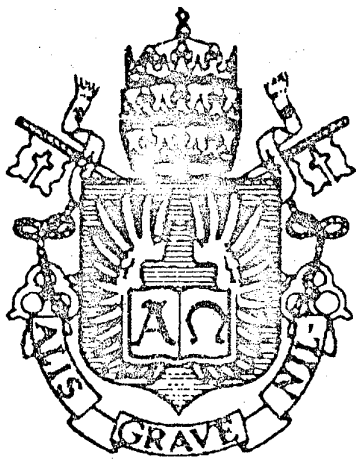


PUC

ISSN 0103-9741

Monografias em Ciência da Computação
nº 37/92

Decoupling Interface and Implementation: the TOOL solution

Sérgio E. R. Carvalho

Departamento de Informática

PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO DE JANEIRO
RUA MARQUÊS DE SÃO VICENTE, 225 - CEP 22453-900
RIO DE JANEIRO - BRASIL

PUC RIO - DEPARTAMENTO DE INFORMÁTICA ISSN 0103-9741

Monografias em Ciência da Computação, Nº 37/92
Editor: Carlos J. P. Lucena Dezembro, 1992

Decoupling Interface and Implementation: the TOOL solution*

Sérgio E. R. Carvalho

* This work has been sponsored by the Secretaria de Ciência e Tecnologia da Presidência da República Federativa do Brasil.

In charge of publications:
Rosane Teles Lins Castilho
Assessoria de Biblioteca, Documentação e Informação
PUC Rio — Departamento de Informática
Rua Marquês de São Vicente, 225 — Gávea
22453-900 — Rio de Janeiro, RJ
Brasil
Tel. +55-21-529 9386 Telex +55-21-31048 Fax +55-21-511 5645
E-mail: rosane@inf.puc-rio.br
techrep@inf.puc-rio.br (for publications only)

DECOUPLING INTERFACE AND IMPLEMENTATION: THE TOOL SOLUTION

S. Carvalho

ABSTRACT

It is expected that a large portion of the software systems to be constructed in this decade will involve object orientation, message passing and graphical user interfaces. Design methodologies for such systems are now being studied. A typical goal pursued in these methodologies is the separation between the interface and the implementation of an application. In this report a design approach providing a high interface-implementation decoupling degree is proposed. This approach takes full advantage of object orientation, by considering the handling of asynchronous messages as part of object behavior. The existence of TOOL, a programming system containing asynchronism at this high level, suggests the construction of mechanisms for automatic design-program translation.

RESUMO

Espera-se que uma grande parte dos sistemas de software que serão desenvolvidos nesta década conterá orientação a objetos, passagem de mensagens, e interfaces gráficas para usuários. Metodologias de projeto para estes sistemas estão agora sendo estudadas. Um objetivo tipico destas metodologias é a separação da interface de um programa da sua implementação . Neste relatório um enfoque de projeto que produz um alto desacoplamento entre interface e implementação é proposto. Este enfoque se aproveita de uma extensão do conceito de orientação a objetos, onde o tratamento de mensagens assíncronas é considerado como parte do comportamento de objetos. A existência de TOOL, um sistema de programação que contem assincronismo neste alto nível, sugere a construção de mecanismos para a tradução automática de projetos para programas.

KEYWORDS AND PHRASES

object orientation, message passing and receiving, graphical user interface, design methodology, interface-implementation decoupling, asynchronous object behavior, the language TOOL.

ACKNOWLEDGEMENTS

The TOOL programming system is the result of a joint research and development project conducted by SPA (Sistemas, Planejamento e Análise), a software house in Rio, and a group from the Departamento de Informática of the Pontificia Universidade Católica, also in Rio. The system contains a programming language and its compiler, a linker, an abstract machine and run-time support, a library of classes, and an environment for the friendly construction of TOOL programs. This report must be understood as another result from this same team effort, although the author takes sole responsibility for its contents. Special thanks are due to Otavio P. Coelho, Nelson Gorini, and Werther J. Vervloet, early day team members. Thanks are also extended to the younger generation, Mario S. Cardoso, Eduardo Galucio and Toacy C. Oliveira, and to several PUC students and colleagues who contributed to the production of this report.

1. INTRODUCTION

There is a strong belief that a reasonably large portion of the software systems to be constructed in this decade will involve three important programming aspects:

- graphical user interfaces;
- message passing;
- object orientation.

Each and every one of these aspects is related to programming languages and techniques which lie outside the regular working environment of most conventional language programmers today. In fact, those dealing primarily with procedural (or conventional) languages are not typically addressing, for example, the construction of objects that, being displayed on a screen, allow the user of a program to asynchronously interfere with the execution of that program.

In order to design and implement modern software systems adequate resources must be available, as for example:

- software platforms;
- programming languages;
- design methodologies.

Several adequate platforms for the development of modern programming systems exist today. As an example we could mention the Windows operating system from Microsoft, which includes message passing features and user interface objects.

Unfortunately, adequate languages where message passing and graphical interfaces can be coded and constructed with ease are more difficult to find. In fact most languages being used today for such purposes are derivations of conventional languages, such as C and Pascal, which were not primarily designed to handle messages. The result is that even for programmers familiar with the basic implementations of such languages, the understanding of the new object oriented and message passing versions is a hard task, often left incomplete. Besides, programmers must also become familiar with large and unstructured sets of functions, in order to code message passing.

Design methodologies for the construction of modern systems are also being discussed [CILS 92, LCIS 92, VPCCB 91]. An important common consideration in some of these methodologies is the proposed separation of the interface part (which loosely concentrates the objects seen by the user of the system) from the implementation part (where the "real" work is done). This seems justifiable enough, if considered, for example, from the reuse point of view: the higher the degree of decoupling between interface and implementation, the higher the possibility of reuse of any of these parts when the other changes. Unfortunately, even though object orientation and graphical interfaces are taken into consideration in proposed

methodologies, when it comes to message passing one tends to see mainly the old-fashioned message dispatcher, at a lower level a loop consulting a queue in order to decode and treat messages, and at a higher level an object which treats all messages relating to some user interface.

In this paper we propose a design methodology in which interface and implementation are highly decoupled. This is achieved, briefly, by taking full advantage of the object orientation concept, through the inclusion, in classes, of message sending and receiving object behavior. (A welcome side effect of this inclusion is that dispatchers are no longer needed; in other words, they can be relegated to the operating system level, where they actually belong.) It seems also appropriate to mention that the design features proposed below can be easily translated into a new, high level programming language built specifically for the construction of the software systems under consideration here. As a convenient consequence, the design-program mapping is more strightforward.

At this point, a disclaimer may be in order. What we propose is yet another yardstick, one that can be used, perhaps when all others have failed, to choose one among several possible designs. We recognize the fact that, especially for small systems, the proposed separation between interface and implementation, in this (possibly as well as in any other) methodology, may lead to undesirable results, even increasing the complexity of the final product.

This paper is organized as follows. The methodology proposed is arrived at through a series of designs, of increasing decoupling degrees, which relate to a case study: the construction of a program to play TicTacToe. This is done in section 3. In section 2 we shortly describe language features which are important to the understanding of the case study solutions that follow. In section 4 we present our conclusions.

2. THE LANGUAGE TOOL

TOOL is a new object-oriented and message-driven programming system built to simplify the construction of programs that involve their users as components: through controls disposed in one or more interface screens, users activate routines that realize their wishes. For example, by "pressing" a button labelled "Cancel", the user generates an event that interrupts the current processing; by selecting the menu option "Edit", the user generates an event that places on the screen a submenu displaying the editing options available.

The current version of TOOL runs on Microsoft Windows; other modern software platforms will be used in the future. A concise description of TOOL can be found in [CARV 92]; the language is fully described in [SPA 92].

Program construction in TOOL is simplified by:

- true object orientation, allowing both a structured definition of interface objects and an elegant and high level mechanism for message handling;

- a view of object orientation acceptable to real life programmers, where "new" concepts such as *class, inheritance, method, polymorphism* and the like can be explained from knowledge already in possession of conventional language programmers;
- a friendly, Windows-like application environment for program construction, compilation, linking and execution.

The main feature of TOOL is the *class*, which encapsulates the attributes necessary to define object structure and behavior. Object structure is obtained from a record-like declaration; object behavior can be expressed either with procedures and functions (called *methods* in TOOL), or with handlers for asynchronous messages. A class in TOOL is a passive entity, used mainly during compilation, to validate object use. Besides modeling objects, classes can be used in the definition of new classes, through a simple inheritance mechanism.

Three possible relationships can be established between two objects in TOOL: the *is-a* relationship, defined by inheritance; the *has-a* relationship, defined by structure declaration; and the *owns* relationship, defined at object creation. An object x owns an object y if the creation of y takes place within an operation being applied to x. The owns relationship is fundamental in the obtention of a high decoupling degree between interface and implementation, as we shall see later.

A TOOL program is a sequence of classes. Since TOOL classes are passive structures, we may ask how an execution environment can be obtained from such a sequence. The answer is that during the construction of an executable module, the TOOL environment asks the user to name the "main class" in the program: a class in the sequence possessing a method called "Main". The execution starts with the application of this main method to an object of the main class, automatically created by the system.

3. INTERFACE DRIVEN PROGRAMS: A CASE STUDY

An interface driven program, or a program controlled by a user who activates routines by acting on graphical interface objects, can usually be "separated" in two main parts, which we can call *interface* and *implementation*. These are communicating parts, of quite different natures. The interface part should contain all actions responsible for placing, in one or more screens, objects that the user sees and appreciates, there recognizing a representation of his/her real world. As examples of interface objects, we could mention control objects (buttons, menus), and application objects (text, geometric figures). On the other hand, the implementation is responsible for the "dirty work", that executed behind the screens (or windows ?), where private algorithms, acting on data structures unseen to the user, respond to the user's wishes.

The execution of an interface driven program usually starts out with the placement, on the screen, of the first user interface in the application program. In TOOL, when this construction terminates, the executable program rests, waiting for the user to start the festivities by activating some control object, say a button, thereby commanding some action.

When this action terminates, or even while this action is still being carried out, the user can affect some other control object on the screen, for example a menu option, to command another action.

The execution of a TOOL program can thus be seen as a "race" between the user, commanding actions by activating interface controls, and the implementation, carrying out the requested actions. At times, when the user is thinking out his/her next move, the system seems to be "at rest": some interface is still being shown, but the implementation may not be executing any actions (any visible actions, that is: maybe some garbage collection is taking place, but this is not the concern of users).

We can exemplify by considering a toy, but representative, application: the construction of a program to play TicTacToe. Here the interface part is responsible for placing, on a screen, a 3x3 matrix of buttons, where the game is to be played - each time the user plays, the program responds, and matrix positions are marked accordingly on the screen. This interface can be improved if we place on the screen a message area, where the turn is indicated, and where congratulations are offered to the winner (if it is the user). Besides, we may want to provide the user with controls to interrupt a losing game, and to elegantly exit the program. We may also consider the display of statistic data, and so on.

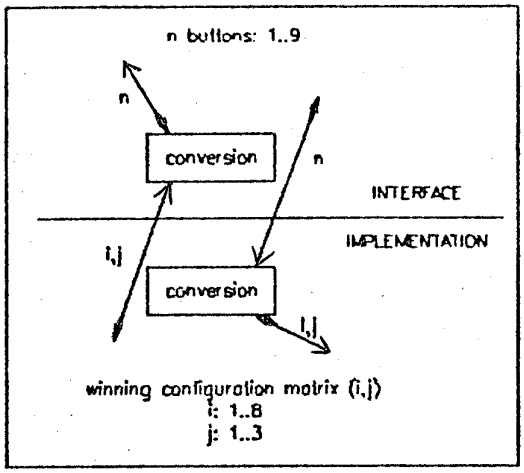
The nature of the implementation is completely different. In the accompanying TicTacToe source code (see the Appendix), the implementation contains all actions related to the winning strategy adopted. This is done by examining, at each system's turn, an 8x3 matrix used to represent all winning configurations, and which is updated each time a position is played.

In systems like these, interface and implementation must clearly correspond. In this example, the interface sends to the implementation the position played by the user, and receives back an answer, to display in the game matrix. Since the interface knows game buttons, and the implementation knows winning configuration positions, data must be transformed along the way. Here lies a crucial decision in the design of interface driven programs: how and where to convert data in the interface-implementation path. In the following paragraphs some solutions to this question are examined. The importance of this consideration lies in the fact that the higher the degree of coupling between interface and implementation, the smaller the possibility of reuse of any of these parts, and the higher the maintenance costs for the system.

Solution 1: mutual knowledge

One solution would be the construction, both in the interface and in the implementation, of special conversion routines, to be invoked when a change of worlds is about to happen. In this way one could increase the degree of independence between interface and implementation, allowing the connection of a single interface to several possible implementations, and vice-versa, at the reduced cost of changing only the conversion routines. The diagram below

illustrates this solution, which contains the highest degree of coupling among the ones to be presented:



In this solution, if an interface button must be converted to an i,j pair of implementation coordinates, then we would need an implementation conversion routine which receives a button n and produces the corresponding pair. Moreover, this routine would have to be invoked from within the interface code, which is executing when the need for this conversion arises. It follows that the interface would have to know the name of an object of the implementation class, the receiving object for this conversion routine. Symmetrically, the implementation must also know some interface object, which could then receive the application of some interface conversion routine that would discover a button from a pair of coordinates.

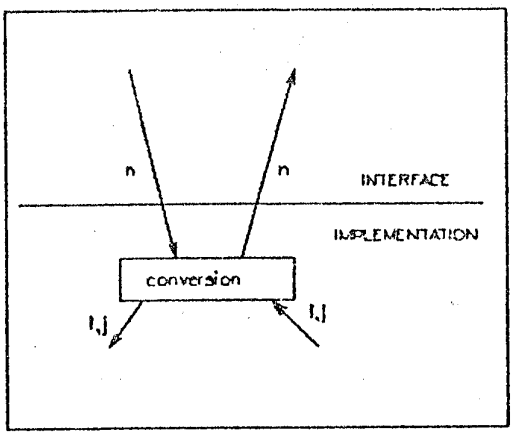
Unfortunately this symmetry in design is not well represented in programming languages (of any nature), including TOOL: the composition mechanism for the construction of object structures allows us to declare, for example, an implementation field in some interface class, making this field known to the interface; but we would not be able to declare the opposite situation as well.

We can solve this cross-referencing problem, and break up this need for symmetry, by choosing a preponderant part between interface and implementation. In the second solution that follows, this approach is discussed, and a corresponding TOOL program model is presented.

Solution 2: favored knowledge

In this second solution, the interface classes, perhaps by being in direct contact with the user, or by being the intermediaries between the user and the realization of their wishes, are favored with knowledge: they know the implementation, which they must activate to carry out

user wishes. The reverse is not true, however: the implementation does not know the interface it is working for. This asymmetric situation is illustrated in the diagram below.



In TOOL this specialization of knowledge would be acted out by two objects: the main interface class object, automatically created by the system, and an object of the main implementation class, created as a component of the interface class object, and therefore known to the interface. When the need for a button-coordinates conversion arises, the interface would apply to the known implementation object a conversion routine defined in the implementation. This routine would return to the interface the position of the button to be played by the system.

We argue that, even though the degree of decoupling has been increased in this solution, the fact that the interface object must know the name of the implementation object, and the fact that the implementation must know interface details to convert button to coordinates and vice-versa, still maintain the coupling between these parts at an unnecessarily high level (in the third solution, presented below, both of these needs disappear).

The TOOL program skeleton below illustrates this organization.

```
XCLASS MainInterface;

    REPRESENTATION  -- object structure.
        -- typically control objects needed to construct the interface.
        MainImplementation x;  -- the known implementation object.
    END REPRESENTATION

    METHOD Main;  -- execution starts here.
        ...
        x <- CREATE;  -- creates the implementation object.
        x <- SetUp;       -- or any such method,
                          -- prepares the implementation for execution.
        x <- ...
```

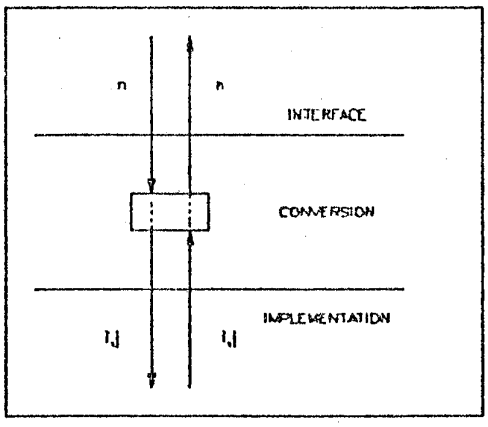
```
    END METHOD -- Main.

    -- other MainInterface operations, containing for example x <- Play (ub, sb);
    -- where ub is the user's button and sb the system's answer.

END XCLASS -- MainInterface.
-- other interface classes, if needed.


CLASS MainImplementation;
    ...
    METHOD SetUp;
        ...
    END METHOD -- SetUp.

    METHOD Play (IN Button ub, OUT Button sb);
        ...
    END METHOD -- Play.

    -- other MainImplementation operations.

END CLASS -- MainImplementation.
-- other implementation classes, if needed.
```

In the skeleton above the interface knows x, the implementation object it creates. It is then able to apply to x operations in x's class (the main implementation class), as for example x <- SetUp and x <- Play (ub, sb). The implementation, through x, merely responds to the interface object, without knowing its name (via parameter passing).

With this program organization the cross-referencing problem mentioned above has been solved, and the quality of the design has been improved, due to the concentration of all data-conversion routines in one part, the implementation part. However, the fact still remains that the implementation must know interface details in order to convert data. We conclude that concentrating the data-conversion burden in one part of the overall system would only marginally increase the degree of decoupling between interface and implementation.

Solution 3: conversion classes

Another solution, suggested by the inadequacies of the ones above, would be the construction of another set of classes, in fact conversion classes containing mainly routines to carry out domain changes. In this solution, whenever a conversion is needed, a routine in this class is invoked, and the proper data is passed on to the proper domain, as shown in the diagram below:



In this solution the need to know is concentrated in the conversion routines, the only ones to be changed when decoupling is to take place. This already increases the degree of independence between interface and implementation: neither needs to know the name and the operations of the other.

The construction of a program structure to reflect this solution, in a first-generation object-oriented language like Smalltalk or C++, would rely on return parameters placed in conversion routines: for example, after the conversion from button to coordinates, this conversion routine would expect, from the implementation, new coordinates to convert to the system's button. In modern object-oriented languages, able to code message passing as object behavior, however, an even higher decoupling can be obtained: both the interface and the implementation, after discovering, respectively, the button or the coordinates played, can send a message to their owner, an object of the conversion class, informing it of the option selected. This message can in fact be sent without knowing the name of the conversion object, as shown in the TOOL program skeleton below.

```
XCLASS DataConversion;  -- the main class in the application.

    REPRESENTATION
        Interface       f;  -- the main interface object.
        Implementation  m;  -- the main implementation object.
        ...
    END REPRESENTATION

    METHOD Main;  -- execution starts here.
        -- local declarations, if needed.
    BEGIN
        f <- CREATE;    -- to establish ownership: the DataConversion object
        m <- CREATE;    -- created by the system receives messages from f and m.
```

```
            ...
            f <- Initialize;   -- to create an interface on the screen.
            m <- Initialize;   -- to create internal data structures.
            ...
        END METHOD -- Main.

        HANDLER FROM f FOR ButtonToCoord (Button  b);
            -- to trap the message ButtonToCoord sent by the interface.
        BEGIN
            -- find coordinates i,j from button b.
            m <- Play (i,j);  -- sends the corresponding pair to the implementation.
        END HANDLER  -- f:ButtonToCoord.

        HANDLER FROM m FOR CoordToButton (Pair  p);
            -- to trap the message CoordToButton sent by the implementation.
        BEGIN
            -- find button b from the pair p of coordinates.
            f <- Mark (b);  --sends the corresponding button to the interface.
        END HANDLER  -- m:CoordToButton.

        -- other methods and handlers, if needed.

END XCLASS  -- DataConversion.

XCLASS Interface;
        ...
        MESSAGE  -- to be sent to owner.
            ButtonToCoord (Button);
        END MESSAGE

        METHOD Initialize;
            ...
        END METHOD  -- Initialize.

        METHOD Mark (Button b);
            ...
        END METHOD -- Mark.

        -- other routines, if needed, containing the message sending statement
        -- OWNER <<- ButtonToCoord (b), issued when a user button is marked.

END XCLASS  -- Interface.

XCLASS Implementation;
        ...
        MESSAGE  -- to be sent to owner.
```

```
        CoordToButton (Pair);
END MESSAGE

METHOD Initialize;
        ...
END METHOD -- Initialize.

METHOD Play (Pair p);
        ...
END METHOD  -- Play.

-- other routines, if needed, containing the message sending statement
-- OWNER <<- CoordToButton (p), issued when a user position is played.

END XCLASS  -- Implementation.
```

In the next paragraphs the skeleton above is described. Execution starts, as mentioned before, with the creation, by the TOOL system, of an object of the Conversion class, indicated as the main class during program construction. We shall call this distinguished object sysobj. The structure of sysobj, as indicated in the declaration of the Conversion class, is a composition of an Interface object f and of an Implementation object t. Once sysobj is available, the Conversion method Main is applied to it, again by the TOOL system.

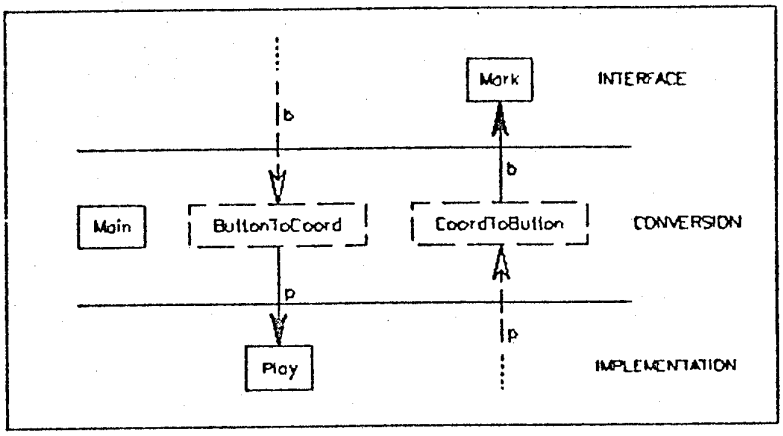
Within Main the objects f and t are created. This establishes ownership: sysobj will receive all messages sent by both f and t to their owners. Once f and t are created, they receive the application of initialization methods in the corresponding classes. These initialization methods typically display a first interface and prepare internal implementation data structures. Since this concludes the application of Main to sysobj, the program rests, waiting for the user to press a button, thus starting the game.

The choice of a button by the user generates actions that affect both the interface and the implementation. In the interface, this button must be marked as belonging to the user. Clearly the interface should be responsible for this action: it concerns its own internal structures, and no domain change is necessary. Once this marking occurs, however, implementation actions must be executed: the system must respond. At this time, using the generic message passing capabilities of TOOL, the interface object f sends a message to its owner (unknown to f), advising it as to the button played by the user. It should be noted that the only information sent along with the message is the button played, and that no result of this message is expected by the interface.

At this time the interface rests, having done its job. Eventually sysobj will receive this message, and the actions specified in the corresponding message handler will take place. Initially, the corresponding implementation coordinates are found; once they are available, the implementation object t receives the application of the method Play. Play receives a pair of user coordinates and produces a pair of system coordinates, corresponding to some (unknown to the implementation) button in the interface. Again the results produced are passed to the

generic owner via an asynchronous message, and not through parameter passing (more comments on this choice of communication mechanism in section 4 below).

At this point it is hopefully possible to see the communication pattern of the system: once either the interface or the implementation produces a result, a message is sent to their owner, passing this result; having translated the information received, the owner passes on adequate domain information. The diagram below illustrates this pattern.



4. Conclusions

In the long section above three solutions to the interface-implementation decoupling problem were presented. The decoupling level increased as we progressed along. Summarizing the solutions obtained, we can state the following:

- solution 1 is inadequate from all points of view, having in fact been presented to create atmosphere, so to speak.
- solution 2 is adequate, being the one basically adopted both in the MVC [KP 88, VPCCB 91] and in the ADV-ADT [CILS 92] methodologies. In it the client-server approach is clearly identified: in our example, the implementation serves its client interface by producing the button played by the system. However, the client (the interface) must know the name of the server (the implementation), and the server must know internal structures of the client.
- solution 3 presents a higher decoupling degree, since both the interface and the implementation serve a common client, a conversion object, without the need to know either its name or its internal details. This is possible with message passing and the use of a generic message receiver, namely the owner of the message sending objects. This totally decouples both the interface and the implementation from the system where included, thus allowing their insertion in other systems with greater ease.

We mentioned in the beginning of this report that a large body of programming systems involving object orientation, graphical user interfaces and message passing is expected to be

constructed in this decade. We have seen, considering the case study above, the advantage, from the design point of view, of dealing with message passing mechanisms at a higher level than that usually found in most object oriented systems today. And yet it seems only natural to consider the handling of messages as part of object behavior, and as a consequence to include in the proper programming unit, the class, operations to implement this behavior.

For suppose, in this next to last paragraph, that we had only methods available, with their usual semantics (that of procedures and/or functions). How could we decouple interface and implementation in such a programming model? The answer would have to be by trying to create, in the conversion class, methods to translate data between domains. But this would imply in a strict control enforced by the conversion class, with the corresponding saving of return addresses on the execution stack. Moreover, this would imply in the passing of parameters back and forth, increasing the coupling among conversion, interface and implementation. But finally, as a last argument, this would deny the very nature of interface driven programs: asynchronism is a built-in feature of such systems, being the natural participating way for their users.

We conclude with our favorite bumper-sticker:

TOOL PROGRAMMERS DO IT ASYNCHRONOUSLY.

BIBLIOGRAPHY

[CARV 92] Carvalho, S., "TOOL: a Short Description", Monografias em Ciência da Computação, n. 25/92, Departamento de Informática, PUC/RJ.

[CILS 92] Cowan, D., Ierusalimschy, R., Lucena, C., Stepien, T., "Abstract Data Views", Technical Report 92-07, University of Waterloo, Computer Science Department, Waterloo, Ontario, Feb 92.

[KP 88] Krasner, G., Pope, S., "A Cookbook for Using the Model View Controller User Interface Paradigm in Smalltalk 80", JOOP Aug-Sept 88, pp 26-49.

[LCIS 92] Lucena, C., Cowan, D. Ierusalimschy, R., Stepien, T., "Application Integration: Constructing Composite Applications from Interactive Components", Technical Report, University of Waterloo, Computer Science Department, Waterloo, Ontario, March 92.

[SPA 92] SPA Sistemas Planejamento e Análise, "TOOL: the Language", Apr 92.

[VPCCB 91] Vecchio, L., Pimenta, M., Cabral, R., Campos, I., Bonelli, A., "O Paradigma Model Pane Dispatcher sem Lágrimas", Relatório Técnico 017/91, Departamento de Ciência da Computação, UFMG, 1991.

APPENDIX

```
-------------------------------------------------------------------------
-- CONVERSION
-------------------------------------------------------------------------
XCLASS TTTConv;

    REPRESENTATION
        TTTInter    intf;
        TTTImpl     impl;
    END REPRESENTATION


    METHOD Main;
    BEGIN
        intf <- CREATE;
        intf <- SetUp;
        impl <- CREATE;
        impl <- SetUp;
    END METHOD -- Main.


    HANDLER FROM intf FOR ButtonToCoord ( SHORT pos );
        SHORT ulin, ucol;
    BEGIN
        CASE pos
            WHEN 1 THEN ulin := 1; ucol := 1;
            WHEN 2 THEN ulin := 1; ucol := 2;
            WHEN 3 THEN ulin := 1; ucol := 3;
            WHEN 4 THEN ulin := 2; ucol := 1;
            WHEN 5 THEN ulin := 2; ucol := 2;
            WHEN 6 THEN ulin := 2; ucol := 3;
            WHEN 7 THEN ulin := 3; ucol := 1;
            WHEN 8 THEN ulin := 3; ucol := 2;
            WHEN 9 THEN ulin := 3; ucol := 3;
        END CASE
        impl <- PlayButton ( ulin, ucol );
    END HANDLER  -- intf:ButtonToCoord.


    HANDLER FROM intf FOR ResetImpl;
    BEGIN
        impl <- Reset;
    END HANDLER -- intf:ResetImpl.
```

```
    HANDLER FROM impl FOR CoordToButton ( SHORT i, SHORT j, STRING m );
        SHORT pos;
    BEGIN
        CASE i
          WHEN 1 THEN pos := j;
          WHEN 2 THEN pos := 2 * i + j - 1;
          WHEN 3 THEN pos := 2 * i + j;
          ELSE pos := 0;
        END CASE
        intf <- MarkButton ( pos, m );
    END HANDLER -- impl:CoordToButton.

END XCLASS -- TTTConv.

-----------------------------------------------------------------
-- INTERFACE
-----------------------------------------------------------------
XCLASS GameButton;
    -- Models pushbuttons that know their positions.

    REPRESENTATION
        PushButton  pb;
        SHORT       pos;
    END REPRESENTATION

    MESSAGE
        GBClicked (SHORT);
    END MESSAGE

    HANDLER FROM pb FOR Clicked (SHORT e);
    BEGIN
        OWNER <<- GBClicked (pos);
    END HANDLER -- pb:Clicked.

    METHOD GBInitialize (IN SHORT x, IN SHORT y,
                         IN SHORT dx, IN SHORT dy,
                        ID Window POLY parent, IN SHORT p);
    BEGIN
          pb <- CREATE;
        pb <- PushButtonInitialize (x, y, dx, dy,
                    " ", TRUE, parent);
        pos := p;
    END METHOD -- GBInitialize.
```

```
        METHOD GBShow;
        BEGIN
                pb <- Show;
        END METHOD -- GBShow.

        METHOD GBGetText RETURNS STRING m;
        BEGIN
                m := pb <- GetText;
        END METHOD -- GBGetText.

        METHOD GBSetText (IN STRING m);
        BEGIN
                pb <- SetText (m);
        END METHOD -- GBSetText.

END XCLASS  -- GameButton.


XCLASS TTTInter;

        REPRESENTATION
                Window      POLY w;
                Brush               b;
                Push Button          new_game, end_game; -- reset and end game.
                Static              s;              -- for user messages.
                GameButton          b1, b2, b3,
                                    b4, b5, b6,
                                      b7, b8, b9;       -- the 3 * 3 matrix.
                SHORT               spos;           -- system's play position.
                STRING              m;              -- message to user.
        END REPRESENTATION

        MESSAGE
                ButtonToCoord ( SHORT );
                ResetImpl;
        END MESSAGE

        METHOD SetUp;
        BEGIN
                w <- CREATE;
                w <- Initialize( OVERLAPPED_WINDOW, "TicTacToe",
                                10, 10, 500, 300, NULL );
                b <- CREATE;
                b <- StockBrushInitialize( GRAY_BRUSH );
                w <- SetBackgroundBrush( b );
```

```
        s <- CREATE;
        s <- StaticInitialize( CHILD, CENTER, "",
                               20, 20, 200, 30, w );

        new_game <- CREATE;
        end_game <- CREATE;
        new_game <- PushButtonInitialize( 250, 20, 100, 30, "new game",
                                          TRUE, w );
        end_game <- PushButtonInitialize( 350, 20, 100, 30, "end game",
                                          TRUE, w );

        b1 <- CREATE;
        b1 <- GBInitialize (200, 100, 30, 30, w, 1);
        b2 <- CREATE;
        b2 <- GBInitialize (230, 100, 30, 30, w, 2);
        b3 <- CREATE;
        b3 <- GBInitialize (260, 100, 30, 30, w, 3);
        b4 <- CREATE;
        b4 <- GBInitialize (200, 130, 30, 30, w, 4);
        b5 <- CREATE;
        b5 <- GBInitialize (230, 130, 30, 30, w, 5);
        b6 <- CREATE;
        b6 <- GBInitialize (260, 130, 30, 30, w, 6);
        b7 <- CREATE;
        b7 <- GBInitialize (200, 160, 30, 30, w, 7);
        b8 <- CREATE;
        b8 <- GBInitialize (230, 160, 30, 30, w, 8);
        b9 <- CREATE;
        b9 <- GBInitialize (260, 160, 30, 30, w, 9);

        b1 <- GBShow;
        b2 <- GBShow;
        b3 <- GBShow;
        b4 <- GBShow;
        b5 <- GBShow;
        b6 <- GBShow;
        b7 <- GBShow;
        b8 <- GBShow;
        b9 <- GBShow;
        new_game <- Show;
        end_game <- Show;
        s <- Show;
        w <- Show;
END METHOD -- SetUp.
```

```
HANDLER FROM b1, b2, b3, b4, b5, b6, b7, b8, b9 \\
        FOR GBClicked (SHORT upos);
    GameButton gb;
BEGIN
    gb <- CREATE;

    -- identify game button:
    CASE upos
      WHEN 1 THEN gb <- SAME (b1);
      WHEN 2 THEN gb <- SAME (b2);
      WHEN 3 THEN gb <- SAME (b3);
      WHEN 4 THEN gb <- SAME (b4);
      WHEN 5 THEN gb <- SAME (b5);
      WHEN 6 THEN gb <- SAME (b6);
      WHEN 7 THEN gb <- SAME (b7);
      WHEN 8 THEN gb <- SAME (b8);
      WHEN 9 THEN gb <- SAME (b9);
    END CASE

    -- if not played before, mark and play this button:
    IF (gb <- GBGetText) <> ""
      THEN s <- SetText ("Position played. Try again...");
    ELSE
        s <- SetText ("Please wait. My turn...");
      gb <- GBSetText ("U");

      -- advise owner, requesting conversion:
      OWNER <<- ButtonToCoord ( upos );
    END IF
END HANDLER  -- b1...b9:GBClicked.

HANDLER FROM new_game FOR Clicked( SHORT enum );
BEGIN
    s <- SetText( "New game.  Your turn..." );
    b1 <- GBSetText ("");
    b2 <- GBSetText ("");
    b3 <- GBSetText ("");
    b4 <- GBSetText ("");
    b5 <- GBSetText ("");
    b6 <- GBSetText ("");
    b7 <- GBSetText ("");
    b8 <- GBSetText ("");
    b9 <- GBSetText ("");
    OWNER <<- ResetImpl;
END HANDLER -- new_game:Clicked.
```

```
    HANDLER FROM end_game FOR Clicked( SHORT enum );
    BEGIN
        SELF.s <- SetText ("Thank you.");
        b1 <- GBSetText ("");
        b2 <- GBSetText ("");
        b3 <- GBSetText ("");
        b4 <- GBSetText ("");
        b5 <- GBSetText ("");
        b6 <- GBSetText ("");
        b7 <- GBSetText ("");
        b8 <- GBSetText ("");
        b9 <- GBSetText ("");
       OWNER <<- ResetImpl;
    END HANDLER -- end_game:Clicked.

    METHOD MarkButton (IN SHORT spos, IN STRING m);
    BEGIN
        IF spos <> 0 THEN  -- mark system button:
          CASE spos
            WHEN 1 THEN b1 <- GBSetText("S");
            WHEN 2 THEN b2 <- GBSetText("S");
            WHEN 3 THEN b3 <- GBSetText("S");
            WHEN 4 THEN b4 <- GBSetText("S");
            WHEN 5 THEN b5 <- GBSetText("S");
            WHEN 6 THEN b6 <- GBSetText("S");
              WHEN 7 THEN b7 <- GBSetText("S");
            WHEN 8 THEN b8 <- GBSetText("S");
            WHEN 9 THEN b9 <- GBSetText("S");
          END CASE
        END IF
        -- send message to user:
        s <- SetText (m);
    END METHOD  -- MarkButton.

END XCLASS -- TTTInter.

-------------------------------------------------------------
-- IMPLEMENTATION
-------------------------------------------------------------
CLASS Cell;
    PUBLIC REPRESENTATION
        SHORT pos, nextl := 0, nextc := 0,
                firstl, firstc;
    END REPRESENTATION
END CLASS -- Cell.
```

```
XCLASS TTTImpl;

    REPRESENTATION
        ARRAY game OF [8, 3] Cell; -- winning combinations.
        ARRAY mark OF [8, 3] CHAR; -- positions played.
    END REPRESENTATION

    MESSAGE
        CoordToButton ( SHORT, SHORT, STRING );
    END MESSAGE

    METHOD Reset;
        SHORT l := 1, c := 1;
    BEGIN
        LOOP
            LOOP
                mark [l, c] := ' ';
                EXIT WHEN c = 3;
                c := c + 1;
            END LOOP
            EXIT WHEN l = 8;
            l := l + 1;
            c := 1;
        END LOOP
    END METHOD -- Reset.

    METHOD SetUp;
    BEGIN
        SELF <- Reset;
        game[1,1].pos:=1; game[1,1].nextl:=4; game[1,1].nextc:=1;
            game[1,1].firstl:=1; game[1,1].firstc:=1;
        game[1,2].pos:=2; game[1,2].nextl:=5; game[1,2].nextc:=1;
            game[1,2].firstl:=1; game[1,2].firstc:=2;
        game[1,3].pos:=3; game[1,3].nextl:=6; game[1,3].nextc:=1;
            game[1,3].firstl:=1; game[1,3].firstc:=3;
        game[2,1].pos:=4; game[2,1].nextl:=4; game[2,1].nextc:=2;
            game[2,1].firstl:=2; game[2,1].firstc:=1;
        game[2,2].pos:=5; game[2,2].nextl:=5; game[2,2].nextc:=2;
            game[2,2].firstl:=2; game[2,2].firstc:=2;
        game[2,3].pos:=6; game[2,3].nextl:=6; game[2,3].nextc:=2;
            game[2,3].firstl:=2; game[2,3].firstc:=3;
        game[3,1].pos:=7; game[3,1].nextl:=4; game[3,1].nextc:=3;
            game[3,1].firstl:=3; game[3,1].firstc:=1;
        game[3,2].pos:=8; game[3,2].nextl:=5; game[3,2].nextc:=3;
```

```
        game[3,2].firstl:=3; game[3,2].firstc:=2;
    game[3,3].pos:=9; game[3,3].nextl:=6; game[3,3].nextc:=3;
        game[3,3].firstl:=3; game[3,3].firstc:=3;
    game[4,1].pos:=1; game[4,1].nextl:=7; game[4,1].nextc:=1;
        game[4,1].firstl:=1; game[4,1].firstc:=1;
    game[4,2].pos:=4; game[4,2].nextl:=0; game[4,2].nextc:=0;
        game[4,2].firstl:=2; game[4,2].firstc:=1;
    game[4,3].pos:=7; game[4,3].nextl:=8; game[4,3].nextc:=3;
        game[4,3].firstl:=3; game[4,3].firstc:=1;
    game[5,1].pos:=2; game[5,1].nextl:=0; game[5,1].nextc:=0;
        game[5,1].firstl:=1; game[5,1].firstc:=2;
    game[5,2].pos:=5; game[5,2].nextl:=7; game[5,2].nextc:=2;
        game[5,2].firstl:=2; game[5,2].firstc:=2;
    game[5,3].pos:=8; game[5,3].nextl:=0; game[5,3].nextc:=0;
        game[5,3].firstl:=3; game[5,3].firstc:=2;
    game[6,1].pos:=3; game[6,1].nextl:=8; game[6,1].nextc:=1;
        game[6,1].firstl:=1; game[6,1].firstc:=3;
    game[6,2].pos:=6; game[6,2].nextl:=0; game[6,2].nextc:=0;
        game[6,2].firstl:=2; game[6,2].firstc:=3;
    game[6,3].pos:=9; game[6,3].nextl:=7; game[6,3].nextc:=3;
        game[6,3].firstl:=3; game[6,3].firstc:=3;
    game[7,1].pos:=1; game[7,1].nextl:=0; game[7,1].nextc:=0;
        game[7,1].firstl:=1; game[7,1].firstc:=1;
    game[7,2].pos:=5; game[7,2].nextl:=8; game[7,2].nextc:=2;
        game[7,2].firstl:=2; game[7,2].firstc:=2;
    game[7,3].pos:=9; game[7,3].nextl:=0; game[7,3].nextc:=0;
        game[7,3].firstl:=3; game[7,3].firstc:=3;
    game[8,1].pos:=3; game[8,1].nextl:=0; game[8,1].nextc:=0;
        game[8,1].firstl:=1; game[8,1].firstc:=3;
    game[8,2].pos:=5; game[8,2].nextl:=0; game[8,2].nextc:=0;
        game[8,2].firstl:=2; game[8,2].firstc:=2;
    game[8,3].pos:=7; game[8,3].nextl:=0; game[8,3].nextc:=0;
        game[8,3].firstl:=3; game[8,3].firstc:=1;
END METHOD -- SetUp.

METHOD PlayButton (IN SHORT ulin, IN SHORT ucol);
        SHORT   slin, scol;  -- system coords.
        STRING  s;
BEGIN
        -- record user's choice:
        SELF <- RecordChoice (ulin, ucol, 'U');
        -- test user's win:
        IF SELF <- Win ('U') THEN
            OWNER <<- CoordToButton (0, 0,
                        "You win; congratulations!");
            RETURN;
```

```
    END IF
      -- find system position:
    SELF <- FindSysPos (slin, scol);
      -- game over, it's a tie:
    IF slin = 0  THEN
          OWNER <<- CoordToButton (0, 0,
                        "It's a tie. Choose new or end game.");
          RETURN;
    END IF
    -- record system position:
    SELF <- RecordChoice (slin, scol, 'S');
    IF SELF <- Win ('S') THEN
          s := "System wins. Choose new or end game.";
    ELSE
          s := "Your turn ...";
    END IF
    OWNER <<- CoordToButton (slin, scol, s);
END METHOD -- PlayButton.

PRIVATE METHOD RecordChoice \\
          (IN SHORT lin, IN SHORT col, IN CHAR player);
        SHORT l, c;  -- local line, column.
        SHORT k;     -- temp for line.
BEGIN
        l := lin; c := col;
        -- record choice in all configurations:
        LOOP
              mark[l, c] := player;
              -- find next configuration:
              k := game[l, c].nextl;
              RETURN WHEN k = 0;
              c := game[l, c].nextc;
              l := k;
        END LOOP
END METHOD --RecordChoice.

PRIVATE METHOD Win (IN CHAR player) RETURNS BOOLEAN w;
        SHORT l := 1;
BEGIN
        w := FALSE;
        LOOP
              IF mark[l, 1] = player AND \\
                 mark[l, 2] = player AND \\
                 mark[l, 3] = player
                THEN w := TRUE;  RETURN;
              END IF
```

```
            RETURN WHEN l = 8;
            l := l + 1;
        END LOOP
END METHOD -- Win.

PRIVATE METHOD FindSysPos (OUT SHORT slin, OUT SHORT scol);
BEGIN
        slin := 0; scol := 0;
        -- try winning config:
        SELF <- FindConfig ('S', 'S', ' ', slin, scol);
        RETURN WHEN slin <> 0;
        -- avoid losing:
        SELF <- FindConfig ('U', 'U', ' ', slin, scol);
        RETURN WHEN slin <> 0;
        -- try to win:
        SELF <- FindConfig ('S', ' ', ' ', slin, scol);
        RETURN WHEN slin <> 0;
        -- find any position:
        slin := 1;
        LOOP
            scol := 1;
            LOOP
                RETURN WHEN mark[slin, scol] = ' ';
                EXIT WHEN scol = 3;
                scol := scol + 1;
            END LOOP
            EXIT WHEN slin = 8;
            slin := slin + 1;
        END LOOP
        -- game over, all positions played:
        slin := 0; scol := 0;
END METHOD -- FindSysPos.

PRIVATE METHOD FindConfig (IN CHAR c1, IN CHAR c2, IN CHAR c3,
                           OUT SHORT slin, OUT SHORT scol);
        SHORT l;  -- local line.
BEGIN
        l := 1;
        LOOP
            IF (mark[l,1] = c1 AND \\
                mark[l,2] = c2 AND \\
                mark[l,3] = c3) OR \\
                (mark[l,1] = c2 AND \\
                mark[l,2] = c1 AND \\
                mark[l,3] = c3)
            THEN slin := game[l,3].firstl;
```

```
                scol := game[l,3].firstc;
                RETURN;
            END IF
            IF (mark[l,1] = c1 AND \\
                mark[l,2] = c3 AND \\
                mark[l,3] = c2) OR \\
               (mark[l,1] = c2 AND \\
                mark[l,2] = c3 AND \\
                mark[l,3] = c1)
            THEN slin := game[l,2].firstl;
                scol := game[l,2].firstc;
                RETURN;
            END IF
            IF (mark[l,1] = c3 AND \\
                mark[l,2] = c2 AND \\
               mark[l,3] = c1) OR \\
              (mark[l,1] = c3 AND \\
               mark[l,2] = c1 AND \\
               mark[l,3] = c2)
            THEN slin := game[l,1].firstl;
                scol := game[l,1].firstc;
                RETURN;
            END IF
            RETURN WHEN l = 8;
            l := l + 1;
        END LOOP
        slin := 0; scol := 0;
    END METHOD -- FindConfig.

END XCLASS -- TTTImpl.
```