リアルタイムシステム（PHS）への オブジェクト指向設計適用とZIPC++の利用

京セラ株式会社
国内パーソナル通信事業部 国内PS技術部 第3技術課責任者
松本　宏規

１．はじめに

弊社では1995年のPHSサービス開始時点より、PHS基地局、PHS端末を提供してきております。

皆様もご存知のとおり、この10年間で携帯電話、PHSは大きく変化しました。

最初期のPHS端末のROM容量は無線プロトコルとマン・マシンインターフェースを合計しても64KBしかありませんでしたが、最近のフルブラウザ搭載端末では20MBを超えており、単純計算で300倍以上の規模となってしまいました。

そのような状況ですから開発効率の向上は経済的にも、開発者の人間的な生活を維持するためにも必要であったわけです。

今回、弊社で2001年から取り組んできたPHS端末開発へのオブジェクト指向設計、ZIPC++の利用による開発効率向上の事例を報告いたしますので、皆様のご参考になれば幸いです。

２．リアルタイムシステム～PHSの場合

まずPHS端末がどのようなリアルタイムシステムであるかをご紹介します。

PHSは無線通信方式に時分割多元接続－時分割多重複信（TDMA-TDD）方式を用いています。この時分割制御に対応できるリアルタイム性を満たす必要があります。

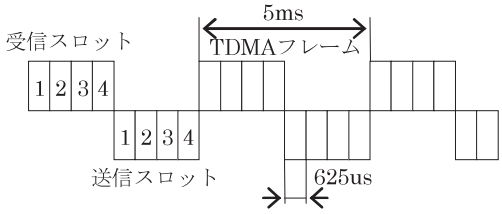

図１　PHSのTDMA-TDDタイミング

図１のように5msのTDMAフレームを単位として、そのフレームを625usの８スロットに分割しています。

送信と受信は同じ番号のスロットを用いて行なわれるので、2.5ms間隔で送信と受信が交互に行なわれることになります。

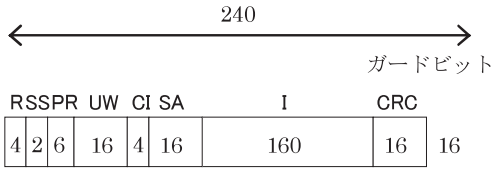

図２　通信スロットの構成

図２のように、１つのスロットは240ビットで構成され、ユーザーデータはI部分の160ビットによって転送されます。

１フレームが5msですから、１秒間に転送されるのは200フレームとなり、１スロットのユーザーデータが160ビットですから、200フレーム×160ビット＝32000ビットとなって、32Kbpsの通信速度を提供することができます。

音声、32Kbpsデータ通信は、送受信に１スロットずつ用います。64Kbpsは２スロット、128Kbpsは４スロットを同時に用いることで実現されます。

スロットを受信すると、図３のMODEMによって復調され、データが受信バッファに格納されます。

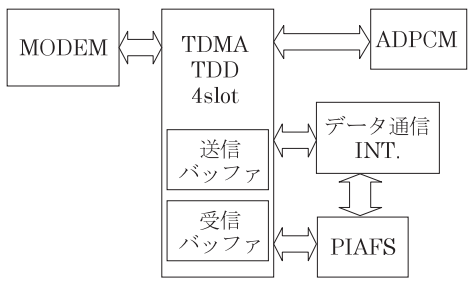

図３　ベースバンド部のブロック図

音声通信時は、ハードウェアが受信データをADPCMに転送して音声出力が行なわれます。

32K/64K PIAFS通信時は、ハードウェアが受信データをPIAFSに転送してPIAFSのフレーム同期処理が行われます。PIAFSのフレームは４スロットのデータによって構成されるため、32K PIAFS時は20ms周期、64K PIAFS時は10ms周期で割り込み要求が発生し、CPUでの処理が必要となります。

ハードウェアによる支援がある音声、PIAFS通信と異なり、パケット通信は全てソフトウェア処理によって実現されます。

パケット通信時は、スロット受信完了割り込みが発生した時点で、受信バッファに格納されたデータをCPU処理によって取り出してメモリへ転送する必要があります。

従って１フレーム内のスロットを全て利用する128Kbps通信時には、625us周期で割り込み処理が発生することになります。

これらの割り込み処理を行なうCPUですが、現在はベースバンド部と１チップになったASICに搭載されているARM7TDMIコアが処理を行なっています。

このARM7TDMIコアを19.2MHzで動作させることによって、32Kbpsの音声/PIAFS/パケット通信、64K PIAFS通信を実現しています。

128Kbps通信を実現する場合は、19.2MHzの３倍速である57.6MHzでARM7TDMIコアを動作させることが可能な、8KBのコード/データ共用キャッシュを搭載したASICを用いています。

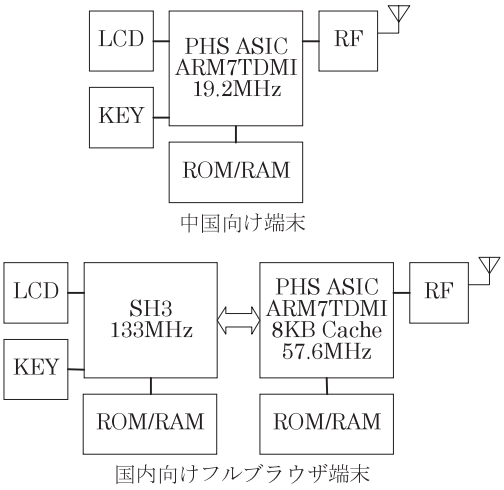

図４　PHSのブロック図

弊社では中国市場にもPHSの基地局と端末を提供していますが、中国向け端末はこの19.2MHzの1CPUで、PHSプロトコルとマン・マシンインターフェースの両方を実行しています。

国内向けのフルブラウザ端末では、57.6MHzのARM7TDMIでPHSプロトコルを実行し、133MHzのSH3でマン・マシンインターフェースを実行する2CPUのシステム構成となっています。

３．ZIPC++の適用領域

PHSプロトコルはOSI参照モデルに準拠したレイヤー1～レイヤー3とそれらの制御部から構成され、標準規格（ARIB STD-28）ではレイヤー２とレイヤー３の状態遷移が定義されています。

PIAFS、パケット通信プロトコルも同様に状態遷移が定義されています。

規格自体が状態遷移を状態遷移図とSDL図によって表現しているものなので、ZIPC++で実装するために用意されたようなシステムであるということができます。

ZIPC++によって状態遷移表を定義しているのは、これらのレイヤー２、レイヤー３、データ通信プロトコル、制御部です。

これらを合計161枚の状態遷移表によって実装しています。

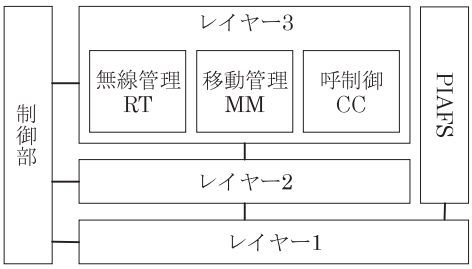

図５　PHSのプロトコルスタック

４．PHSプロトコルのクラス構成

図６がPHSプロトコルのクラス概要です。

図６において、ハードウェア抽象化部分を除くほぼ全てのクラスがZIPC++の状態遷移表によって定義されています。

図７がZIPC++の状態遷移表を駆動するための仕組みです。ここではLCHとSCCHを例にしています。

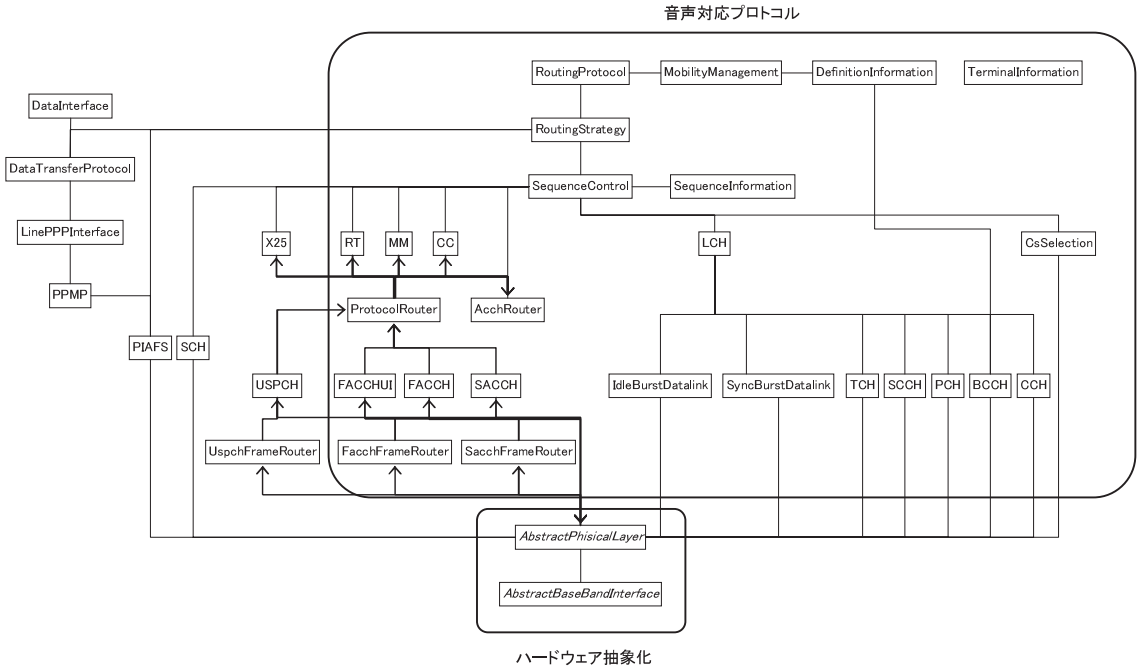

図6　PHSプロトコルのクラス概要

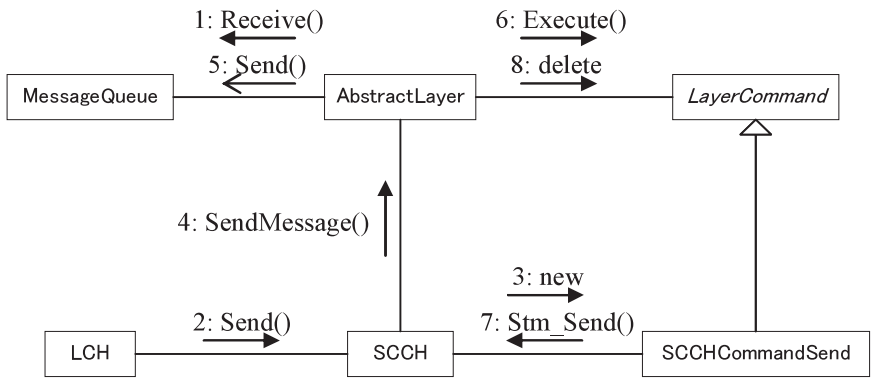

図7　状態遷移表の駆動

これらの図には、レイヤー間で受け渡されるPHSプロトコルのメッセージを格納した640個のクラス、状態遷移表を駆動するためのイベントとなる1158個のクラスは記載していません。

これらのクラスはZIPC++と連携しているUMLツール側のクラス図でも記載していません。それは、個々のクラス間に本質的な相違が無いこと、標準規格で定義されたメッセージのメンバーをそのまま持っているだけで特別な制御ロジックを持たないこと、状態遷移表のイベントを機械的にコーディングするだけであること、これらの理由からクラスの関係や動的なコラボレーションを記述するためには、基底クラスの１つだけで事足りてしまうからです。

つまり、このPHSプロトコルのソフトウェアにおいて、本質的な部分は全てZIPC++によって実装されています。

５．生成コード量

表１はコード生成及びビルド後の結果です。

		行数			ROM容量（MB）	
	ファイル数	実行行	コメント	合計	.text	合計
プロトコル全ソースファイル	4814	200000	160000	490000	1.39	1.78
ZIPC++生成ファイル	582	90000	110000	250000	0.54	0.64
ZIPC++生成率	12%	45%	69%	51%	39%	36%

ZIPC++で生成した12%のファイルが実行行数の45%、ROM容量合計の36%を占めています。

表１　ZIPC++のコード生成量

６．再利用性

表２は機能拡張とハードウェア変更の２つの事例での再利用率です。

		ファイル数			行数				
変更内容		追加	編集	全体	追加	修正	削除	合計	全体
(1) ASIC変更と		430	2932	6346	57000	54000	32000	144000	597000
128Kプロトコル追加	変更/追加率	6.8%	46.2%		9.5%	9.0%	5.4%	24.1%	
	再利用率		53.8%		90.5%	91.0%	94.6%	75.9%	
(2) ASICメーカー変更		271	497	5704	7000	6000	2000	15000	519000
	変更/追加率	4.8%	8.7%		1.3%	1.2%	0.4%	2.9%	
	再利用率		91.3%		98.7%	98.8%	99.6%	97.1%	

表２　コードの再利用率

(1) のASIC変更と128K対応とは、32Kbpsパケット通信対応であったASICとPHSプロトコルソフトウェアを、４つのスロットを同時使用する128kbpsパケット通信に対応したASICとソフトウェアに更新するという開発です。
(2) のASICメーカー変更とは、ARM7TDMIコア以外の互換性が無い、別メーカーのASICが利用できるソフトウェアに変更するという開発です。

128K対応においては、ソース行の75.9%が無変更で再利用されています。これは図６の音声プロトコルに関する部分が該当します。

ASIC変更においては、97%のソース行が無変更です。個々のＡＳＩＣへの対応は、図６のAbstractPhysicalLayer、AbstractBaseBand Interfaceの派生クラスを追加することで行なわれ、それ以外の箇所からの完全な独立を保っています。

７．設計上の特徴

基本構造の決定

PHSプロトコルのソフトウェアを、オブジェクト指向とZIPC++によって設計するにあたって、重要だったことが何点かあります。

図６でPHSプロトコルのクラス概要を示しましたが、このような構成とした背景思想を最初に上げることができます。別の言い方では、クラス分割の方針です。

これ以前のPHSプロトコルソフトウェアでは、図５に示したPHSプロトコルのレイヤーをμITRONの１つのタスクとして分割していました。この構成の本質的問題は、どうやって実装

するか、どうやって動作させるかという観点で分割されているということです。分割されたその部分が果たさなければならない責務と、実装上の制約が明確に分離されていないため、責務と制約のどちらが変化しても互いにその影響が波及してしまうことを本質的に防ぐことができない、ということです。

PHSプロトコルは、サービスの追加に合わせて毎年のようにバージョンアップによる機能追加が行なわれていましたので、責務の追加が行なわれ続けることに耐えることができる設計でなければならない、という本質的な要件がありました。従って、レイヤー＝タスクというような、ある意味で安易な設計では要件を満たせないのは明らかでした。

そこで最初の決定として、レイヤー≠タスクとし、同時にタスクという名称の仕様を取りやめ、スレッドという名称を利用することとしました。同じ名称を使っていたのでは、無意識のうちに従来と同じものになってしまう危険性があるからです。異なる概念には異なる名称を用いることが重要な点です。

レイヤーをタスクと分離するという決定をしたことで、標準規格を実現するクラス群と、それらのクラス群と実装上の制約の間を取り持つクラス群、という２つの大きな責務によって分けることが可能になりました。次の決定はプロトコルとシーケンスを分離するということです。

プロトコルとシーケンスを分離する、とは変な物言いですがこれは次のような意味です。

図８はPHS端末が発信する際の端末と基地局間の手順です。

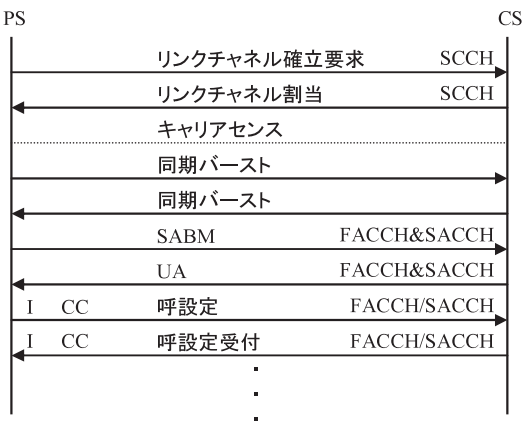

図８　PHSの発信シーケンス

リンクチャネル確立、キャリアセンス、同期バースト、SABM、UA、CC呼設定、・・・というように進んでいきますが、この順序の制御をどのように行なうのか、ということです。

ある時点で、ある種のメッセージを基地局に送信する、例えばUA受信後にCC呼設定を送信する、これをそのまま実装するという考え方では、制御対象であるプロトコルの手順が隠蔽されないまま制御を行なう側に出てきてしまっています。制御対象の具体的な手順を完全に隠蔽しつつ、複数ある制御対象が正しい順序で動作する、という要件を満たす必要があります。

そこで取り入れた考え方が次の２つです。

（1）全てのプロトコルは「接続」－「データ転送」－「切断」という手順を取る。

（2）あるプロトコルの「接続」開始のために必要となる別のプロトコルの「接続」状態、という依存関係がシーケンスを生み出す。

この考え方によって上のシーケンス例を書き直すと、SCCHの接続/接続完了→TCHの接続/接続完了→キャリアセンスの接続/接続完了→同期バーストの接続/接続完了→ACCHの接続/接続完了→CCの接続、・・・というようになります。

シーケンスを司る部分では、単に下層のプロトコルに対して順に接続要求を発行していくだけとなり、プロトコルは標準規格で定められた手順が完了したなら上位に接続完了を通知する、この単純な構造を反復することで全体を構成することができます。単純であることから、機能追加も同じことの繰り返しで容易に実現できます。

重要なことは、プロトコル同士の依存関係によってシーケンスが生成された、ということです。依存先は依存元より先に成立していなければ、依存元で利用することができない、という当たり前のことによって、

依存先を成立させる（）；

依存元で依存先を利用する（）；

というように実行順序が定まります。これの意味するところは、クラス図によって対象の構造を記述し、互いの依存関係を明らかにしていくことで、ソフトウェアの実行順序が自ら明らかになった、ということです。

つまり、あるサービスを実行するためにはど

のプロトコルが必要か、という複数プロトコルの集合を定義することで、そこには自らシーケンスが現れる、となります。

これによって、PHSプロトコル全体を、標準規格で定められた手順を実行する各プロトコルのクラス群、通信サービス毎に必要なプロトコルの集合を定義したクラス群、それらのクラス群と実装上の制約との間を取り持つ足回りのクラス群、という３種類に大別できるようになりました。

この状態ならば機能追加に対しては、新たなプロトコルの追加、新たなサービスに必要なプロトコル集合の定義の追加を行うことで、無関係な既存部分に影響を与えずに対応することができます。

スレッドの割り当て

レイヤー≠タスクとし、プロトコル毎にクラスを定義するとしたことで、実際のクラスはどのように処理を行うのかを考える必要があります。

ここはレイヤー＝タスクであったときに利用していた、メッセージキューによるスレッド間通信を行うことでよいのですが、メッセージを何にするかが問題です。

メッセージにIDを割り当てて分岐する、という方式ではIDの重複を避けるために、メッセージ定義箇所が互いに全く関連のないクラスに対して依存するという事態が生じてしまいます。これでは、無関係な既存部分に影響を与えず機能追加ができるとは言い難いです。

ここでGoFのデザインパターンからCommandパターンを利用しています。メッセージキューにはCommandのポインタを積むことにし、キューからの取り出しに１スレッドを割り当て、順次処理されるようにします。図７がそのコラボレーションです。

従来の設計ではレイヤー＝タスクであったことから、レイヤー毎に１スレッドを用意すれば問題はないと考えられます。（実際に問題はありませんでした）

このようにすることでIDの存在自体が消滅するので、メッセージ定義箇所が互いに関連のないクラスに対して依存するという事態がそもそも生じなくなります。またCommandの処理は仮想関数によるポリモフィズムで処理されるため、キューからの取り出し箇所は個々のCommand、更にはCommandの対象である個々のプロトコルからも独立した存在となります。これにより、プロトコルが増えても問題は生じません。

Commandの生成は受け側で行うようになっています。これは、呼び出し側にとっては、その処理がCommandパターンで処理されるというのは受け側の都合でしかないからです。こうすることで、呼び出し側の処理は全て通常の関数呼び出しとなり、受け側の種類によらなくなります。同時にCommandは受け側で閉じたものとなり、他に影響を与えなくなります。

状態遷移表の駆動

状態遷移表の駆動は前述したCommandパターンによって行われます。

Commandが表のイベントを呼び出し、Entry処理でCommandのポインタを現在実行中のCommandポインタとして格納します。ZIPC++の生成したコードが該当するセルを呼び出して処理が行われ、必要なら状態遷移するというように進んでいきます。

ここでセルの中に定義したコードが呼び出し側から渡された引数を利用する場合に、先ほどEntry処理で格納しておいた現在実行中のCommandポインタが利用されます。ここで問題なのは、ポインタを格納するための変数はCommandの基底クラスを型として持っていることです。このため個々のCommand固有の変数を取り出すことはこのままではできません。

解決策としては、

（1）全ての引数について、コピーするための変数を用意しておき、Entry処理でコピーを行いセルで参照する。
（2）格納するポインタ自体を個々のCommand毎に用意する。
（3）個々のCommandで定義されている変数の全てを基底クラス側で定義する。
（4）個々のCommandで定義されている変数へのアクセス関数の全てを基底クラス側で仮想関数として定義する。
（5）実行時型情報（RTTI）を用いて個々のクラスへダウンキャストする。

といった方法がありますが、実際に利用しているのは（5）の実行時型情報を用いたダウンキ

ャストです。このようにしたのは、他の方法よりも実装の手間が非常に少ないことと、未知のポインタをダウンキャストするわけではないことからです。

dynamic_castのコストが懸念されましたが、実際に問題になることはありませんでした。

８．実装上の特徴

ROM詰め、高速化

一般論として、C++で実装するとCよりも大きく遅いプログラムになると言われていますが、それに対して行った内容をご紹介します。

ROM詰めに対して簡単、かつ効果的だった内容は仮想関数の削除です。これは、初期の設計で用意はしたが実際には利用されていないもの、実装を段階的に行うために個別のクラスと仮想関数にしたが、全てを実装し終った現時点では単独のクラスと型識別用の変数で実装可能なもの、これらを整理することでした。

640個のメッセージや1158個のイベントという具合に、共通の基底クラスからの大量の派生クラスが存在しているので、仮想関数を１つ削除すると640×4バイト＝2560バイト、1158×4バイト＝4632バイトというように、Vtableで使用している分の容量が減っていきます。更に実装部分でも容量が減っていきます。また、未使用であった箇所や、拡張が行われなくなった箇所のため、弊害が発生する危険性や拡張性への妨げとならないことも好都合です。

仮想関数の削除とそれに伴う実装の見直しの結果、音声プロトコル部で約10%のROM容量を削減した実績が残っています。これを行なったのは最初に実装してから２年が経過して安定した後であり、設計の見直しが安全に行なえる、いわゆる「枯れた」状態であったということが重要な点です。

このことから、JavaのObjectクラスのような全てのクラスの基底クラスを組み込み開発で定義し、使用するかどうか分からない仮想関数を用意することは、ヒープ内のインスタンスの型を全てダンプする必要があるとか、ROMの使用量を増やしたいといった理由がない限り、避けるべきです。

ROM詰めと高速化の両方に効果があったのはクラス変数へのアクセス関数（ゲッター/セッター）に対するインライン関数の利用です。インライン関数の場合、クラスによる情報の隠蔽を維持したまま関数呼び出しが削除されるので、関数のエントリアドレスのロードとジャンプ及びリターンが不要になります。一箇所では10数バイトにしかなりませんが、塵も積もれば山となる、の諺どおり、１回の試験ビルドで労せずして数Kバイトが得られることがあります。

また、インライン展開されることでコンパイラによる最適化がレジスタレベルで行われるようになり、それによってもROM使用量と速度の両方が改善します。ジャンプが行われないようになることで、パイプラインが乱れない、キャッシュへの読み込みが効率的に動作するということも高速動作時には重要になります。

高速化のためには、変数がメモリ上でどのように配置されているか、ということにも注意することが必要です。

アルゴリズム上では1バイトのデータ配列であっても、32ビットCPUなら4バイト単位でアクセスする方が、メモリバスの使用効率が上がります。

16ビットバスのシステムなら、8ビットと16ビットのデータに対するアクセスコストは同一です。

memcpyなどの標準ライブラリ関数では、そのような点を考慮したコードになっている場合があります。

主に配列データ処理を行なうクラスでは、変数定義時に共用体や#pragmaを利用するなどして、アクセス効率が最も良くなる配置となるように注意を払うべきです。

実際に速度に関して最も影響があったのはOSでした。先に１スロットで160ビット＝20バイトのユーザーデータを受信すると説明しましたが、受信割り込み時にはコンテキストスイッチのために16個のレジスタの退避とロードが行われています。32ビット×16個＝512ビット＝64バイトですから、実際に処理されるデータは受信データよりもOSのデータの方が3倍以上多いのです。

このように、実際の処理をプロファイルしていくとC++だから遅いということはなく、それ以外の要因によるという状況でした。

また、仮想関数呼び出しは条件分岐より高速に処理される場合があります。仮想関数の呼び

出しは、ポインタを読み出し、Vtableを参照してジャンプとなり、クラスが何種類あっても一定時間となるのに対し、条件分岐では変数を読み出し、比較してジャンプとなりますから、３つ以上の分岐となった時点で不利になります。

このことは、派生クラスの数が少なく、一つのクラスの仮想関数も少ない、クラスが追加される可能性もなく、実行時にインスタンスとなるクラスが切り替わることもない、という設計モデルではC++の弊害が出る可能性がある、逆にこれらを満たしていないモデルをCで組むと、大量の条件分岐が随所に現れて複雑なプログラムとなってしまうということかもしれません。

オブジェクト指向設計によって効果が出るプログラム対象であるかどうか、オブジェクト指向設計の効果を得ている設計モデルであるかどうかも、前記の要素で判断することが可能ではないかと思います。

デバッグ用ログ

状態遷移表のセルで実行時型情報（RTTI）を利用していることを説明しましたが、RTTIはデバッグ用のログを残すためにも利用しています。

メッセージキューからCommandを取り出す時に、Commandのクラス名を取得しログに記録しています。このような箇所は他にもありますが、これの利点はログ用の文字列をコンパイラが自動的に用意してくれるということ、新しいプロトコル関係のクラスを追加すれば自動的にログが残るということです。

クラス名がそのままROMを消費しますが、動作がCommandによって制御されているため、Commandの履歴が完全に残ることによって得られる動作解析の容易さは欠点を補って余りあるものでした。それが問題にならない状況ではRTTIを有効にしておくべきではないかと思います。

９．オブジェクト指向設計とZIPC++によってもたらされたもの

オブジェクト指向設計を行い、設計モデルをUMLで記述し、状態遷移表で振る舞いを記述して得られた最大のものは、ソフトウェアの完全な設計図であると言えるでしょう。

特にZIPC++を利用している状態遷移表は、常に実装と100%一致した状態が保たれるため、デバッグしていった結果、最初の設計とは細部が全然異なるとか、設計を実装に一致させる保守工数が手配できないといったことと無縁になったのは非常に大きい効果です。

また、ZIPC++での開発作業自体が簡便で、余計な工数を費やすことがないために、開発メンバーが使い続けることができる、ということが継続できた理由でもあります。

UMLによる設計モデル記述の効果は、やはりコミュニケーションが容易になったことです。利用可能なデザインパターンを取り入れ、責務を適切に表現したクラス名を与えることで、クラス図を見れば概略の振る舞いを読み取ることができるということの効果は非常に大きいです。それは海外の開発者との間であっても同じであった、ということは特筆すべきでしょう。

概念を整理し、それによい名前を付けることができれば、設計の大部分は終ったようなものだ、ということかもしれません。

10．おわりに

状態遷移表設計は非常に優れた手法ですが、それだけでは解決しない問題が残されていると思っています。

それは、状態遷移表は遷移の漏れや抜けを検出することはできますが、遷移の妥当性を保証することはできない、ということです。

別の言い方をすれば、文法が正しいからといって、その文章の内容が正しいとは限らない、ということです。

これは状態遷移表設計に限った話ではなく、ソフトウェア開発のあらゆる場面で生じていることではありますが、未来のZIPC++では状態遷移表が表現している対象の意味を取り扱うことができるようになり、状態と事象の意味から遷移の妥当性判断まで可能になるように、今後もキャッツ様には研究開発を意欲的に続けていただきたいと思います。

（ほとんど人工知能を開発して欲しいと言っているような気がしますが・・・）