ソフトウェア工学 107－17
（1996．1．19）

ウィンドウシステムの移植作業におけるソフトウェア工学的評価

平井孝史　小島大吾　玉山尚太朗　早川栄一　並木美太郎　高橋延匡

東京農工大学

本稿では、ウィンドウシステムの移植時に起きたバグ及びデバッグ作業の報告とそれに対するソフトウェア工学から見た評価について述べる。今回われわれは、当研究室において独自に開発されたウィンドウシステムである“未”の移植を三人で分担して行った。この移植に関しては、複数人での作業であること、異なる OS アーキテクチャへの移植であること、などが特徴であり、関数の二重定義、インタフェースミス、などのバグが発生した。これらのことから、複数人での作業においては協調作業環境の必要性、ウィンドウシステムの移植においてはその画面表示の多様性からデバッグのための画面のハードウェアやツールが必要、などを知見として得た。

Software Engineering Evaluation of the Transplantation of a Window System

HIRAI Takashi, KOJIMA Daigo, TAMAYAMA Syoutarou,
HAYAKAWA Eiichi, NAMIKI Mitarou, TAKAHASHI Nobumasa

Tokyo University of Agriculture and Technology

This paper describes the work report of bugs and debugging in the transplantation of a window system, and the software engineering evaluation of them. Three members shared the work of transplanting the window system developed originally in our laboratory. This transplantation involved group work and transplantation to a different OS architecture. Bugs such as multiple declarations of functions, and mistakes in the interface accused. From these, we attained knowledge such as the necessity of an environment for cooperative work in group work, and the necessity of hardware and tools for displaying debugging, this coming from the diversity of display in the transplantation of the window system.

1. はじめに

われわれの研究室には、ペンの UI の研究基盤として独自に開発されたウィンドウシステムである“未”が存在する。“未”は 1993 年に開発されたあと、われわれの研究室の在籍者によって保守・改良され、現在も稼動、その上で分散手書き KJ 法やペンを用いたアニメーション記述言語などの研究も行われている。ここで、現在の“未”においては、動作するハードウェアやその発色数等の問題があり、これらのことから、“未”の移植の要望が高まってきた。

本稿では、この“未”のプログラムを移植した際に発生したバグをまとめたものについて考察を述べる。

なおこれ以降、現在の“未”を「“未”V3」、移植された“未”を「“未”98」、特にどちらかの限定はなく“未”一般を指すときに単に「“未”」と呼ぶことにする。

2. “未”V3 の問題点

“未”V3 の問題点として次のことが挙げられる。

(1) 動作するハードウェア

“未”V3 は、日立製ワークステーション 2050 の上で動作しており、ペンの環境として表示一体型液晶タブレットを用いている。

2050 は 198X 年に発表され CPU として 68020 を用いた計算機であり、現在においては特殊なハードウェアであり性能にも問題がある。

また、この表示一体型液晶タブレットは高解像度（1000 × 700）という利点を持っているが、試作品であるため保守が困難となっている。

上記のことから、動作するハードウェアとして、量産されているタブレットを使用できる、パーソナルな計算機が望まれる。

(2) 発色数

“未”が用いているハードウェアとも大きく関係しているが、“未”は白黒二色のモノクロ表示のウィンドウシステムである。これは、既存のウィンドウシステムと比べると頼りない。GUI としての表現力もモノクロとカラーではその差は歴然としている。今後“未”を用いた研究が多く行われるとすると、“未”上で動作実現する AP にカラーを用いたいという要求は、これからさらに増えるといえる。

(3) フォント

“未”V3 のフォントは、その大きさだけを指定することしかできず、種類は固定であり別のフォントや任意のフォントを使用することはできない。

また、AP が使用できるフォントだけでなく、“未”そのものが使用できるフォントも状況に応じて使い分けたいという要求がある。しかし“未”V3 のフォント構成では要求に柔軟に対応することはできない。

そこで現在のフォント環境をマルチフォント環境にして、ビットマップフォントやアウトラインフォントをユーザが自由に選択できるようにしたい。

これらの要求から、“未”V3 をパーソナルな計算機へ移植を行うことにした。

3. “未”V3 の移植

3. 1 “未”V3 の構成

“未”V3 は次のような構成になっている（図 1）。

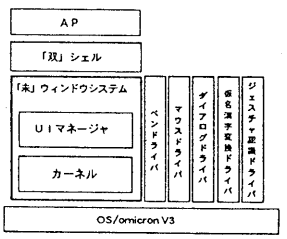


図 1 “未”V3 の構成

“未”V3 は大きく二つに分けられており、ウィンドウシステムそのものの機能を持つ「カーネル」と、ウィンドウシステムとしての UI を決定している「UI マネージャ」からなっている。そしてこれらはそれぞれマルチタスクで動作している。

3. 2 移植の目標

移植を行う際の目標を次のこととした。

(1) ハードウェア

“未”98 を動作させるハードウェアは、パーソナル

であるとともに、カラー化を行いたいため（そのためには画面の構造も変更しなくてはならない）にその内部が公開されているものでなくてはならない。そのためここでは、NEC の PC9801（解像度 640 × 400, 16 色）を移植するハードウェアとした。

(2) カラー化

“未” V3 ではハードウェアの制約からモノクロであったが、問題点としても述べたようにウィンドウシステムとしてモノクロ表示では頼りない。またハードウェアを選択したのもカラー化を行いたいためである。そこで今回の移植の際には“未”のカラー化を行うこととした。

(3) フォント

今回ハードウェアの変更にともなってディスプレイの解像度も変更になる。そうすると今までと同じ大きさの文字フォントでは画面のバランスがおかしくなる可能性がある。またこれから先、ディスプレイの解像度が変更される可能性もある。そのときに限られたフォントしか使えないのは不便である。そのため移植の際には現在のフォント構成をマルチフォント環境とし、ユーザがフォントの変更や追加ができるようにする。

(4) OS

今回ようにハードウェアの変更にともない CPU アーキテクチャも変更されれば当然その上で動くべき OS も変更になる。“未” V3 は当研究室で開発された OS /omicron V3（以下 V3）の上で動作している。これは 680x0 をターゲットとしており、PC9801 では動作しない。

ここで、現在当研究室では、80x86 (x ≧ 3) をターゲットとした OS/omicron V4（以下 V4）を開発中である。そのため移植の際の OS の最終的な目標は V4 とする。しかし開発中であるので“未” V3 の移植には用いることはできない。

そのため、今回は暫定的に V4 への橋渡し的な役割として MS-DOS + DOS extender という動作環境を用いることにした。つまり、マルチタスクからシングルタスクへの移植を行うことになった。

またコンパイラに関しては、“未” V3 はこれもわれわれの研究室で開発された CAT という内部コードがフル 2 バイトの C 言語を用いてコーディングされている。移植を行う際にはソースコードの再利用を行いたいため、その環境で同じようなフル 2 バイトのコンパイラが必要になる。そのため、コンパイラはこれも当研究室で開発された 80386 用コードを生成するフル 2 バイト C コンパイラである、CAT386 を利用することとした。

3. 3 移植の方針

また移植を行う際の方針は次のこととした。

(1) 早期稼動を目指す

今回の作業はあくまで「移植作業」であるので、現在の“未”に対して機能拡張や改良などの変更をを多く行ったりすることはせず、まず移植し稼動させることを第一とする。機能拡張に関しては多くの要求があると考えられるが、それらは移植作業が終わった後で考えるべきこととする。

(2) ソースの再利用をする

移植をする際の最大のメリットは以前作られたソースの再利用ができることである。開発の手間をできるだけ軽減するために、ソースに変更を加えることなく、できるだけ再利用をする。変更によってバグが引き起こされてはその分手間が増えてしまい、移植をするメリットが減ってしまう。

(3) 拡張性を考える

今回の移植作業はターゲットマシンとして PC98 としたが、今後別の計算機に移植をしたいというような要求があるかもしれない。そのような場合、ハードウェア依存部をモジュール分けしておけばその部分だけを取り替えればすむ。今回の移植作業をそれ自身で閉じてしまうことをせず、これから先のことを考えて拡張が容易にできるよう、コーディング等行うこととする。

3. 4 変更点にともなう再設計

“未” V3 の問題点から、移植の際には次のことを変更する。

(1) ハードウェア
(2) 発色数
(3) フォント構成
(4) OS

これらの変更点の再設計についての概要を次に述べる。

(1) ハードウェア

“未” V3 においてハードウェア依存の部分は、ディスプレイドライバ、マウスドライバ、描画ライブラリとフォントドライバである。

これらはアセンブラで書かれているため、移植の際にはそれぞれ新規に作成する必要がある。ここで、ディスプレイドライバ以外はソースコードの保守性から C で書くことにした。ディスプレイドライバに関しては速度をできるだけ早くしたいことからアセンブラで記述することにした。

(2) 発色数

今回の移植では 16 色を実現するが、将来的にはもっと多くの色を扱えるようにし、状況に応じて発色数（解像度も）を変更できるようにしたい。それには現在の画面構成であるプレーン方式では扱いづらい。そこで移植の際には画面構成をプレーン方式からパックドピクセル方式に変更することにした。

(3) フォント構成

現在のフォントドライバは、それ一つでフォントデータの取得とフォントデータの表示を行っているため、別のフォントを表示させることはできない（図 2）。

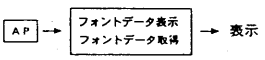


図 2 “未” V3 のフォント

そこで取得部と表示部を分け、さらにそのあいだに取得部を管理する機構を設け、自由にフォントを利用できるようにした（図 3）。

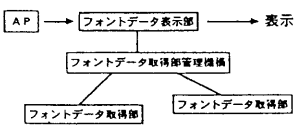


図 3 “未” 98 のフォント

(4) OS

“未” V3 はマルチタスクであったが、移植の際にはシングルタスクにしなくてはならない。そのため、次の点に気をつけなければならない。

・無限ループを書くことができない
・並列処理を逐次処理に直す必要がある

特に二番目については処理の流れを考えないと、順番によっては動作しないものもあると考えられる。

マルチタスクからシングルタスクへの変更によって “未” 98 は次のような構成になる（図 4）。

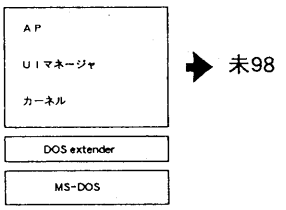


図 7 “未” 98 の構成

3. 5 移植作業

(1) 分担

移植作業は三人で次のように分担させた。

担当者	担当部
A	・描画ドライバ ・描画ライブラリ
B	・マウスドライバ ・カーネル
C	・UI マネージャ ・AP 用ライブラリ ・フォント

Aはハードウェアに実際に描画をさせる描画ドライバとそれを扱うためのライブラリ。Bは “未” のカーネルとマウスドライバ。Cは “未” の UI マネージャと AP が “未” を使用するためのライブラリとフォント部分。このように担当させた。

(2) 作業期間

作業期間は次のとおりである。

1994-09	移植プロジェクト開始
10	移植の方針・変更点・分担　決定
11	ここから各自の作業に入る
	元ソースの理解
	心ソースの作成
1995-05	各自の作業終わり
	デバッグ開始
06	完成

移植のプロジェクトが動き始めたのは 1994 年の 9 月であり、それから各自の担当の作業に入ったのは 2 ヶ月後であった。

各自のソースの移植が終わり、リンクしデバッグを開始したのは 5 月の終わりである。それからデバッグの終了までにおよそ 10 日かかった。

4. 結果と考察

ここでは移植の際に発生したバグについての考察を述べる。

4. 1　ソースの再利用率

ソースの再利用率

	ヘッダファイル	ソースファイル
A	175 / 868 (20 %)	760 / 3300 (23 %)
B	704 / 1863 (38 %)	11368 / 13476 (84 %)
C	1520 / 2456 (62 %)	9394 / 10867 (86 %)

ソースの再利用率はAの担当分の 23% が最も低い。これは、Aの担当分である描画ドライバ・ライブラリはハードウェア依存度が高く、ドライバは完全に新規作成であるし、ライブラリに関しても以前の関数名で使えるように“未” V3 の形式にあわせるところ以外は新規作成を行ったためである。つまり再利用は以前のものとのインタフェースをとるためだけに行われた。ハードウェア依存部は再利用がしにくいといえる。

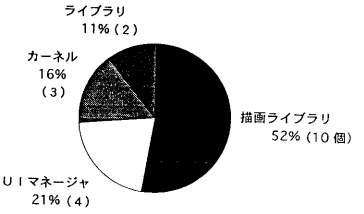
“未” V3 本体を移植したBとCはどちらも 80% 以上の再利用率となっている。これは OS 依存部分以外はほとんど再利用できたためである。これらはどちらも 10000 行を越えているが、行数の規模に比べ移植の作業は容易に行えた。

“未” V3 は、ファイル単位で見ればカーネルが 17 ファイル、UI マネージャが 11 ファイルと、その内部が細かくモジュール分けされている。移植はファイルごとに内容を理解すると同時にハードウェア依存部と OS 依存部がないかを調べ、必要ならば変更をする、という手順をとった。またカーネルでは 9 ファイル、UI マネージャでは 3 ファイルは変更を行う必要がなかった。

これらの再利用率の高さとモジュール分けの細かさ、またシステムそのものがカーネルと UI マネージャの二つに分けられていることから、“未” は再利用が行いやすいといえる。

4. 2　バグの発生した場所

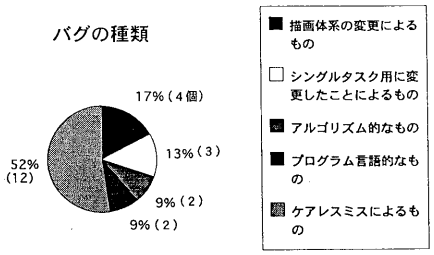
バグの発生した場所



バグは描画ライブラリに 10 件発生し、52%（10 個）と全体の半分を占めている。先のソースの再利用率からもわかるように、これは描画ライブラリが、カーネルや UI マネージャのような単純な移植作業よりも、新規に作成した部分が多いことが原因として考えられる。以前に一度完成しているソースを再利用するよりも、新規に作成したソースにより多くバグが含まれてしまうのは仕方のないことである。

また、このように描画に関して多くのバグが発生することは、描画の処理を多く行わなければならないウィンドウシステムの開発に特有のことと考えられる。

4. 3 バグの種類



一番多いのはケアレスミスで、52%（12 個）と全体の半分を占めている。人間が作業をしている以上避けれらないことだろうとは思うが、それにしても多い。そこで、これらのケアレスミスをもっと細かく分析してみる。なぜこのようなケアレスミスが起きたのかを考えてみた。

これらがなぜ起きたのかを表にまとめてみると次のようになる。

バグ	原因
・未定義シンボル	三人の連絡の不徹底
・マクロの使用ミス ・インクルードファイルの変更ミス ・ID の二重変換 ・free() ミス ・関数の返値のミス	設計をせずに変更・作成
・オブジェクトファイルの転送ミス	開発環境
・アイコンデータの変数型宣言忘れ ・変数の代入ミス	ケアレスミス
・渡す引数のミス	変更前のソースに存在

このようにしてみると、ケアレスミスが原因で起きたバグも、「三人の連絡の不徹底」や「設計をせずに変更・作成」などのバグは未然に防げたものだと考えられる。そうすればケアレスミスの数は半分に減らすことができたはずである。

特に多いのは 5 個あった、「設計をせずに変更・作成」したためであり、これは移植を行う際、変更する必要のあるところをその場で変更等し、半ば行き当たりばったりなコーディングをしたことであった。例えば表中の「マクロの使用ミス」は、配列の大きさを指定するマクロと、その配列を初期化するループで回す条件式のマクロとが違っていたものだった。

プログラムを作成するときに設計を行っておくことは当然のことであるが、それをせずに作業を進めてしまったのは大きな反省点である。移植を行う際にも変更の必要のあるところはきちんと設計をしておく必要があることを痛感した。

ケアレスミスの中で、「渡す引数のミス」は、変更前の“未”V3 のソースに含まれていたもので、今回移植をして初めてそのバグの存在が明らかになったものである。“未”V3 では偶然動いていたのだと考えられる。

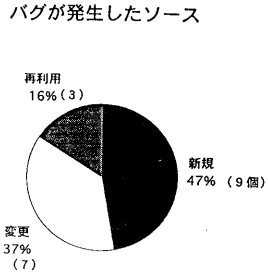
ケアレスミスの次に多いバグの種類は描画体系の変更によるものの 17%（4 個）とシングルタスク用に変更したことによるものの 13%（3 個）である。

いいかえると描画体系の変更はハードウェア依存部の変更であり、シングルタスク用への変更は OS 依存部の変更である。つまりこれらは他のバグの種類に比べ今回の移植の際にソースの変更量が多いもので、その分バグを含みやすくなっているといえる。

またこのうちのいくつかは、新たに作成したライブラリの使い方を間違えていたというような、インタフェースをとるところで起きている。

今回三人で作業を進めたが、これらは連絡を取り合っていれば未然に防げたものだと考えられる。複数人で開発作業を行う際には、そのインタフェースをどのようにとるのかをお互いがしっかり把握しておかなければならないと強く感じた。

4. 4 バグが発生したソース

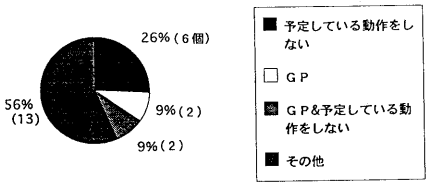


バグが発生したソースは、再利用をしなかった部分（新規作成分と変更分）が 84% とその大半を占めている。再利用をした部分は移植をする前の段階でデバッグが行われているのでバグは含まれていないはずであるが、新たに人の手が加わった部分はその分新たなバグが発生しやすい。変更分の 7 個に比べ、新規作成分に 9 個とより多くのバグが発生したのも、新たに人の手が加わった割合が多いためである。

また、今回移植をすることで以前のソースにバグがあったことがわかったケースがあり、再利用をした部分にはバグが含まれていないと断言することはできない。

4. 5 バグの症状

バグの症状



バグの症状として最も多かったのは、「予定している動作をしない」というもので 26%（6 個）である。これは例えば、ここは直線が引かれているはずなのに引かれていない、ウィンドウの操作ができるはずなのにできない、イベントが発生しているはずなのに発生していない、などである。

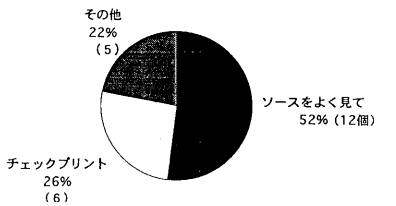
これらのバグは、その位置を特定するのに時間がかかった。ウィンドウシステムという大規模なシステムプログラムでは、その内部も多くのルーチンからなっているため、それら一つ一つに、またそれらを含んだ大きなまとまりに気を配るようなことをしないと、ある一つのバグ入りルーチンが他の正しいルーチンへ影響を与えることが多くある。システムを局所的かつ大局的に把握している必要がある。

GP（General Protect）はポインタの指す先が不正で、症状としては突然プログラムが終了するものである。突然プログラムが終了するため、どこでバグが発生したのかを見極めるためにソースコード中に逐次チェックプリントを入れた。ただしこのバグは場所が特定できてもその理由は別のところにあることが多い。例えば配列のインデックスが配列宣言の大きさを越えていたため GP が起きたとわかっても、そのインデックスの中身はどこで決定されたのかはまた探す必要がある。そのため、GP は 9%（2 個）と数は少ないが解決に多くの時間を必要とした。

また、「GP & 予定している動作をしない」とは GP が起きたり予定している動作をしなかったり場面場面によってその動作が安定していないバグである。これはいわば上記の二つのバグが融合しているものであり、GP と同様 9%（2 個）と数は少ないが、解決するのに最も時間のかかるバグである。

4. 6 バグの発見方法

バグの発見方法



バグの発見方法として最も多かったのは「ソースをよく見て」というもので 52%（12 個）と全体の半分を占めている。これは、バグの症状からまず誰の担当分かを予測し、次にその担当分のソースのこのモジュールだろうという予測をたて、関連する処理を目で追っていく、というものである。

これは最も原始的なデバッグ方法であり効率は悪い。バグのありそうな箇所を特定するのにも時間のかかるものになってしまう。複数人での作業においては誰の担当分かを間違っている場合もあるし、誰の担当分かがわからない場合さえある。そのような場合にこの方法ではもっと時間がかかってしまう。

また、今回のわれわれのデバッグ環境は、実行環境とまったく変わらず、デバッガもなく計算機に支援させることといえばソースコード中にチェックプリントを挿入するくらいしかなかった。尚チェックプリントの出力先はディスプレイにしてしまうとウィンドウシ

ステムの画面がよくわからなくなってしまうのでファイルに書き出すようにした。

チェックプリントはあまり多く使用すると実効速度の低下につながってしまうし、そのたびにコンパイル・リンクしなくてはならないので効率がは悪い。ソースコードも見づらいものになってしまい、やはり多用はしたくない。

5. 得られた知見

ここでは、プログラムの移植作業から得られた知見について述べる。

(1) 複数人での作業

今回の作業の大きな特徴として、三人（つまり複数人）で作業を行ったことが挙げられる。それに起因する問題として、インタフェースのとり方に関したバグが発生したことがある。これはお互いの連絡の不徹底がその原因であり、未然に防げたはずのことである。このようなことをなくすためには、インタフェースのとられるべき部分はきちんと公開し、また公開された方はそれを正確に把握しておく必要がある。

また、複数人で作業をしたことによって誰のバグなのかよくわからないといった状態がよくあった。これはお互いの担当部分の不理解がその原因として考えられるが、さらに加えてそれはお互いのことを自らはなかなか知ろうとしないことが原因として考えられる。そこで、このようなことをなくすには、自分のしたこと・していること・することを公開しまた公開された方はそれを正確に把握しておく必要がある。

(2) 異なる OS への移植

今回の移植では、マルチタスクの OS からシングルタスクの OS への移植を行った。ここで、実際の作業に移る前に OS の違いからくる変更点は何か考えた時には、マルチタスク実行関数はできない、無限ループはできない、セマフォはいらない、SVC コールは使えない、また結果的に複数のモジュールを一つにまとめることから関数名などの衝突をなくさなければならない、などであった。しかし実際には、セマフォではなくある変数を用いて排他のようなことを行っていたり、すべてが逐次処理で行われるために処理の流れを考える必要があったりと、あとから別の問題が、それもそれなりの重要度を持って発覚した。

ここで考えてみると、当初に考えていたことは、ソースコードから簡単に読みとれる、表面的で記号的に OS に依存すること（例えば、～セマフォ～とでてきたらその処理はいらない、while(1)～とでてきたらそれは削除する必要がある、など）であり、その処理内容に関しては OS に依存するであろう事を考えていなかった。

つまり、異なる OS に移植をする際には、表面的に読みとれる OS 依存の処理だけでなく、その処理内容や処理方法も OS 依存のところはないか、よく調べて考える必要がある。

6. おわりに

本稿では、ウィンドウシステムの移植と移植時に発生したバグについての考察及びそれから得られた知見について述べた。“未”98 は現在のところ特に問題なく動作しており、アウトラインフォントも使用できるようになった。ただ、テストプログラムとして動作させた AP（百数十行で“未”のプログラムとしては小規模のものを数個）はそれだけでは十分とはいえない。今後はもっと規模の大きな AP も動作させてみる必要がある。

謝辞

電気通信大学の藤野喜一教授には、当大学との合同セミナーの折に多くの助言をしていただき非常に参考になった。ここに深く感謝の意を表す。

参考文献

[1] 津田道夫　他：ソフトウェア変更作業の分析と支援機能，情報処理学会，ソフトウェア工学研究会，102-1，1995

[2] 丸山勝久　他：ソースコード再利用における能動的部品変化メカニズム，情報処理学会，ソフトウェア工学研究会，102-13，1995

[3] 秋山義博：大規模複雑プログラム理解のためのプログラム分析と可視化技術，情報処理学会，ソフトウェア工学研究会，102-6，1995