(19)日本国特許庁(JP)　(12)特　許　公　報(B2)　(11)特許番号
特許第3599740号
(P3599740)
(45)発行日　平成16年12月8日(2004.12.8)　(24)登録日　平成16年9月24日(2004.9.24)

(51)Int.Cl.[7]	F I
G06F 5/00	G06F 5/00 Z
G06F 17/21	G06F 17/21 592Z

請求項の数 28　(全 26 頁)

(21)出願番号	特願平7-519538
(86)(22)出願日	平成6年3月11日(1994.3.11)
(65)公表番号	特表平9-508726
(43)公表日	平成9年9月2日(1997.9.2)
(86)国際出願番号	PCT/US1994/002628
(87)国際公開番号	WO1995/020198
(87)国際公開日	平成7年7月27日(1995.7.27)
審査請求日	平成13年2月27日(2001.2.27)
(31)優先権主張番号	08/184,127
(32)優先日	平成6年1月21日(1994.1.21)
(33)優先権主張国	米国(US)

(73)特許権者
オブジェクト　テクノロジー　ライセンシング　コーポレイション
アメリカ合衆国　９５０１４―２０８３　カリフォルニア州　クパチーノ　メールステーション　３８―オーティーエル　インフィニト　ループ　１
(74)代理人
弁理士　谷　義一
(74)代理人
弁理士　阿部　和夫
(72)発明者　デイヴィス，マーク，イー．
アメリカ合衆国　９５０１４　カリフォルニア州　クパチーノ　ポターズ　ハッチカモン　１０１７４

最終頁に続く

(54)【発明の名称】数値フォーマッティング・フレームワーク

(57)【特許請求の範囲】

【請求項１】

プロセッサおよびストレージを有するコンピュータシステムにおいて使用され、第１の数形式と２進数との間で数値情報を変換する装置であって、

（ａ）前記ストレージ内に設けられ、標準中間形と２進数との間で数値情報を翻訳する規準数値フォーマッタオブジェクトと、

（ｂ）前記ストレージ内に設けられ、１つのクラス階層で配列された複数の数値フォーマッティングクラスで定義される複数の数値フォーマッティングオブジェクトを有する数値フォーマッタフレームワークであって、

前記複数の数値フォーマッティングクラスの各々は、複数の第１の数形式の１つと標準中間形との間で数値情報を変換するためのオーバーライド可能な変換方法を有しており、

（ｃ）所定の第１の数形式に基づいて前記複数の数値フォーマッティングオブジェクトの中から１つの変換方法を抽出することによって、前記数値情報を変換する手段と

を具えたことを特徴とする装置。

【請求項２】

前記複数の数値フォーマッティングクラスの各々は、

テキストから少なくとも１つの数に変換するための少なくとも１つのテキスト／数値変換方法を含むことを特徴とする請求項１記載の装置。

【請求項３】

前記テキスト／数値変換方法は、

テキストから標準中間形に変換するための少なくとも１つの方法を含み、
前記規準数値フォーマッタオブジェクトは、
標準中間形から少なくとも１つの２進数に変換するための少なくとも１つの方法を含むことを特徴とする請求項２記載の装置。
【請求項４】
前記複数の数値フォーマッティングクラスの各々は、
数値情報をテキストに変換するための少なくとも１つの数値／テキスト方法を含むことを特徴とする、請求項１記載の装置。
【請求項５】
前記少なくとも１つの数値／テキスト方法は、
信頼度表示を提供するための手段を含むことを特徴とする請求項４記載の装置。
【請求項６】
前記数値／テキスト変換方法は、
標準中間形からテキスト形式に数値情報を変換するための少なくとも１つの方法を含むことを特徴とする請求項４記載の装置。
【請求項７】
前記複数の数値フォーマッティングクラスの各々は、
言語特有の数値を管理するための少なくとも１つのオブジェクトを定義するためのクラスコードを含むことを特徴とする請求項１記載の装置。
【請求項８】
前記複数の数値フォーマッティングクラスの各々は、
正数および負数をフォーマッティングするための少なくとも１つの方法を含むことを特徴とする請求項１記載の装置。
【請求項９】
前記複数の数値フォーマッティングクラスの各々は、
範囲外処理を管理するための少なくとも１つの方法を含むことを特徴とする請求項１記載の装置。
【請求項１０】
前記数値情報を変換する手段は、
（ａ）前記第１の数形式の数値情報に応答して、前記複数の数値フォーマッティングオブジェクトの中から第１の変換方法を抽出する手段であって、
該第１の変換方法によって、数値情報の第１の変換を試み、該第１の変換の試み結果を示す信頼度値を回復させ、
（ｂ）前記第１の数形式の数値情報に応答して、前記複数の数値フォーマッティングオブジェクトの中から第２の変換方法を抽出する手段であって、
前記第１の変換が所定の基準を満たさないようなある信頼度値になったとき、該第２の変換方法によって第２の変換を実行する
ことを特徴とする請求項１記載の装置。
【請求項１１】
前記数値フォーマッタフレームワークは、文字の値をマッピングする少なくとも１つのオブジェクトを定義するクラスコードを含むことを特徴とする請求項１記載の装置。
【請求項１２】
数をテキストで表現した一切のものを動的に位置合わせするディスプレイ手段を含むことを特徴とする請求項１記載の装置。
【請求項１３】
前記複数の数値フォーマッティングクラスの少なくとも１つは、
加法形式の数値情報をフォーマティングする方法を有することを特徴とする請求項１記載の装置。
【請求項１４】
前記複数の数値フォーマッティングクラスの少なくとも１つは、

位取り形式の数値情報をフォーマッティングする方法を有することを特徴とする請求項１記載の装置。

【請求項１５】

プロセッサおよびストレージを有するコンピュータシステムにおいて使用され、第１の数形式と２進数との間で数値情報を変換する方法であって、

（ａ）前記ストレージ内において、標準中間形と２進数との間で数値情報を翻訳する規準数値フォーマッタオブジェクトを構成するステップと、

（ｂ）前記ストレージ内において、クラス階層で配列された複数の数値フォーマッティングクラスで定義される複数の数値フォーマッティングオブジェクトを有する数値フォーマッタフレームワークをストアするステップであって、

前記複数の数値フォーマッティングクラスの各々は、複数の第１の数形式の１つと標準中間形との間で数値情報を変換するためのオーバーライド可能な変換方法を有しており、

（ｃ）所定の第１の数形式に基づいて前記複数の数値フォーマッティングオブジェクトの中から１つの変換方法を抽出することによって、前記数値情報を変換するステップと

を具えたことを特徴とする方法。

【請求項１６】

前記第１フォーマットは、テキストであり、

ステップ（ｃ）は、

（c1）テキスト形式の数値情報を少なくとも１つの２進数に変換するステップを含むことを特徴とする請求項15記載の方法。

【請求項１７】

ステップ（c1）は、

（c1a）テキストを標準中間形に変換するステップと、

（c1b）前記標準中間形を少なくとも１つの２進数に変換するステップとを含むことを特徴とする請求項16記載の方法。

【請求項１８】

ステップ（c1a）は、

（c1）前記２進数の数値情報をテキスト形式に変換するステップを含むことを特徴とする請求項15記載の方法。

【請求項１９】

前記ステップ（c1）は、

信頼度表示を提供するステップを含むことを特徴とする請求項18記載の方法。

【請求項２０】

ステップ（c1）は、

（c1a）少なくとも１つの２進数を標準中間形に変換するステップと、

（c1b）前記標準中間形をテキスト形式に変換するステップと

を含むことを特徴とする請求項18記載の方法。

【請求項２１】

ステップ（ｃ）は、

（c1）言語特有の数値を有する前記第１のフォーマットの数値情報を、２進数に変換するステップを含むことを特徴とする請求項15記載の方法。

【請求項２２】

ステップ（ｃ）は、

（c1）正数および負数をフォーマッティングするステップを含むことを特徴とする、請求項15記載の方法。

【請求項２３】

ステップ（ｃ）は、

（c1）範囲外処理の状態を検出するステップと、

（c2）範囲外処理の状態を管理するステップと、

を含むことを特徴とする請求項15記載の方法。

【請求項２４】
ステップ（ｃ）は、
（c1）数値情報の第１の変換を試み、該変換に基づいて信頼度値を生成するステップと、
（c2）前記信頼度値が所定の規準を満たさないとき、第２の変換を実行するステップと
を含むことを特徴とする請求項15記載の方法。
【請求項２５】
ステップ（ｃ）は、
（c1）数値情報をテキストで表現した一切のものを、動的に位置合わせして表示するステップを含むことを特徴とする請求項15記載の方法。
【請求項２６】
ステップ（ｃ）は、
文字の値をマッピングするステップを含むことを特徴とする請求項15記載の方法。
【請求項２７】
ステップ（ｃ）は、
加法形式の数値情報をフォーマッティングするステップを含むことを特徴とする請求項15記載の方法。
【請求項２８】
ステップ（ｃ）は、
位取り形式の数値情報をフォーマッティングするステップを含むことを特徴とする、請求項15記載の方法。
【発明の詳細な説明】
著作権表記
本特許出願の一部には、著作権保護の対象となる内容が含まれている。著作権所有者は、米国特許商標庁の特許ファイルまたはレコードに記録されている特許文書または特許開示内容を何人もが原文通りに複製することを妨げるものではないが、その他の場合は一切の著作権を所有することを留保する。
発明の分野
本発明は一般的にはコンピュータ・システムにおける改良に関し、より具体的には数値データをフォーマッティングするシステムおよび方法に関する。
発明の背景
現代のコンピュータ・システムは、以前には得られなかった広範囲にわたる効率性を提供するものである。ワード・プロセッサ、スプレッドシート、データベース・マネージャ、ファイナンス・マネージャ、およびその他のアプリケーションの使用は、以前に見られなかった成長を享受している。この成長は本質的に地球的規模にわたっている。現在では、アプリケーションはスタンド・アロン（独立型）ユニットとして代表的に設計され、すべてではないとしも、特定のアプリケーションに関係する処理の大部分を取り扱っている。時には、アプリケーションは基本的入出力の実行をディスク・オペレーティング・システムに要求する場合もあるが、処理の大部分はアプリケーション・プログラムによって実行されている。
計算において最も共通しているデータ・タイプの１つは数値データ（numerical data）である。ほとんどすべてのアプリケーション・プログラムはデータの処理と操作を必要としている。各アプリケーション・プログラムは数値データを内部で処理しているので、数値データの扱い方は様々である。そのために、アプリケーション・プログラム間に非互換性（incompatibilities）という問題がしばしば起こっている。もう１つの問題は、地球的文化（global culture）が縮小していることから起こっている。世界の通貨制度（monetary system）は様々であり、通貨情報を伝達するために使用される表記法（notaition）と文法も、それに応じて様々になっている。この多様性の結果として、数値で表された通貨情報はその受取人に理解されるように適当に作り替える必要があるために、世界各国間で情報を転送することが複雑化している。通貨その他の用途に使用される単位が異なると、その取扱いが困難であることは、IEEE Transactions on Software Engineering,vol.Se－

8,no.6,November,1982,New York,US,pages 605－611,N.H.Gehani,Databases and units of Measureに要約して概説されている。この文献は、意味指向のデータベースを可能にさせるセマンティック・データ・モデリングを使用して、データベース・テクノロジ分野において測定単位が異なっているが、等価である値間の変換を行うためのアプローチも論じている。

他にも、数値情報が維持されるときの基本形式に起因する問題がある。この問題は、数（number－数値ともいう）がテキストとして、つまり、数学的観点から見たとき扱いづらい形式で始まっているときよく起こっている。数値情報は扱いやすい形式に、つまり、テキスト以外の形式に変換する必要がある。上述した問題の他にも、数値データの非整合性（inconsistencies）がないという問題があり、この問題も多様化している。しかし、ある設定値（setting）の中に現れる数が別の設定値の中で使用されるのは容易でないとの認識があることも確かである。それがアプリケーション・プログラム間であるか、地理上の地域間であるか、ある記憶形式と別の記憶形式間であるか、通貨制度間であるか、数体系の形式間であるかに関係なく、非整合性があると、常に非効率性の問題が起こっている。

非オブジェクト指向フレームワークの仕方でこれらの問題を解決する１つの方法は、PCT国際公開WO－Ａ－93 21591（TCS）に説明されている。この文献には、独立の変換ルーチン（パイプ）を提供してデータ転送を行う一方、特定の変換を行うことによって数値変換を行う装置／方法が開示されている（この装置／方法はオブジェクト指向プログラミング・メソッドを使用している）。これらのパイプは、パイプを使用するアプリケーションを変更しなくても、新しいパイプをそのアプリケーションに追加することが可能なモジュラ（modular）になっている。従って、この発明によれば、パイプを通してデータを送受するアプリケーションから独立して、変換ルーチンをパイプに組み入れるように編成することにより新しい変換を追加することが可能になっている。そのために、この発明によれば、アプリケーション・システム全体を再コンパイルし、リンクし、テストしなくても、新しいデータ変換機能を、通信し合うアプリケーションのシステムに追加することが可能になっている。従って、新しいデータ変換ルーチンは、そのシステムの既存機能に影響を与えることなく、開発し、テストし、アプリケーションのシステムに組み入れることが可能になっている。しかし、この文献には、変換がパイプ内でどのように行われるのか、オブジェクト指向フレームワークをどのように設計するとこれらの変換を行うことができるのか、については説明されていない。この文献は、既存プログラミングをパイプの終端で変更するのではなく、新しいパイプをアプリケーションのシステムに追加することによって将来の拡張を可能にする複数のプログラム（パイプ）を使用してデータを変換する手段を説明しているだけである。

以上のような理由から、必要なことは、数値情報の非整合性を除去またはその除去を支援することであり、少なくとも、数値情報をある体系から別の体系へ変換する効率的なシステムおよび方法を提供することである。

発明の概要

上記に鑑みて、本発明の主目的は、改善された数値フォーマッティングを提供するシステムおよび方法を提供することである。

本発明の別の目的は、数値情報をある体系から別の体系へ変換するためのツールを提供することである。

さらに、本発明の別の目的は、数値情報および関連情報をある体系から別の体系へ変換するためのツールを提供することである。

さらに、本発明の別の目的は、ある言語の数値情報を別の言語の数値情報へ変換することを提供することである。

本発明の好適実施例によれば、種々の数値フォーマティング・サービスを提供するオブジェクトのシステムが開示されている。オブジェクトには、テキストをスキャンし、そのテキストを数値情報に変換するものがある。他のオブジェクトは非テキストの数値情報をテキストに変換するものがある。数値形式（フォーマット）間の変換と言語間の変換もサポートされている。このシステムは共通基盤を提供するので、アプリケーション・プログラ

ムは、数値情報が表現体系（systems of representations－記数法ともいう）間でどのように変換されるかの詳細を知らなくても、外見上共通の数値言語で相互に対話（speak）できるようになっている。本発明によれば、アプリケーション開発者は数値情報の詳細を取り扱う負担から軽減される。このシステムによれば、通貨制度間および異種言語をもつ文化間で情報を転送することも容易化される。さらに、このシステムによれば、数値情報の処理および／または理解を容易にするために数値形式間の変換の提供もできる。上記およびその他の目的と利点は下述する説明において明らかになる。

【図面の簡単な説明】

図１は、好適実施例によるコンピュータの代表的なハードウェア構成を示す図である。

図２は、好適実施例による数値フォーマッティングで使用される種々のクラス間の関係を示すBooch図である。

図３は、好適実施例によるテキスト全体から数への変換プロセスを示す図である。

図４は、好適実施例による数からテキストへの変換を示す図である。

図５は、好適実施例による２つの数値フォーマッタを使用する変換を示す図である。

図６は、好適実施例による最適一致アルゴリズム（best match algorithm）を使用する２つのフォーマッタを示す図である。

図７は、好適実施例によるバイナリとテキスト間の２ステップ変換プロセスを示す図である。

図８は、好適実施例によるConversionResultオブジェクトとこのオブジェクトが維持するデータを示す図である。

発明の詳細な説明

以下、本発明の実施例について説明する。なお、開示された実施例は本発明の単なる例示であり、本発明は種々態様に具現化できることは勿論である。従って、以下に開示されている詳細は限定的なものではなく、請求の範囲の基礎となると共に、本発明の製造方法および／または使用方法を当業者に教示するときの単なる基礎となるものである。オブジェクト指向プログラミングの歴史とフレームワークの開発は文献に詳しく説明されている。また、Ｃ＋＋とSmalltalkは文献に詳しく説明されているので、ここで詳しく説明することは省略する。同様に、カプセル化（encapsulation）、多相性（polymorphism）および継承（inheritance）といった、オブジェクトの特性は文献や特許明細書に詳しく説明されている。オブジェクト指向システムを解説した参考文献として、Grady Booch著「オブジェクト指向設計とアプリケーション（Object Oriented Design With Applications）」がある。

多くのオブジェクト指向システムは、基本的入出力を実行する基本オペレーティング・システムの上で動作する設計になっているが、本発明を使用すれば、特定のフィーチャ（feature－機能ともいう）に対するシステム・レベル・サポートを提供することができる。なお、留意すべきことは、ここで開示されている革新的なオブジェクトは、処理階層（processing hierarchy）の異なるレベルでオブジェクトのサポートを提供するために、システム・レベルより上位の層（レイヤ）に現われることも可能であることである。本明細書、請求の範囲および添付図面で使用されている「数値情報（numerical information）」には、数（number）に関係する一切の情報を含むことができる。この中には、数を表すテキスト、数を表すシンボル（記号）、種々の数体系（number system）、数に関連するテキスト、および数に関連するシンボルが含まれる。数値情報は、数に関する意味を伝達する情報ならば、どのような情報であっても構わない。一般的に、本発明が目的としているのは、ここで明示的に説明されており、数に関する情報を伝達するものとして知られている分野の情報間の変換である。

本発明は、好ましくは、IBM（登録商標）PS/2（登録商標）またはアップル社（登録商標）マッキントッシュ（登録商標）コンピュータなどのパーソナル・コンピュータに常駐したオペレーティング・システムを背景にして実施される。図１は代表的なハードウェア環境を示しもので、本発明によるコンピュータの代表的なハードウェア構成を示していて、このハードウェア構成は、従来のマイクロプロセッサなどの中央演算処理ユニット10と、

システム・バス12を介して相互接続された複数の他のユニットとを含んでいる。図１に示すコンピュータはリード・オンリ・メモリ（ROM）16、ランダム・アクセス・メモリ（RAM）16、ディスク・ユニット20などの周辺デバイス、および21で示す他の入出力周辺デバイスをシステム・バス12に接続するための入出力アダプタ18、キーボード24、マウス32、スピーカ28、マイクロホン26および／またはタッチ・スクリーン・デバイス（図示せず）などの他のユーザ・インタフェース・デバイスをバス12に接続するためのユーザ・インタフェース・アダプタ22、ワークステーションを23で示すデータ処理ネットワークに接続するための通信アダプタ34を含んでいる。バスをディスプレイ・デバイス38に接続するためのディスプレイ・アダプタ36も含んでいる。ワークステーションには、アップル・システム/7（登録商標）オペレーティング・システムなどのオペレーティング・システムが常駐している。

本発明によれば、数に関する変換サービスを提供するクラスの集合が説明されている。数を抽出するためのテキスト・スキャニング（text scanning）と数をテキストに変換するための数値フォーマッティングは共に同じクラスによって取り扱われる。数形式（number format）は、大部分の数値形式（numerical format）用と異種の言語用に用意されている。数値フォーマッティング・クラス（number formatting class）を使用すると、2.99792458×10^8や（1,234.56）といった数を正しくフォーマッティングすることも可能であるが、ただし、かっこは負数を示すために使用される。図２は、数値フォーマッティング・システムにおけるクラス関係を示す概要図である。数値フォーマッティング・クラスはTCanonicalNumberFormatter202とTCanonicalNumber204クラスを内部で使用する。数値フォーマッティング・クラスはテキストをTCanonicalNumber204に変換し、そのあとTCanonicalNumberFormatter202を使用して、バイナリ・レベルへのさらなる変換を行う。新しい数値フォーマッティング・クラスを作成した人だけがTCanonicalNumberFormatter202、TCanonicalNumber204、およびそれぞれのメソッドをアクセスする必要がある。数形式の中には、アウトラインにローマ数字を使用するといった特殊用途や序文におけるページの番号付けを目的としたものである。基底数形式クラスはTNumberFormat218であり、これは抽象基底クラス（abstract base class）である。以下に示したのは、この基底クラスでプロトコルを定義している最も重要なメソッド（method）である。

```
void TextToNumber (const TText& theText, TConversionResult& theAccuracy,
        long double& theNumber, TRange& theTextRangeToUse).
```

図３は、テキスト全体を数に変換するプロセスを示している。これは、TTextオブジェクトを数に変換するときの最も共通的なスキャニング・メソッドである。このメソッドは300でTTextを入力として受け取り、302でテキストを変換し、スキャンした数を返す。テキストがどれだけよく解析（parse）されたかの情報は、ステップ304でTConversionResult200に入れて返される。TRangeオブジェクトはテキストのどの部分を変換するかを決定するために使用される。デフォルト（default）では、テキスト全体が変換される。

次のようなメソッド群も用意されている。

```
long double TextToLongDouble (const TText& theText, TConversionResult&
        theAccuracy, TRange& theTextRangeToUse);

double TextToDouble (const TText& theText, TConversionResult& theAccuracy,
        TRange& theTextRangeToUse); and
```

これらのメソッドは、変換された数を関数コール（function call－関数呼出しともいう）や正確な数値のタイプ（type of number）が分かっている類似の場合で使用するときに利用する便利である。

```
void NumberToText (const long double, TText&, TConversionResult&).
```

図４は数をテキストに変換する場合を示している。このメソッドは400で入力された数からの変換を処理し、402でその数をそのテキスト表現（textual representation）に変換

する。TConversionResult200は404で返され、変換が成功したかどうかを示している。例えば、1.618のような数をローマ数字で書こうとすると、問題が起こるおそれがある。（不完全な変換を示すために例外がNumberToText（）によって引き起こされ（throw）ない理由は２つある。その１つは、ある種の数形式はすべての種類の数を単にサポートしているのではない、というものである。少なくとも一部の変換を行うことは可能であるので、このケースでは、1.618は“ I"に変換されるが、これはエラーではなく、例外は引き起こされない。）この場合も、異種の数値タイプにはオーバロード形式（overloaded forms）が存在する。

```
virtual GCoordinate MaxWidth (long double minNum, long double maxNum,
        const TStyleSet& styleSet) = 0;
```

すぐ上に示したメソッドは数の範囲を入力として受け取り、その範囲内のある数が、styleSetを使用して表示されるときにとることができる最大幅を返す。これは純仮想メソッド（pure virtual method）であり、いずれかの派生クラス（derived class）でオーバライドされなければならないものである。

```
virtual long MaxCharCount (long double minNum, long double maxNum) = 0;
```

このメソッドは、数の範囲が与えられているとき、その範囲内のある数がTTextに変換されるときとることができるUniChar文字の最大数を返す。これは純仮想メソッドであり、いずれかの派生クラスでオーバライドされなければならないものである。

正符号と負符号

正の数字と負の数字のそれぞれのフォーマッティングは、フォーマッティングされる数字の先頭または最後にテキストを付加することにより行われる。ある種のメソッドは、正または負の数字のどちらかの側に使用されるテキストおよび関係プロパティを指定するために使用される。すべての数字の前後にその符号を示す、なんらかのテキストが付加されることがこれらのメソッドが仮定しているデフォルトの振舞い（behavior）になっている。このテキストは空である場合と、空でない場合とがある。負の数字は特定のテキストが常に付加されている。正の数字はそうである場合と、そうでない場合とがある。これはSetShowPlusSign（）によって制御される。従って、よく知られた振舞いでは、負の数字の先頭には負符号が付けられ、最後にはなにも付けられないことになる。正の数字は、先頭に“＋”を挿入すると目立つようにすることができる。

すべての数字フォーマッティング・クラスが、このデフォルトの振舞いを使用する必要があるとは限らない。TRomanFormat230のようなクラスは、プレフィックス（接頭部）とサフィックス（接尾部）テキストに対して空のストリングを指定することによってデフォルトの振舞いを無視している。他のクラスは、以下に示すようにデフォルトの振舞いをセットアップするTNumberFormat218におけるメソッドをオーバライドすることができる。

```
Boolean UsesStandardNegation()
        //    デフォルトでは常にTRUEが返される
NormalizeSign(TStandardText& text, TConversionResult& result,
    Boolean& isNegative)
        // isNegativeがTRUEならば負のプレフィックスと
        // サフィックス・テキストをテキストに付けて結果を更新する
        // isNegativeがFALSE で  ::GetShowPlusSign() が
        // TRUEならば、正のプレフィックスとサフィックス・
        // テキストをテキストに付けて結果を更新する
ConfirmSignPrefix (const TStandardText& text, TRange& range,
    TConversionResult& result, Boolean& negFlag)
ConfirmSignSuffix (const TStandardText& text, TRange& range,
    TConversionResult& result, Boolean& negFlag)
```

正の数字の場合のデフォルトのアフィックス（affixes）は、負の数字の場合のデフォルトのサフィックスと同じように空のストリングである。しかし、負の数字の場合のデフォルトのプレフィックスは“－”であるが、これは、その外見にもかかわらずだれもが知っておりASCIIで親しみのある負符号／ハイフンではなく、外見上負符号と同じにしたものである。

範囲外（Out of Bounds）

すべての数形式は、有効な結果が得られる関連づけられた数値範囲を有している。例えば、ローマ数字は１以下の数を表すために使用されないのが一般であり、5000を越える数を表すためにはまれにしか使用されない。数フォーマッタ・メソッドの範囲に関連して使用されるメソッドとして、次の５つがある。

```
virtual Boolean         IsValidNumber    (long double) const;
        //    数は最小値と最大値の間にあるか？
virtual long double     GetMinNumber     () const;
virtual long double     GetMaxNumber     () const;
virtual void            SetMinNumber     (long double);
virtual void            SetMaxNumber     (long double);
```

図５は、数フォーマッタを使用して数をテキストに変換するプロセスを示している。ステップ500において、範囲外の数（out of range number）をテキストに変換する試みが失敗したことが502で検出されると、汎用数値フォーマッタ（universal number formatter）が504で使用される。各数値フォーマッタは、そのような場合にだけ使用される別の数値フォーマッタを含んでいる。この範囲外数値フォーマッタ（out of bounds number formatter）はTUniversalNumberFormatterがデフォルトであり、どの数でも扱えるように保証されている。

これと同じようなことは、図６に示すようにテキストを数に変換する場合にも起こる。各数値フォーマッタは、600で、自身でテキストを数に変換することを試みる。テキスト全体を数に変換できないときは、範囲外数形式がなにを行うことができるかを602,604で確かめて、２つのうちのより一致する方（つまり、オリジナル・テキストを多く使用している方）を606で返す。変換が反対方向に行われる場合は、このプロセスは逆に行われる。

数値フォーマッタで使用される範囲外数形式（out of bunds number format）は、次のメソッドでセットすることができる。

```
void  AdoptOutOfBoundsNumberFormat (TNumberFormat *);
```

なお、ここでは採用セマンティックス（adopt semantics）を使用している。つまり、数形式は採用された範囲外形式をストアする責任を受け持つと仮定する。最初に採用された数形式よりも小さい範囲をもつ範囲外数形式はほとんど無意味であるので、AdoptOutOfBoundsNumberFormat（）は、それを採用する前に、範囲外数形式がより大きな範囲をもっていることを決定する。

TConversionResult200、TNumeral206、およびTCanonicalNumber204

TCanonicalNumber204

テキストとバイナリ数との間の変換プロセスをアルゴリズム的に分けると、２つの異なる部分からなっている。一方は主に、丸め（rounding）といった数値フォーマッティングに係わりのある部分であり、他方は書き方のシステムの仕組み（mechanis of writing system）を扱う部分である。TNumberFormatterクラス218は、図７に示すように、プロセスを次の２ステップに分割することによって数値をフォーマトする。700での２進数と標準中間形（standard intermediate form）との間の変換（TCanonicalNumber204）と702での標準中間形とテキストとの間の変換である。

TConversionResult200

TConversionResultは２つのクラス、TScanResultとTFormattingResultにより実現されて、より管理された（directed）変換結果のクラスを提供する。

数値フォーマッティング・クラスは例外を多用していない。その理由は２つある。１つの理由は誤りは実際には例外的でないことである。ユーザはよくない形の（ill－formed）数字をタイプし、それが２進数に変換されるものと期待することが時々ある。また、ユーザは不適切な形式を使用しても２進数がテキストに変換されるものと期待するのも普通のことである（例えば、ローマ数字を使用してπを書くような場合）。

例外が多くあてにされないもう１つの理由は、完全な変換が不可能な場合であっても、部分的な変換が可能である場合があり、なにもしないよりもその方がよいからである。ローマ数字でπを書くことが可能ではなくても、“III"と書くと、それに近いものが得られる。丸め（rounding）オプションによっては、表記を所望の結果に対応づけることが可能である。

TConversionResult200クラスは、図８に示すように、変換がどれだけ正しく行われたかの広い情報をプログラマに提供するために存在するものである。このクラスはプログラマに役立つと思われる変換に関する他の情報も提供する。

TConversionResult200から得られる情報は以下で説明する。

```
virtual ParseConfidence GetConfidence() const;
```

ParseConfidence800は0.0と1.0の間の浮動小数点数であり、これは変換の全体的評価（overall rating）を示している。どちらの数形式が特定の変換にとって良好な形式であるかを示すために代表的に使用される。

```
//     long double aNum、および数形式format1，format2が与えられているとする
TConversionResult result1, result2;
TText text1, text2, theText;
format1.NumberToText (aNum, text1, result1);
format2.NumberToText(aNum, text2, result2);
//     変換の結果がよい方を使用する
if (result1.GetConfidence() > result2.GetConfidence())
        theText = text1;
else theText = text2;
virtual unsigned long GetLengthUsed() const;
```

GetLengthUsed802は、ある与えられたテキスト・ストリングの中の文字がテキストから数

への変換でいくつ使用されたかを示している。これをテキスト・パーサ（text parser－テキスト構文解析プログラム）で使用すると、構文解析の次のステージがどこで行われるかを示すことができる。

virtual Boolean GetSeparatorError() const;

GetSeparatorError804は、テキストから２進数への変換におけるディジット・セパレータ（digit separtor－桁分離符号）に含まれるエラーを示すために使用される。例えば、数字“1,23.45"は、“123.45"を意味するつもりであることは明らかである。この例では、変換が行われると、GetSeparatorError（）からTRUEが返される。

virtual Boolean GetIncompleteSign() const;

GetIncompleteSign806は、テキストから２進数への変換において正符号または負符号に含まれるエラーを示すために使用される。例えば、負の数字をかっこで囲むようにセットしていたとすれば、“（123.45"は（123.45）の誤りであることは明らかである。この例では、変換が行われると、GetIncompleteSign（）からTRUEが返される。

virtual Boolean GetValueOrderError() const;

GetValueOrderError808は、桁順序（digit order）に影響されやすい数形式、具体的にはローマ数字、で桁の順序に含まれるエラーを示すために使用される。“IILCMM"というストリングはローマ数字を書こうとしたように見えるが、桁の順序が正しくない。ローマ数字フォーマッタはこのストリングを２進数に変換しようとするので、GetValueOrderError（）からはTRUEが返されることになる。

virtual Boolean GetCanNormalize() const;

GetCanNormalize810は、テキスト・ストリングを２進数に変換し、再びテキスト・ストリングに戻すとき、次のような場合にTRUEを返す。

（ａ）丸めが行われない。

（ｂ）正規化されたストリングの最後に文字を付加すると有効な数が得られる。

例えば、ディジット・セパレータにカンマを、小数点セパレータ（decimal separators）にピリオドを使用する標準の米国数値フォーマッタを採用し、負符号がかっこで、小数点以下の桁数が最大でも４桁であるものとする。

“123.45"は正規化が可能である－正規化すると“123.45"になる。

“1,23.45"は正規化が可能である－正規化すると、“123.45"になる。

“12345.67"は正規化が可能である－正規化すると、“12,345.67"になる。

“123.456789"は正規化が可能ではない－正規化すると、“123.4567"になり、“89"は失われることになる。

“（123.45"は正規化が可能ではない－正規化すると、“（123.45）”になり、“6"を付加すると、これは無効なストリング“（123.45）6"になってしまう。

正しい有理数フォーマッタを使用すると、次のような結果が得られる。

“3"は正規化が可能である－正規化すると、“3"になる。

“3 1"は正規化が可能ではない－有理数フォーマッタは“1"を分母とみなすため全体が“4"に変換されることになる。これは丸めとカウントされる。

“3 1/7"は正規化が可能である－有理数フォーマッタはこれを“3 1/7"に正規化する。

virtual unsigned long GetIntegerBoundary() const;

多くの数値フォーマッタは非整数値をフォーマットするために使用されている。小数点タブを行うには、数値の整数部と小数部間の区切りをどこに置くかの情報が必要になる。GetIntegerBoundary（）712は、その情報を正確に与える数からテキストへの変換によって生成されたテキスト・ストリングまでのインデックスを返す。

virtual unsigned long GetDigitSequenceEnd() const;

多くの数形式では、数字とその前後のテキストを含む。例えば、“（$1,000）”におけるように、かっこを使用して示した負数を見ることがあっても、これは異常なことではない。従って、重要なことは、フォーマッティングされた数の数値部分がどこで終わってい

るかを知ることであり（特に、正規化された場合）、そうすれば、正しい個所からテキストの編集を行うことが可能になる。この情報はGetDigitSequenceRange（TTextRange＆）714によって返される。

```
virtual Boolean GetOutOfBoundsError() const;
```

すべての数値フォーマッタは、それが表現できる値の範囲が関連づけられている。また、変換するために与えられた数が範囲外にあるとき使用される、関連の数値フォーマッタももっている。数値フォーマッタがその範囲外フォーマッタを使用して数をテキストに変換する必要が起こったときは、GetOutOfBoundsError（）716からTRUEが返される。例えば、以下に示すコード断片では、範囲外の数値変換を示すために異なるカラーが使用されている。

```
TRGBColor theInBoundsColor(0.2,0,0);
TRGBColor theOutOfBoundsColor (0.25, 0.25, 0.25);
TConversionResult                    theResult;

theFormat->NumberToText ( aNum, numText, theResult);
if (theResult.GetOutOfBoundsError())
        DrawNumber (numText, theOutOfBoundsColor);
else DrawNumber (numText, theInBoundsColor);
```

TNumerals206

TNumerals206は、個別の数字文字－値（numeral char－value）のマッピングを行うために数値フォーマッタによって使用されるクラスである。つまり、kDigitSevenの値は米国10進形式で７になり、kRomanNumeralTenはローマ数字を使用して値10になる。

数値フォーマッティングを行うとき使用される、TNumerals206内のメイン・メソッドは次のとおりである。

```
virtual Boolean NumeralToValue (UniChar ch, long& value) = 0;

virtual Boolean ValueToNumeral (long value, UniChar& ch) = 0;
```

特定のTNumberFormat218によって使用されるTNumerals206は、オブジェクトのそのインスタンスの場合に限って数字がなんであるかを定義している。１に“q"の英字を、７に“z"の英字を使用するTNumberFormat218のインスタンスを得ることはまったく可能である。開発者が独自の数形式オブジェクトを書くときは、そのオブジェクトのTNumerals206から派生された情報を使用すべきである。これの例外は、TUniversalNumberFormatクラスなどの数値フォーマッタの場合であり、これは有効な数字桁ならば、どの数字桁でも扱えるように設計されている。

TNumerals206のサブクラスは現在４つある。Unicodeによって定義されたデフォルト数値特性を使用するTUnicodeNumerals216と、値がシーケンシャルになっている場合の数値集合に対するTSequentialNumerals208と、値がシーケンシャルになっているほかに、値を表す文字が連続している場合のTContiguousNumerals210と、数字－値ペアの任意の集合のTHybridNumerals214である。数字－値ペアはUniCharとその数値を関連づけたものであり（例えば、ローマ文字の“L"＝50、ギリシヤ文字の“β”＝２）、TNumeralPairオブジェクトによって表される。

TUnicodeNumerals216

TUnicodeNumerals216クラスは、テキストからバイナリへの変換用に定義されたディジット値をどれでも受け付ける。従って、ラテン語－アラビア語のディジット１、アラビア語－インド語のディジット１、デーヴァナガリー語（Devanagari）のディジット１、その他はすべて値１をもっている。

バイナリからテキストへの変換を取り扱うには、TUnicodeNumerals216オブジェクトはど

のセットのディジットを使用するかを知っている必要がある。これを行うには、次のメソッドが使用される。

```
void SetScript (TUnicode::ScriptName);

TUnicode::ScriptName GetScript (void);
```

スクリプトは、TUnicodeNumerals216オブジェクトの作成時に指定することもできる。（デフォルトはTUnicode::kRomanである。）ある値がマッピングされるUniCharを提供するとき、TUnicodeNumerals216クラスはスクリプト情報を使用して、どのUniCharが適当であるかを決定する。従って、スクリプトがTUnicode::kRoman、TUnicode::kGreek、またはTUnicode::kCyrillicに等しければ、１を表すTUnicode::kDigitOneが返される。スクリプトがTUnicode::kArabicであれば、TUnicode::kArabic_indicDigitOneが返され、以下同様である。

THybdirNumerals214

THybridNumerals214オブジェクトは数字ペアを２セット含んでいる。一方のペア・セットはテキストを数に変換するときにだけ使用される。これはスキャニング・ペアのセットである。２つの異なるUniCharはこのリストの中で同じ数値と関連づけることができる（例えば、ローマ数字のときは、“ i"または“ I"のどちらも、１に変換される）。数字ペアは、メソッドAddScanningPair（）を使用してスキャニング・ペアのセットに追加される。フォーマッティング・ペアのセットもあるので、各数値を最大でも１つのUniCharと関連づけることができる。フォーマッティング・ペアのコレクション（集合）の中のすべてはスキャニング・ペアのセットの中にあるものと自動的にみなされる。数字ペアは、メソッドAddFormattingPair（）を使用してフォーマッティング・ペアのセットに追加される。従って、ローマ数字は“ i"または“ I"のどちらも、テキストを数に変換するとき１に等しいものとして使用するように定義することができるが、数をテキストに変換するときは“ I"だけが使用される。

TNumberFormat218のサブクラス

TPositionalNumberFormat220

TPositionalNumberFormat220は値ベースの数体系（value－based system）で書かれた整数用に使用され、そこでは、ディジットの値は部分的にはそのディジット自体と、そのディジットが数字のどこに置かれているかによって決まるようになっている。TPositionalNumber形式は、例えば、使用する丸めメソッド（利用できる丸めメソッドはCanonicalNumber.hで定義される）、表示するディジット数、その他を決定するための、追加のセッタ（setter）とゲッタ（getter）メソッドをいくつか定義している。その中で最も重要なものを示すと次のとおりである。

```
// ディジット・グループ・セパレータは数字のディジットのグループ
// 間で使用されるテキストである
// 米国ではディジット・グループ・セパレータに"," が使用されるのが通常
// ヨーロッパでは"." が使用されるのは通常。スペースもよく使用される

virtual void GetDigitGroupSeparator (TText&) const;
virtual void SetDigitGroupSeparator (const TText&);

// グループ内の桁（ディジット）数：
// 西洋では３桁が共通(1,000,000)
// 東洋では４桁が使用される(1,0000,0000)

virtual int GetSeparatorSpacing () const;
virtual void SetSeparatorSpacing (int);

// 整数の区切りを表示するか？
// 1000000(false)にするか、1,000,000(true) にするか

virtual Boolean GetIntegerSeparatorOn () const;
virtual void SetIntegerSeparatorOn (Boolean);

// 精度はフォーマットした数(NumberToText)を
// どのように丸めるかを指定する
// 倍数は増分値であり、丸めは変換でどのように
// 丸めるかを指定する

virtual void GetPrecision (long double& multiple,
 ERoundRule& rounding) const;
virtual void SetPrecision (long double multiple,
 ERoundRule rounding);

// minIntは数をテキストとしてフォーマット
// するとき表示する最小桁数である。
// ゼロ埋込み(zero-padding)とも呼ばれる

virtual int GetMinIntegerDigits () const;
virtual void SetMinIntegerDigits (int);
```

TFloatingPointFormat222

TFloatingPointFormats222は位取りをベースにした数体系（position－based numeral systems）で使用され、非整数を含むようにアルゴリズムを拡張している。TFloatingPointFormat222は、例えば、πを3,14159265358979323846264338327950で表すことができる。TFloatingPointFormat222は、表示する小数点の個数などのパラメータを指定するための特殊なメソッドを含んでいる。メソッドには、次のものがある。

```
// 小数点文字は少数表現の「左」側と「右」側の
// 間に置かれる文字である。米国では"." が使用され、
// ヨーロッパでは"," が使用される
virtual void GetDecimalSeparator (TText&) const;
virtual void SetDecimalSeparator (const TText&);

// 整数の場合でも小数点を表示するか？
virtual Boolean GetDecimalWithInteger () const;
virtual void SetDecimalWithInteger (Boolean);

// 分数部(fractional part) にセパレータ文字を使用するか？
virtual Boolean GetUseSeparatorForFraction () const;
virtual void SetUseSeparatorForFraction (Boolean);
// 指数部にセパレータ文字を使用するか？
virtual Boolean GetUseSeparatorForExponent () const;
virtual void SetUseSeparatorForExponent (Boolean);

virtual int GetMinFractionDigits () const;
virtual void SetMinFractionDigits (int);
virtual int GetMaxFractionDigits () const;
virtual void SetMaxFractionDigits (int);
```

TUniversalNumberFormat

これはTFloatingPointFormat222の特殊なサブクラスであり、汎用範囲（および他のすべての数形式ではデフォルト範囲外数形式として使用されるクラス）をもっている。TUniversalNumberFormatは、種々タイプの無限数のNaN（ゼロによる除算結果など）を扱うことができる。その主目的はデフォルト範囲外形式を提供し、いずれかの２進数を数字に変換することが要求されたとき、どの形式からも、なにかが返されることを保証することである。

TRationalNumberFormat

これはTPositionalNumberFormat220から派生されたもので、非整数値を２整数間の比率として書く能力を追加したものである。このクラスには、特殊なenum（列挙型）が２つあり、１つは分数部の「作法（propriety）」を指定するためのものであり、もう１つは分数部が最初に分子を書いているか（これは西欧で普通に行われている）、最初に分母を書いているか（これは中国で伝統的に行われている）を決定するためのものである。

```
enum EFractionPropriety { kProperFraction, kImproperFraction };

enum EFractionDirection { kNumeratorFirst, kDenominatorFirst };
```

正しい分数部とは、その分子が分母より小さくなっているものであり（例えば、“3 1/7）、正しくない、または洗練されていない分数部とは、その分子が分母より小さくなることもあれば、そうでないこともあるようなものである（例えば、22/7）。有理数の整数部と分数部の間に使用する文も指定され（通常は、TUnicode::kSpace）、数字をスーパスクリプト（上付き）にするかどうか、分母をサブスクリプト（下付き）にするかどうかも指定される。この文字を使用すると、例えば、“355/113"と“355/113"との間の切替えが容易になる。

しかし、各有理数形式では、整数をフォーマッティングすることに関連する情報が必要である。これは次のメソッドを使用して指定する。

```
virtual TNumberFormat*      GetIntegerFormat() const;
virtual void                AdoptIntegerFormat (TNumberFormat *);
```

整数形式のデフォルトは標準TPositionalNumberFormat220であるが、これはオーバライドすることができる。これを使用すると、有理数の分子と分母の表現を完全に制御することができるので、“III^{I}/VII"は“22/7"と書くのと同じように容易化される。
従来の文字セット標準との互換性をもたせるために、“1/2"や“1/3"のような分数文字のセットも定義されている。kSlash（つまり、“/"）やkFractionSlash（“/'）のように、分数の分子と分母を区別するために使用できる異なる文字もある。
TRationalNumberFormat224オブジェクトは、これらの文字をテキストから数への変換のときだけ使用することをサポートしている。分数文字を表す絵文字（glyphs）をもつフォントはほとんどないので、数からテキストへの変換では、標準アラビア数字文字とスタイル化テキスト（styled text）を使用して分数を表すことになる。

```
// TRationalNumberFormat を構築する
TRationalNumberFormat* rationalFormat;
// デフォルトのベースは10、すなわち、kDigitZero..kDigitNineであり、
// 数値オブジェクトを渡さないこと

// 変換の例
TConversionResult parseResult;
TText testText("42 2/16");
long double num = rationalFormat->TextToLongDouble(testText,
parseResult);
      // num は42.125になる
rationalFormat->NumberToText(num, testText, parseResult);

// testTextは" 42 1/8 "になる
```

TOutlineNumberFormat
このメソッドは、TNumberFormat218から派生されたもので、シーケンスがa,b,c,..,z,aa,bb,zz,aaa,...のアウトライン・スタイルの番号付け方式をインプリメントしている。

```
// TOutlineNumberFormatを構築する
TContiguousNumerals outlineNumerals(TUnicode::kLatinSmallLetterA,
1,26);
// 数字は`a`..`z`で値は1から始まっている
TOutlineNumberFormat* outlineFormat;
outlineFormat = new TOutlineNumberFormat(outlineNumerals, 26); //
数字、ベース
// TOutlineNumberFormatを使用して変換する
TConversionResult parseResult;
TText testText("cc");
long num = outlineFormat->TextToLong(testText, parseResult);
// num は29になる
outlineFormat->NumberToText(num, testText, parseResult);    // 再び
戻る
```

注意： TOutlineNumberFormat は図２にエレメント２２８で示されている。

TAadditiveNumberFormat
数が位取りになっている数体系は最も親しまれており、そこでは、各シンボル（symbol）の値はその位置によって決まっている。よく使用される他の数体系は現存している加法（additive）であり、そこでは、各シンボルは固有値（inherent value）をもち、数の合計値は各シンボルの値を加算することによって得られ、シンボルが現れる順序は無視しているのが普通である。米国で最も普及している事例はローマ数字である。
TAdditiveNumberFormat236はTNumberFormat218から派生されたもので、単純な加法数値表現を取り扱っており、そこでは、異なる数字の値は互いに加算されるだけで数を形成している。つまり、数47は古典ギリシア語では“μζ”と表される。ただし、“μ”は40を表すために使用され、“ζ”は７を表している。このクラスはそれ自体インテリジェントではない。これは“μμζ”、“μκζ”または“κζκ”を、正しい“μζ”と同じように即座に47として受け取る。これは数をテキストに変換するのにあまり精巧化されていない。もっと精巧化された変換アルゴリズムをもち、あるいはもっと精巧化されたエラー検査を行う数体系はローマ数字やヘブライ数字で行われているように、サブクラスで表す必要がある。
TRomanNumberFormat230
このメソッドはTAdditiveNumberFormat236から派生されたもので、ローマ数字を取り扱う。ローマ数字に対するサポートはページ番号やアウトラインの番号といったように、限られた場合を主に目的としている。ローマ数字のケース（case）の制御は次のメソッドによって容易化されている。

```
enum ERomanNumeralCase { kUpperCase, kLowerCase };
ERomanNumeralCase   GetRomanNumeralCase()      const;
void                SetRomanNumeralCase (const ERomanNumeralCase);
```

ここで注意すべきことは、どのローマ数字形式もスキャニングを行うとき、大文字と小文字のどちらも自由に受け付けることである。“MCMXCII"、“mcmxcii"および“mCmXcIi"はいずれも1992になる。なお、２進数を数字に変換するとき大文字を使用するか、小文字を使用するかは明示的に制御する。ローマ数字の記数法は数種類があるが、これらは「減法

原理（subtractive principle）」の使い方が相互に異なっている。これによれば、４は“IIII"（１＋１＋１＋１）ではなく、“IV"（５－１）と表されている。

TRomanFormat230は、この振舞いを制御するように定義された特殊なenumをもっている。

```
enum ERomanNumeralType { kShort,
                         kNormalLong4Long8,
                         kNormalLong4Short8,
                         kNormalShort4Long8,
                         kNormalShort4Short8,
                         kLong };
```

ショート（short）記数法は減法原理を可能な限り多用して、ローマ数字をできるかぎり短縮化している。ロング（long）記数法は減法原理をまったく使用していない（これは純粋論者（purists）が支持していた最初の記数法である）。その他の記数法は４と８を表すために減法を使用するかどうかで異なっている。

数	4	6	8	9	1999
kShort	IV	VI	IIX	IX	IM
kNormalLong4Long8	IIII	VI	VIII	IX	MCMXCIX
kNormalLong4Short8	IIII	VI	IIX	IX	MCMXCIX
kNormalShort4Long8	IV	VI	VIII	IX	MCMXCIX
kNormalShort4Short8	IV	VI	IIX	IX	MCMXCIX
kLong	IIII	VI	VIII	VIIII	MDCCCCLXXXX VIIII

また、5,000より大きい数の扱い方も様々である。大きい数を書くとき古代の人達の間には統一性がほとんどなかったが（大きい数のカウントに苦労していたためである）、現代ではその必要はほとんどない（すぐれた数体系が存在するためである）。旧来の文字セット標準との互換性をもたせるために、好適実施例では、ローマ数字文字のセットが対応するラテン文字と共に定義されている。文字“I"とは区別されるローマ数字“I"、ローマ数字“ii"その他が、大文字と小文字の両方に存在する。

TRomanFormat230オブジェクトは、テキストから数への変換のときだけこれらの文字を使用することをサポートしている。ローマ数字文字の絵文字が得られるHォントはほとんどないので、数からテキストへの変換では、ローマ数字文字はUnicodeでは使用されず、この中には、

5,000(ↁ)、10,000(ↂ)および 1,000(ↀ)

の代替形式のように、まれにしか使用されず、アルファベットに類似のものがないローマ数字が含まれる。しかし、これらの文字はテキストから数への変換では使用することができる。

最後に検討するのは、１つの変種の記数法である。あるコンテキストでは、ローマ数字の最終文字に“I"の文字ではなく“J"の文字が使用されるのが普通であるので、６は“VI"または“vi"ではなく“VJ"または“vj"と書かれている。この振舞いも制御が可能である（デフォルトでは、“I"で終わらないようになっている）。

```
Boolean     GetTerminatesWithJ() const;

void  SetTerminatesWithJ(const Boolean);
```

任意のTHybridNumerals214オブジェクトの使用はTRomanFormat230ではサポートされない。TRomanFormat230の構築子（constructor）では、THybridNumerals214オブジェクトまたはベースを指定する必要はない。この振舞いはTRomanFormat230をサブクラス化することによってオーバライドされる。

```
// TRomanNumberFormatを構築する
TRomanNumberFormat* romanFormat = new TRomanNumberFormat
(TRomanNumberFormat::kUpperCase);

// 変換の例
TConversionResult parseResult;
long num = 1992;
TText testText;
romanFormat->NumberToText(num, testText, parseResult);
      // testTextは"MCMXCII" になる
romanFormat-
>SetRomanNumeralCase(TRomanNumberFormat::kLowerCase);
romanFormat->NumberToText(num, testText, parseResult);
      // testTextは"mcmxcii" になる
num = romanFormat->TextToLong(testText, parseResult);      // num は
1992になる
```

THebrewNumberFormat232

ヘブライ数字は、古代ギリシア数字と同じようにヘブライ・アルファベットを使用して書かれているが、そこでは、最初の９文字に１から９までの値が割り当てられ、次の９文字に10から90までの値が割り当てられ、以下同様である。数字の値はその数字を構成する文字の合計値である。これはTAadditiveNumberFormat236を使用して表されるのが普通であるが、軽々しく口にすべきでない神の名前の一部を数字として使用するのを避けるには、１４は１０＋４（יה）とではなく９＋５（טו）と書き、１５は９＋６（טה）と書かなければならない。この結果、ヘブライ数字には特殊なオブジェクトを使用する必要がある。ヘブライ語は大文字と小文字を区別しないため、数をヘブライ数字に変換するとき使用できる代替文字はないので、THebrewNumberFormat232では、これ以上に複雑化することはない。THebrewNumberFormat232のコンストラクタでは、THybridNumerals214オブジェクトまたはベースを指定する必要はない。

ヘブライ数体系の変種として、ユダヤ中世の神秘教義書－カバラ（Cabala）で使用されているものがある。THebrewNumberFormat232は、カバラ数字と標準ヘブライ数字とを区別するときのインタフェースを提供する。

THybridNumberFormat234

これはTAdditiveNumberFormat236から派生されたもので、しきい値をもつことができる能力を追加していて、そこでは、数字は加法ではなく乗法で使用されている。つまり、中国語の数字を使用している次の例では、数1729は［１］［1000］［７］［100］［２］［10］［９］（一千七百二十九）と表され、これは１＊1000＋７＊100＋２＊10＋９を意味する。

```
//THybridNumberFormat を構築する
THybridNumerals chineseNumerals;
chineseNumerals.AddFormattingPair(TUnicode::kHanNumeralOne, 1);
        // パラメータはUniChar, long である
        // 数字を追加するたびに繰り返す
chineseNumerals.AddFormattingPair(TUnicode::kHanNumeralOne,
1000);
THybridNumberFormat* chineseFormat;
chineseFormat= new THybridNumberFormat(chineseNumerals, 10, 10);
        // パラメータは数字、
        // ベース、しきい値である
```

THanNumberFormat226

日本、朝鮮、ベトナム、そして特に中国などの東アジアの古代文明では、高度に発達したハイブリッド数体系があり、現在でも共通して使用されている。これらはTHanNumberFormat226によってサポートされていて、「漢（Han）」はそれぞれの文明を参照するために東アジアで使用されている共通名である。

ローマ数字の場合と同じように、漢数字を書くときに使用される記数法には種々のものがある。しかし、古代中国人は、ローマ人と異なり、熟練した数学者であったため、また古代帝国は大きく、人口が多く、安定し、長期間に存続していたので、漢数字の変種では、数が非常に大きく、数千万以上のオーダになっていた。また、漢数字の変種として、西欧の数字に基づき、完全に位取り記数法になっているものもあり、そこでは、21から39までの数は特殊な短縮形が使用されている（カレンダ数字）。これらは、次のようにTHanNumberFormat226クラス内のenumによってサポートされている。

```
enum EHanNumberType     {     kHanCalendar,
                              kHanStandard,
                              kHanXiaDeng,
                              kHanZhongDeng,
                              kHanShangDeng,
                              kHanWestern };
```

標準的記数法は最も頻繁に使用されているものであり、99,999までの数を書くための統一された方法を提供する。もっと大きな数を書くための標準的記数法には、xiadeng（ローレベル）記数法、zhongdeng（中間レベル）記数法、shangdeng（ハイレベル）記数法の３つがある。これらの３つの記数法はすべて、数字を表すために10個の追加文字を使用している。

	TUnicode::	Xiadeng 値	Zhongdeng 値	Shangdeng 値
億	kHanNumeralYi	10^5	10^8	10^8
兆	kHanNumeralZhao	10^6	10^{12}	10^{16}
京	kHanNumeralJing	10^7	10^{16}	10^{32}
垓	kHanNumeralGai	10^8	10^{20}	10^{64}*
補	kHanNumeralBu	10^9	10^{24}	10^{128}*
秭	kHanNumeralZi·	10^9	10^{24}	10^{128}*
壤	kHanNumeralRang	10^{10}	10^{28}	10^{256}*
溝	kHanNumeralGou	10^{11}	10^{32}	10^{512}*
澗	kHanNumeralJian	10^{12}	10^{36}	10^{1024}*
正	kHanNumeralZheng	10^{13}	10^{40}	10^{2048}*
載	kHanNumeralZai	10^{14}	10^{44}	10^{4096}*

漢計数法（Han enumeration system）に使い慣れている大部分の人は、３つの記数法の違いを上手に使いこなせるはずである。これらの計数法がどのような働きをするかは、Georges Ifrah著“From one to zero"「１からゼロまで」（New York:Viking Penguin,Inc.,▲C▼1985）、pp.365以下に詳しく説明されている。

漢数字の使用における決定的妙案（final wrinkle）は、中華人民共和国において簡易文字が公表されたことである。大陸の中国では、伝統的形式の大部分は簡易文字によって置き換えられているが、伝統的形式は日本、朝鮮、香港、および台湾で引き継がれて普及化されており、THanNumberFormat226クラスでは、デフォルトの漢字として使用されている。しかし、通常通りに、どちらの形式もフォーマッタによって正しく読み取られるので、数をテキストとしてフォーマッティングするとき、どちらを使用するかを指定するためのインタフェースが用意されている。

```
enum EHanSimplification { kHanSimplified, kHanTraditional };

EHanSimplification      GetHanSimplification()  const;

void  SetHanSimplification (const EHanSimplification);
```

新しいTNumberFormat218の書き方

独自のTNumberFormat218オブジェクトを書く開発者は、テキストをTCanonicalNumberFormatter202形式へ変換し、またテキストをそれから変換するメソッドを書く必要がある。これらの変換は、下にリストした２つの純仮想メソッドを通して行われる。

```
virtual void TextToCanonicalNumber (const TText&, const TRange&,
        TCanonicalNumber&, TConversionResult&) = 0;

virtual void CanonicalNumberToText (const TCanonicalNumber&, TText&,
        TConversionResult&) = 0;
```

独自のTNumberFormat218オブジェクトを書く開発者は、NumberToTextメソッドとTextToNumberメソッドをオーバライドしないで済むことを知るはずであり、というのは、これらのメソッドはCanonicalNumberToTextとTextToCanonicalNumberをコールするだけで、規準数形式（canonical number format）を作成し、適当なTCanonicalNumberFormatter202メソッドとTCanonicalFormatter204メソッドを使用して、２進と規準数形式間の変換を最も効率よく行うからである。

なお、TNumberFormat218によって使用されるTCanonicalNumberFormatter202は持続的である。本発明によって定義されている数値フォーマッティング・クラスはTCanonicalNumber

Formatter202内のフィールドにはデフォルト値を用意しており、該当する場合には、これらのフィールドのセッタとゲッタを含んでいる。このことは、独自のTNumberFormat218オブジェクトを書く開発者はTCanonicalNumberFormatter202とTCanonicalNumber204をセットアップするときの詳細の多くについて気にする必要がなく、開発者の要求に関係する詳細だけに集中できる。具体的には、精度や丸めといった、TCanonicalNumberFormatter202フィールドの値をセットし、返すメソッドオーバライドしないで済むことになる。これらはTPositionalNumberFormat220、TFloatingPointFormat222、およびTRationalNumberFormat224オブジェクト内のセッタとゲッタによって処理される。

数字フォーマットの使用

上述したインタフェースは、下に示す例においてTNumberFormat218* 形式を使用して、数とテキスト間を変換するために使用されるものである。

```
TText testText;
long double num = 42;
TConversionResult parseResult;        // parse infoを返すオブジェクト

format->NumberToText(num, testText, parseResult);
// testTextはformatによって定義された数のテキスト表現
を有する
// 次にテキストから数へ逆変換する
num = format->TextToLongDouble(testText, parseResult);        // num は
42になる
// 変換がどれかで正確であったかをTConversionResult
// オブジェクトに照会する
ParseConfidence confidence = parseResult.GetConfidence();
// 信頼度(confidence)は浮動小数点数であり、1.0 が満点である
unsigned long lengthUsed = parseResult.GetLengthUsed();
// lengthUsedはUnicharsが入力テキストからいくつ使用されたかを返す
//次にTText オブジェクトの一部だけ (最初のUniChar)を変換する
num = format->TextToLongDouble(testText, parseResult, TRange(0,1));
// num はテキスト表現がどうなっているかに応じて
// 大部分がたぶん異なっている
// 数42は数形式`format`になっている
```

数フォーマットの操作

以下の例では、TFloatingPointNumberFormatオブジェクトを使用して、数形式のそのクラス用に変更できるいくつかの固有パラメータを示している。

```
// 最初はデフォルトの振舞い
long double num = 42125.67;
TText testText;
TConversionResult parseResult;
floatFormat->NumberToText(num, testText, parseResult);
      // testTextは"42,125.67" になる
// 最高でも１つの小数点を含むようにformatを変更する
floatFormat->SetMaxFractionDigits(1);
floatFormat->NumberToText(num, testText, parseResult);
      // testTextは"42,125.6"になる
// 末尾にゼロがある場合でも、常に少なくとも３つの10進数(decimals)
// を含むようにformatを変更する
floatFormat->SetMinFractionDigits(3);
floatFormat->SetMaxFractionDigits(6);
floatFormat->NumberToText(num, testText, parseResult);
      // testTextは"42,125.670"になる
// 1000の位のセパレータを使用しないように変更する
floatFormat->SetIntegerSeparatorOn(FALSE);
// TPositionalNumberFormat から継承
floatFormat->NumberToText(num, testText, parseResult);
      // testTextは"42125.670" になる
```

数フォーマットの作り方

ユーザ選好のメカニズムがいったん用意されれば、ユーザにとって望ましい形式を得ることができるので、TNumberFormat218のインタフェースを多態的に使用するだけで数の変換を行うことができる。異種言語領域用に事前定義された数形式も用意されているので、ユーザ自身が数形式を作成する必要が起こるのはまれである。以下は、現在用意されている具象化数値フォーマッティング・クラスであり、これらを作成する方法のコード・サンプルも示されている。なお、TNumberFormat218オブジェクトには、そのコンストラクタでTNumerals206とベースが必要になるものと、そうでないものとがある。上述した補助的オブジェクトのセクションを参照。

【図１】

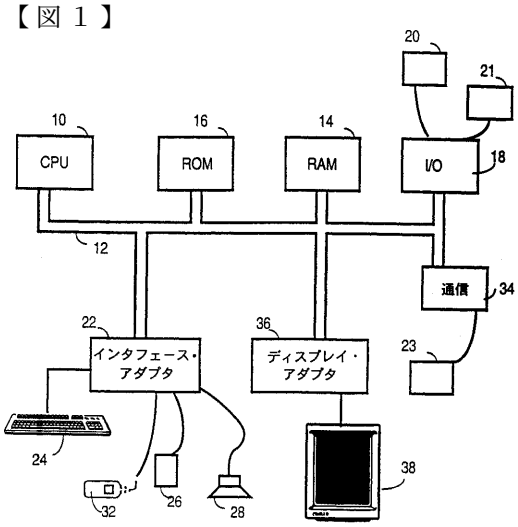


Figure 1

【図２】

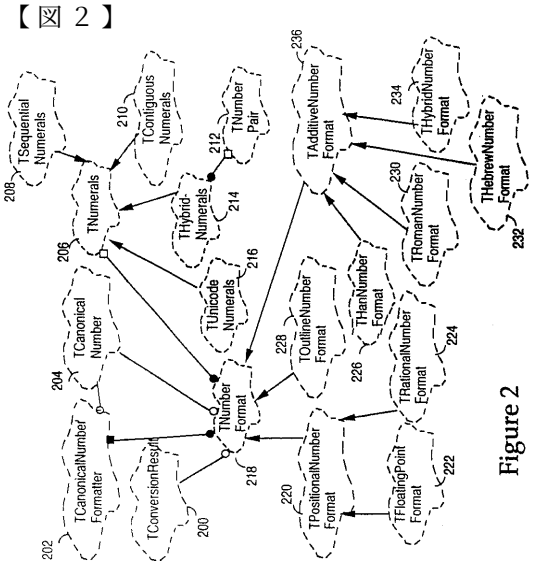


Figure 2

【図３】

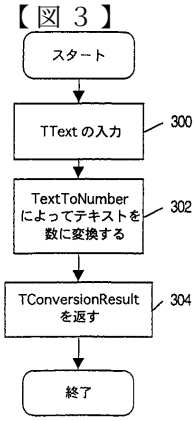


Figure 3

【図４】

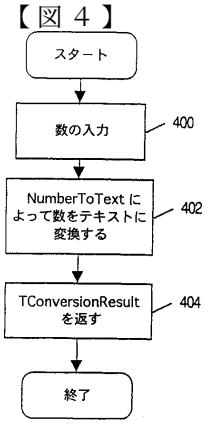


Figure 4

【図５】

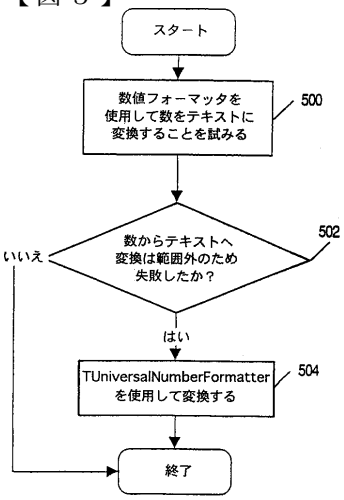


FIGURE 5

【図７】

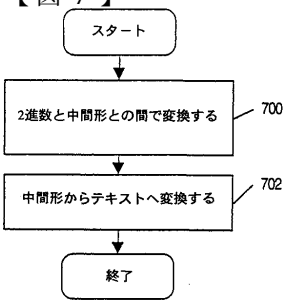


Figure 7

【図６】

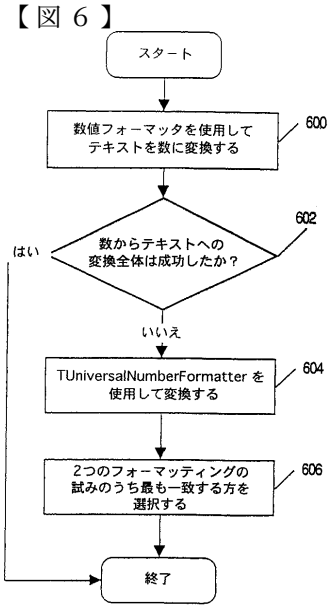


Figure 6

【図８】

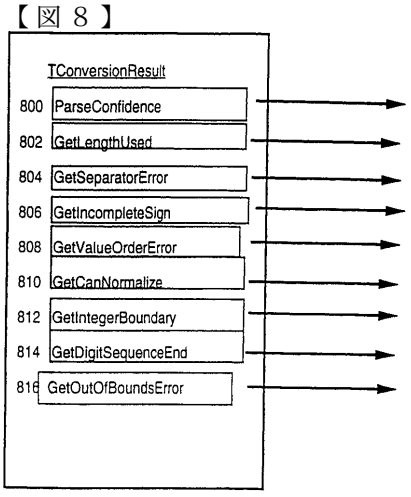


Figure 8

フロントページの続き

(72)発明者　ジェンキンス，ジョン，エイチ.
アメリカ合衆国　９５１２９　カリフォルニア州　サンノゼ　カメオ　ドライブ　１５０３
(72)発明者　プーネン，サンジャイ，ジェイ.
アメリカ合衆国　９４０８７　カリフォルニア州　サニーヴェイル　ペイエット　アヴェニュ　１０４９

審査官　田中　友章

(56)参考文献　特開平０５－１２７９６１（ＪＰ，Ａ）
特開平０５－００６３９５（ＪＰ，Ａ）
特開平０３－０８３１６１（ＪＰ，Ａ）
国際公開第９３／２１５９１（WO，Ａ１）

(58)調査した分野(Int.Cl.[7]，ＤＢ名)
G06F　5/00
G06F 17/21
G06F 17/22