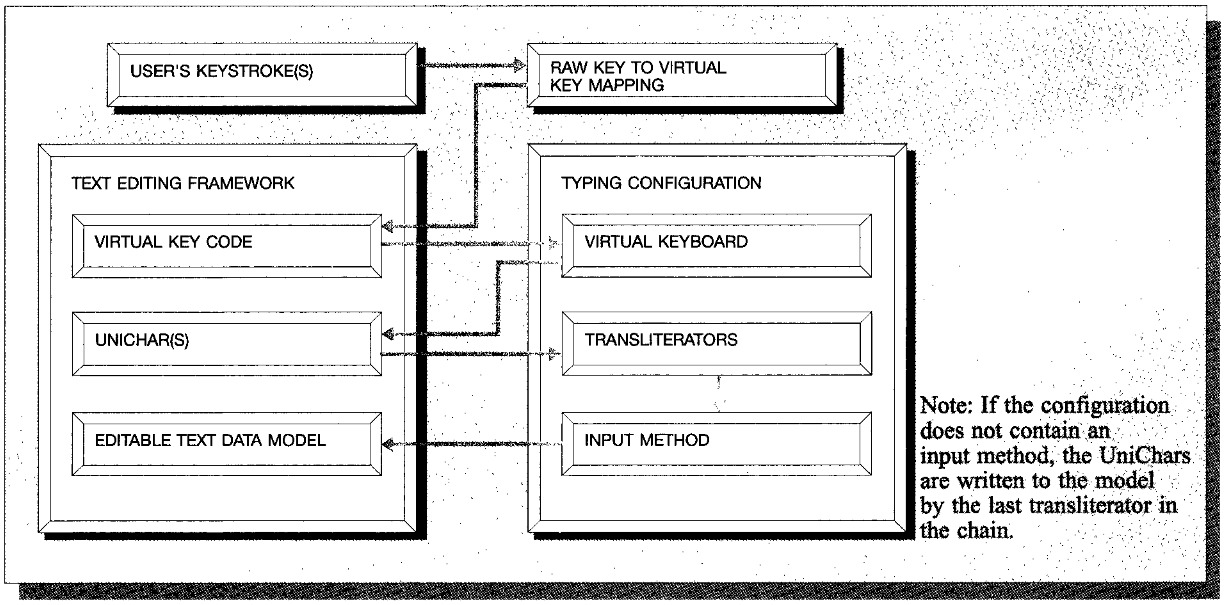

USER'S KEYSTROKE(S)
RAW KEY TO VIRTUAL KEY MAPPING
TEXT EDITING FRAMEWORK
VIRTUAL KEY CODE
UNICHAR(S)
EDITABLE TEXT DATA MODEL
TYPING CONFIGURATION
VIRTUAL KEYBOARD
TRANSLITERATORS
INPUT METHOD
Note: If the configuration does not contain an input method, the UniChars are written to the model by the last transliterator in the chain.

Table 2 Sample rules for transliteration

Rule	Input	Result	Preceding Context	Succeeding Context
Change straight quotation mark to right quotation mark when ending a quotation	"	”	Letters, punctuation ...	NIL
Change straight quotation mark to left quotation mark when beginning a quotation	"	“	NIL	NIL

input field. For example, all nonletter characters could be identified by creating an inverse rule based on the ranges a–z and A–Z.

Once a transliterator is built, it can be used anywhere in the system to perform text processing—even programmatically outside the typing configuration. For example, commands that change letters from lowercase to uppercase or that change the alphabet of selected text could use transliterators.

Input methods. Finally, the typing configuration can include an input method to allow phonetic entry of ideographic characters (used primarily by Chinese, Japanese, and Korean). Input methods often take the results of transliterators and provide more sophisticated processing. For example, a Japanese typing configuration can use transliterators to convert from a Roman transcription of Japanese text into Kana, and then use an input method to convert from Kana to Kanji. Input methods define both the technique for parsing the input and the user interaction model, for example, how alternate options for homonyms are chosen.

The CommonPoint input method framework allows input methods to be modular. The interface for an input method is the same no matter what type of document is being created or changed. The system currently includes an input method for Japanese; others can be added. The system also provides a porting interface to make it easier to port existing non-Unicode input methods.

Localization

The CommonPoint system includes specific services that support localization. The services support the creation of customized resources for a particular language, country, or region. Many of these resources are modular and can be added to the system at any time for use by any application. They include:

- Language-specific text analysis features such as collation, searching, and boundary analysis
- Language- or region-specific text-to-binary scanning and formatting for data types such as dates, times, and numbers
- Conditional formatting

Additionally, mechanisms are provided for collecting and accessing the resources for a particular region, including the ability to define fine-grained localizations and to archive program interfaces localized for a number of different presentation languages.

Collating and searching. Common sorting operations on strings rely on an ordering of the characters, called a collation order (often thought of as an alphabetic order). For example, if an alphabetic sequence specifies that [a] is less than [b], in an alphabetized list strings beginning with [a] would come before strings beginning with [b]. Collation orders are used to enable more natural sorting and searching than a simple ordering based on character codes can support. For instance, in the ISO Latin-1 character set, the code for [Z] is less than the code for [a] and the code for [z] is less than the code for [ñ], which leads to incorrect sorting results. Collation orders differ between languages, sometimes even when those languages use the same script—for example, French sorts differently than Swedish. Some languages also do not have a single "standard" collation order that can be used for every sorting operation.

Because of the requirements of natural language, collation orders must implement a number of features to provide intuitive sorting. The CommonPoint system defines a table-based approach for creating collation orders (instances of the class TTextComparator), based on the Unicode standard, that allows full functionality for language-sensitive sorting. To support this functionality, CommonPoint collation orders support:

- Ordering priorities for up to three levels of collation
- Normal or reverse "French" orientation
- Multiple character mappings for grouped or expanding characters
- Two levels of ignorable characters
- Ordering for unmapped characters

Ordering priorities enable collation features in which characters that are often considered equivalent for sorting purposes (for example, the uppercase and lowercase versions of a character) are sorted together as users expect. Simply reordering characters, such as ordering an uppercase [P] between the lowercase [p] and [q], is not sophisticated enough to produce the preferred results; with this scheme, for example, "put" would sort before "Pet" because the lowercase [p] is less than the uppercase [P].

Ordering priorities define the priority of the difference between any two adjacent characters in the ranking; differences between two characters can be primary, secondary, or tertiary. In the English collation order, for example, the difference between [a] and [b] is primary, the difference between [a] and [ä] is secondary, and the difference between [a] and [A] is tertiary. Other European languages may have other features that correspond to secondary or tertiary differences.

When comparing, secondary differences are considered only when there are no primary differences. So, for example, the secondary difference between [e] and [é] is used to sort the following strings:

resume
résumé
resumes

Likewise, tertiary differences are only considered when there are no primary or secondary differences.

The tertiary difference between [r] and [R] is used to sort the following strings:

resume
Resume
résumé

Note that ordering priorities can be used to control the sensitivity of searches. For example, case-insensitive searches in English text can ignore tertiary differences: resume and Resume would be considered equal.

The ability to specify orientation enables correct sorting of French, which attaches more weight to accent differences that occur later in the strings being compared rather than earlier in the strings. For example, the string "pêche" sorts before "péché," but the string "pécher" sorts before "pêcher." Whether to use the reverse French orientation for sorting only secondary differences or both secondary and tertiary differences can be specified.

Multiple character mappings allow correct comparisons when a single character expands to multiple characters (for example, the [ö] in German is sometimes mapped to [o][e]) or when multiple characters are grouped as a single character (for example, the [ch] character grouping in Spanish). This allows "Tönen" to be sorted before "Ton" in German, or "czar" before "chico" in Spanish.

Ignorable characters allow the identification of characters, such as punctuation marks or accents, that should be ignored in certain contexts. These are characters that generally represent a secondary or tertiary difference but can be ignored if there are no other differences in the string. For example, by identifying the hyphen as an ignorable character, the strings "blackbird" and "black-bird" would be considered equivalent.

Finally, the CommonPoint collation orders allow control of the ordering of characters that do not need to be mapped by the ordering object. Generally these are characters for which ordering is not relevant, such as dingbats or other symbols. For efficiency they can be ordered using the values of their Unicode character codes, or sorted before, within, or after the general ordering defined by the collation object.

Collation orders also provide a high level of parametrization, allowing programmers to exercise control over the collation process and to retrieve information about the results. For example, the programmer may need to know exactly the point at which differences occur between two strings so that common initial substrings can be identified; the collation object's comparison functions return this information. The CommonPoint collation orders implement the algorithm defined by Unicode Version 1.1[3] that identifies when two sequences are considered equivalent—for example, the composed sequence [a][¨] is equivalent to the precomposed character [ä]—allowing them to be collated identically.

These collation orders also form the basis for the CommonPoint text-searching mechanism. The system includes classes for objects that iterate through text, looking for a particular string, using collation orders to provide language-sensitive searching. The features described can be used to influence the search results—for example, in English a case-insensitive search can be done by telling the collation ordering object to ignore tertiary differences. The system implements an algorithm for sublinear searching that enables searching to be done very quickly, yet handles international comparisons.[7–11]

Collation ordering objects provide an efficient mechanism for building international sorting and searching features. Because collation orders are interchangeable, features can be created that are not dependent on a particular language. The program interfaces with the collation object, and different collation objects can be substituted to provide correct results for text in different languages. These operations also benefit tremendously from the use of Unicode—cross-language searches are more easily enabled because of the uniform character encoding. Because they are table-based, collation objects can be built easily by supplying the appropriate data; orderings that require more sophisticated processing, such as those requiring dictionary lookup, can be created by subclassing the abstract class TTextOrder. Collation objects can also be merged. For example, French and Arabic objects could be combined to create a collation order for use in North Africa. One of the original orderings would be specified as the "master" so that its rules take precedence if any conflicts arise. The rules from the "slave" are added wherever they do not conflict with the master.

Performing boundary analysis. Boundary analysis is done to programmatically break up a Unicode text string into logical text elements such as characters, words, lines, or sentences. This allows users to navigate through a display of characters one at a time using cursor keys, or to select a word, line, or sentence with a double or triple mouse-click. It also determines the necessary information to allow text to be dynamically formatted into lines.

This analysis is not trivial, especially given that the desired results may vary between regions. Often the text alone does not provide enough information to determine boundaries—for example, the ambiguous use of the period character as both an end-of-sentence indicator and within abbreviations can make it difficult to determine sentence boundaries correctly. However, heuristics can be applied that produce reasonable results in most cases, especially for user selections that do not need to be exact. The requirements can also change based on the operation being performed. For example, trailing spaces can be included in word elements, but for operations such as search-and-replace, the inclusion of trailing spaces as part of the word can cause the search to fail.

Rather than develop algorithms to search for the actual text elements (words, sentences, and so on), the CommonPoint system implements a mechanism that looks for the boundaries between those elements. This simplifies the necessary computation, allowing the elements to be identified more quickly. The CommonPoint system also gives the ability to override this mechanism for instances in which more sophisticated processing, such as dictionary lookup, is also required.

Boundary analysis for particular kinds of text elements is performed by iterators, each of which is based on a specification of where boundaries can occur for an element. The boundary specification defines different collections of characters and lists the rules for boundaries in terms of those character collections. The character collections can be specified either as a list of characters (literal characters or ranges of characters) or as a Unicode character property, as defined in the CommonPoint Unicode implementation (for example, letters or closing punctuation marks).

For example, a specification for determining sentence boundaries might define the following character collections:

- Paragraph and line separator characters
- Space separator character
- Nonspacing marks
- Closing punctuation
- Terminating characters ("!" and "?")
- Period character
- Capital letters (uppercase letters, title case letters, and noncased letters)
- Lowercase letters

The following rules could then be defined to determine sentence boundaries:

1. Always break after paragraph separators.
2. Break after sentence terminators, but include inside the sentence boundary any nonspacing marks, closing punctuation, and trailing spaces.
3. Handle periods separately because they may be within an abbreviation or number rather than a terminating character. Do not break the sentence after the period if it is followed by a lowercase letter instead of an uppercase letter.

An implementation of this specification is created by mapping characters to a type identifier (TypeID) and using the TypeIDs to determine the boundaries with a state table that expresses each rule as a state transition. This is encapsulated by an iterator object that can be used to iterate either forward or backward through a block of text, searching for boundaries of a particular type of text element. The system also includes a mechanism that ensures accuracy when the iteration is begun at a random point within the text. For example, if the cursor is placed in the middle of a sentence when iteration for sentence boundaries begins, the iterator will locate the starting boundary of the sentence before moving forward.

Scanning and formatting numbers and times. Text formatting is the process of converting binary data (such as a number) into a meaningful textual representation; scanning is the reverse operation. The CommonPoint system provides objects called formatters that perform both formatting and scanning for the system data types representing numbers and times. The formatter classes can also be subclassed to provide scanning and formatting capabilities for other data types.

Generally, localized instances of these formatters are created to provide the correct behavior for specific regions. For example, formatters could be created that produce a string representing the same binary number in either an American (9,999.99) or a French (9.999,99) format, or represent the number using a different numbering system (for example, Roman numerals). Because these formatters manipulate TText data, styling information can be used to express or interpret meaning. For example, negative numbers can be represented in red, or scientific notation in the format 1.23×10^3, where the superscripted number is always the exponent. Nondigit characters can be attached to the output, so, for example, number formatters could provide currency formats such as "$300.00" or "£19,999."

The CommonPoint system provides a set of number formatters for a variety of formats. These include floating-point numbers, scientific notation, Roman numerals, fractions, and Han numbering. These formatters give a great deal of control over the details of number formatting, for example, separator characters, rounding precision, zero-padding, and so on.

The system also includes formatters for converting the system's internal time representation into the correct local time in the desired format. The internal representation is a flow of time according to the international standard called Coordinated Universal Time (UTC) which is equivalent to Greenwich Mean Time. These formatters use several resources to produce a meaningful textual representation: a time zone defining the differential from GMT, a calendar object defining the calendaring system (for example, Gregorian or Arabic), and a pattern defining the fields of interest and their format. The same binary UTC data could be formatted into any of the following: "2:15 pm," "1415 hours," "Monday, May 1," "1-5-95," or other created patterns.

Allowing conditional formatting. CommonPoint formatters allow for conditional formatting and subformatting of text representations based on the value of certain fields. This allows, for example, the creation of messages that are grammatically correct. For example, a conditional formatter could generate the messages:

"You have not deleted any files."
"You have deleted 1 file."
"You have deleted 3 files."

instead of the generic message:

"You have deleted 1 file(s)."

To generate these messages, a special formatter called a parameter formatter is created that checks an input parameter (in this case, the parameter representing the number of files) and chooses the right text based on its value. This is particularly useful in more complex scenarios; for example, in some Slavic languages the plural form of a noun changes based on whether there are two items, three to five items, or more than five items. These formatters can also be used to format and scan the date and time patterns described above. A parameter formatter maps the numeric value of a field, for example, "1" in the "month" field, to a language-specific textual representation such as "January" or "Janvier."

Parameter formatters also enable flexible scanning by allowing the specification of alternate text matches for specific fields or ranges in the scan text. For example, when scanning formatted dates, the dates "1-1-99" and "1/1/99" should scan identically. Parameter formatters allow the character [/] to be specified as an alternate match for the [-] character.

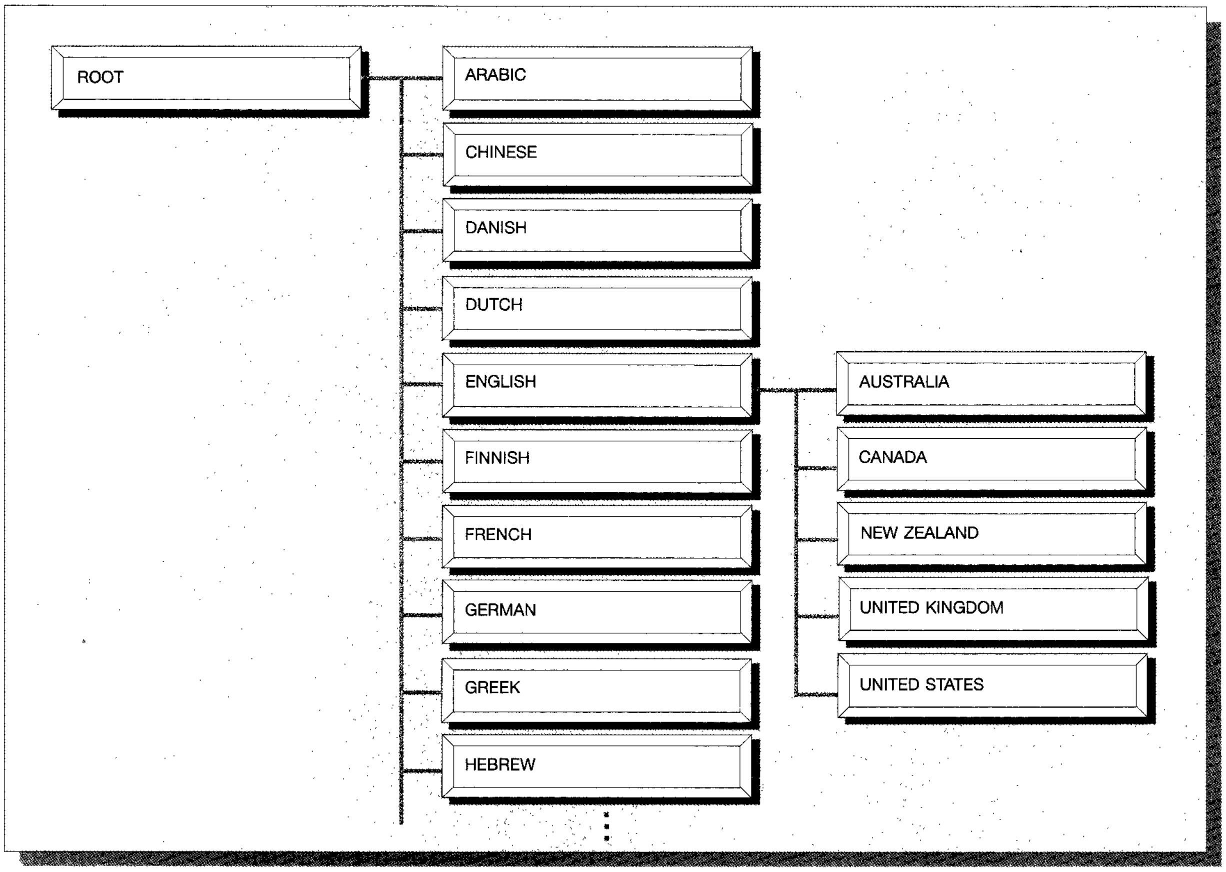
Enabling the localization process. To facilitate the process for localizing software, the CommonPoint system includes the locale mechanism. Conceptually, a locale represents a geographic region for which objects need to be localized—locale objects then collect all the objects for a particular locale. Locales are organized into a hierarchy that descends from a single root locale. The first level beneath the root generally represents languages, and levels beneath that represent increasingly fine-grained locales.

Figure 7 shows part of the CommonPoint locale hierarchy. This hierarchy allows regions to share common resources from a high-level locale rather than duplicating it. For example, the United States and the United Kingdom can share language-specific resources, while lower-level locales provide resources that are country-specific, such as date, number, and currency formats.

Locales are associated with open-ended collections of heterogenous locale objects. Objects commonly referenced by locales include a country identifier, a language identifier, a default typing configuration, font preferences, and a number of typical date and number formatters. New kinds of objects can be added to any locale at any time.

The locale hierarchy also helps organize multiple localized interfaces for applications. Each program has an associated archive that contains all the localizable interface elements, such as menu labels, sounds, and icons. The information in these archives is organized according to the locale hierarchy shown above, allowing it to contain multiple versions that correspond to different languages or regions. A program's archive could contain localized interfaces in French, Russian, English, and Japanese. The user

Figure 7 Part of the CommonPoint locale hierarchy



can then choose a preferred language, and the system presents the program interface in that language if it exists in the archive. If the program has not been localized to that language, the default presentation for the program is used—generally the language in which the program was originally created.

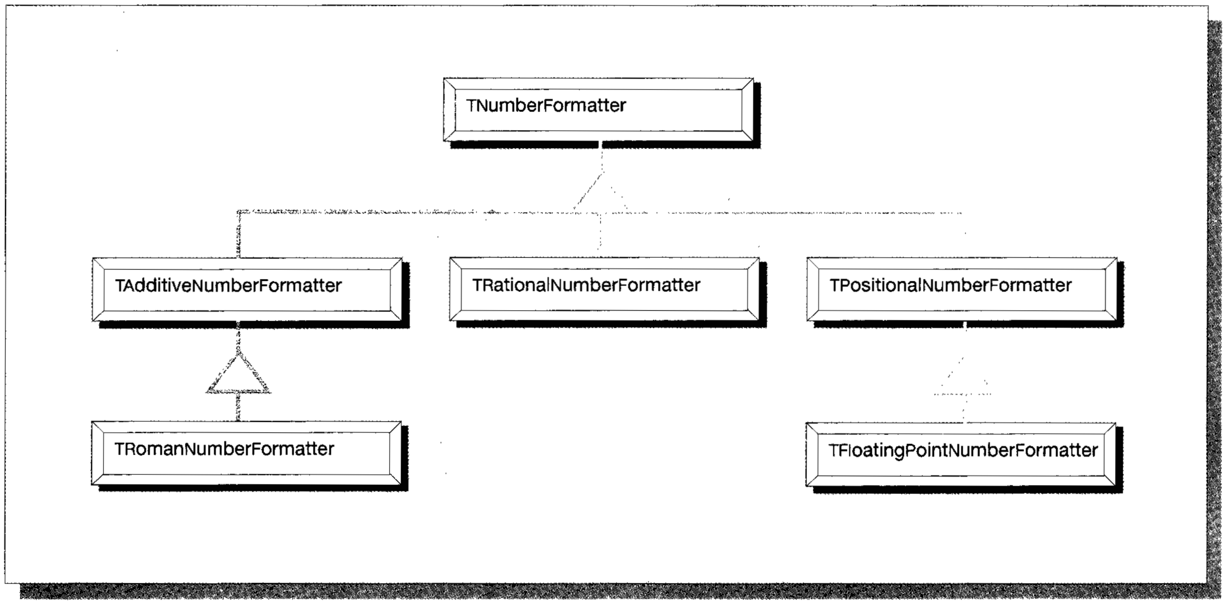
Taligent provides an interface development tool, the *cp*Constructor** User Interface Development Tool, that makes it easy to create an archive for a program's interface and to localize that interface for use in any number of locales.

Taking advantage of object-oriented technology

Object-oriented design philosophy provides several implementation advantages for the international software features described in this paper. The CommonPoint application system, including the text and international frameworks described here, derive several advantages from being object-oriented that are important to the development of international software: they are modular, data-driven, and tailorable.

The modular nature of an object-oriented system allows individual pieces of functionality to be added arbitrarily. This means that new localized resources can be added to the system at any time. They are immediately usable, and do not interfere with any already-installed objects in that locale. Multilingual or international users can choose between virtual keyboards or collation orders for different languages, number and date formatters for various standard formats, and so on. From the developer's standpoint, many of these resources are already available and can be used as-is in new programs, making the development of international software much more ef-

Figure 8 Class inheritance hierarchy for CommonPoint number formatters



ficient. Because these objects are independent of the locale, they can be accessed, or copied and altered. They can also be shipped to a server which may not even have the original locale.

Development is also more straightforward because all objects of a particular type behave the same way, regardless of what region or language they are localized for. Standard protocols are defined for each type of object, but the behavior of a particular instance is determined by the encapsulated data. For example, a collation ordering object can be relied on to provide protocol for comparing two strings—the result of the comparison, however, is determined by the language-specific data that define that collation order. New collation objects for other locales can be built easily, just by providing the correct table of collation rules, or an existing object can be customized by editing the collation rules. This customization is easily provided with any of the international objects that are table-based, including collation orders, text boundary specifications, and virtual keyboards. Other types of objects are parametrized to allow them to work with different data. For example, number formatters can be instantiated to work with the different sets of digits supported by the Unicode standard, such as Arabic or Bengali digits.

The inheritance mechanism provided by the object-oriented implementation can be used to create objects that are more finely tailored to specific needs. Subclassing allows preexisting data and functionality to be reused, so that programming efforts can be concentrated on providing more specialized or sophisticated features. The CommonPoint number formatters provide an example of how inheritance can be used to create increasingly specialized objects, as shown in Figure 8.

Higher-level classes provide more generic functions for converting between binary numbers and text representations. Classes further down in the hierarchy inherit that functionality and provide additional functionality that is more specific to particular formats. For example, the rational-number formatter provides functions for specifying whether to superscript and subscript the numerator and denominator portions of the fraction, while the floating-point-number formatter provides functions for setting formatting options for exponential notation.

The inheritance mechanism also makes customization easier when large amounts of data are involved. For example, the JIS character encoding set includes thousands of characters. Variations of JIS use slightly different encodings but are all algorithmically derived from the same basic standard. Instead of creating a different transcoder to encapsulate all the necessary data for each variation, a more efficient implementation could be created using inheritance. The transcoder for each variation could be subclassed from a base JIS transcoder class, and, instead of duplicating the data, functions can be overridden or added to create the variation algorithmically. The implementation is not only simpler but the derived classes can be used polymorphically.

Conclusions

Taligent's CommonPoint application system provides an example of how object-oriented principles can be applied to provide improved functionality in areas that are crucial to the development of global software—text handling, character input, and localization. The text system provides a model for multilingual text manipulation, as well as a simple mechanism for creating and collecting the resources that are appropriate for any given locale. The CommonPoint system's object-oriented implementation makes it easy to use, extend, and customize, giving developers a high degree of flexibility. Exclusive usage of the Unicode character encoding standard is also integral to the CommonPoint strategy for international software development, providing a much greater degree of integrity for character data.

The CommonPoint text and international frameworks described in this article comprise hundreds of classes and thousands of functions. This rich international feature set, combined with pure object-oriented implementation, make the CommonPoint application system a powerful foundation for developing global software applications that can be localized for fine-grained regions, yet remain integrated and compatible across modern international organizations.

*Trademark or registered trademark of International Business Machines Corporation.

**Trademark or registered trademark of Taligent, Inc. and Unicode, Inc.

Cited references and notes

1. This paper has been substantially revised and expanded from an earlier version published in *Objects in Europe, a Supplement to SIGS Publications* **2**, No. 2 (July–August 1995).
2. The Unicode Consortium, a nonprofit organization, was founded in 1991. Members of the consortium include major computer corporations, software producers, database vendors, research institutions, international agencies, and various use groups. For more information on the Unicode Consortium, see the Unicode home page at http://www.stonehand.com/unicode.html.
3. The Unicode Consortium, *The Unicode Standard: Worldwide Character Encoding, Version 1.0* (2 volumes), Addison-Wesley Publishing Co., Reading, MA (1991).
4. *The Unicode Standard, Version 1.1., Unicode Technical Report #4* (prepublication edition), The Unicode Consortium, Mountain View, CA (1993).
5. Taligent, Inc., *The Power of Frameworks*, Addison-Wesley Publishing Co., Reading, MA (1995).
6. S. Cotter and M. Potel, *Inside Taligent Technology*, Addison-Wesley Publishing Co., Reading, MA (1995).
7. R. Boyer and S. Moore, "A Fast String Searching Algorithm," *Communications of the ACM* **20**, No. 10, 762–772 (October 1977).
8. G. H. Gonnet and R. Baeza-Yates, *Handbook of Algorithms and Data Structures in Pascal and C, Second Edition*, Addison-Wesley Publishing Co., Wokingham, UK (1991).
9. A. Hume and D. M. Sunday, "Fast String Searching," *Software Practice & Experience* **21**, No. 11, 1221–1248 (November 1991).
10. P. D. Smith, "Experiments with a Very Fast Substring Search Algorithm," *Software Practice & Experience* **21**, No. 10, 1065–1074 (October 1991).
11. D. M. Sunday, "A Very Fast Substring Search Algorithm," *Communications of the ACM* **33**, No. 8, 132–142 (August 1990).

General references

Apple Computer, Inc., *Guide to Macintosh Software Localization*, Addison-Wesley Publishing Co., Menlo Park, CA (1992).

Apple Computer, Inc., *Inside Macintosh: Text*, Addison-Wesley Publishing Co., Menlo Park, CA (1993).

Designing NL Enabled Products, SE09-8001, IBM Corporation (1987); available through IBM branch offices.

NLS Reference Manual Release 4, SE09-8002, IBM Corporation (1994); available through IBM branch offices.

The Unicode Consortium, *The Unicode Standard, Version 2.0*, Addison-Wesley Publishing Co., Reading, MA (to be published in 1996).

Accepted for publication January 31, 1996.

Mark E. Davis *Taligent, Incorporated, 10355 North De Anza Boulevard, Cupertino, CA 95014-2233 (electronic mail: mark_davis@taligent.com).* Dr. Davis is the director of the Core Technologies department at Taligent. He received a B.A. degree from the University of California at Irvine in 1973 and a Ph.D. degree from Stanford University in 1979. After four years at Systime AG in Zurich, where he developed commercial software for multilingual text and data manipulation, he joined Apple Computer, Inc. There he coauthored the KanjiTalk Japanese Macintosh system and the Macintosh Script Manager and authored the Arabic and Hebrew Macintosh systems. He has been with Taligent from the time that it was formed. Dr. Davis cofounded the Unicode effort and is the president of the Unicode Consortium. He had a key role in the development of the Unicode Standard and the successful resolution of its merger with the ISO/IEC 10646 standard, as well as in its further development.

Jack D. Grimes *ICVERIFY, 473 Roland Way, Oakland, CA 94621 (electronic mail: jgrimes@icverify.com).* Dr. Grimes is Vice President, Engineering at ICVERIFY, a company that creates authorization software for credit and debit cards. He worked for Taligent from early 1992 to early 1996. Before joining Taligent he held executive positions with Mass Microsystems, Intel Corporation, and ITT Corporation. Over the past two decades he has worked as both an electrical and a software engineer, in engineering management, in marketing, and in advanced technology. Dr. Grimes earned his Ph.D. degree in electrical engineering and computer science and his M.S. and B.S. degrees in electrical engineering from the University of Iowa. His M.S. degree in experimental psychology was received from the University of Oregon. His many speaking engagements include tutorials on the psychology of user interface design, and he has published more than 45 papers on subjects ranging from visual perception to VLSI (very large scale integrated) graphics and object technology.

Deborah J. Knoles *Taligent, Incorporated, 10355 North De Anza Boulevard, Cupertino, CA 95014-2233 (electronic mail: debbie_knoles@taligent.com).* Ms. Knoles has been a member of the Technical Communications department at Taligent since shortly after its inception in 1992, with responsibility for documentation and training materials for the CommonPoint text and international frameworks. She previously spent three years at IBM, supporting system software products. Ms. Knoles holds a B.A. degree in English from Stanford University.

Reprint Order No. G321-5603.