# Practica Profesional - User Interfaces (I)

**Copyright (c) Gabriel Pimentel, 2010**

All rights reserved. Without limiting the rights under copyright reserved above, no part of this publication may be reproduced, stored or introduced into a retrieval system, or transmitted, in any form or by any means (electronic, mechanical, photocopying, recording or otherwise), without the prior written permission of the copyright owner.

# Interfaz de Usuario (I)

## Introducción

En general las herramientas de desarrollo rápido de aplicaciones, conocidas como RAD, no promueven a nivel del desarrollador, ningún enfoque particular para el desarrollo de aplicaciones interactivas. Mas aun en muchos casos la arquitectura sugerida para este tipo de aplicaciones esta lejos de ser ideal.

Por tanto resulta necesario analizar que mecanismos se requieren poner en juego por parte de quienes desarrollan esta aplicaciones a efectos de promover tanto el reuso como la manipulación concurrente de ciertas abstracciones del dominio.

## Interfaz de Programación, Interfaz de Usuario

Las interfaces de usuario son diferentes de las interfaces que hemos estudiado hasta aquí conocidas genéricamente cómo interfaces de programación. Las interfaces de usuario son las interfaces a través de las cuales los usuarios, no los desarrolladores, interactúan con las abstracciones del dominio.

Muchos de los enfoques seguidos por las herramientas de desarrollo rápido no soportan (si habilitan) ni la reutilizacion de las abstracciones del dominio y mucho menos aun de aquellas abstracciones diseñadas para resolver la interacción con los usuarios.

Independientemente de esto, a lo largo del tiempo se plantearon diferentes modelos de interacción como solución a la necesidad, cada vez mas recurrente, de reutilizar las abstracciones del dominio entre aplicaciones con diferentes mecanismos de interacción.

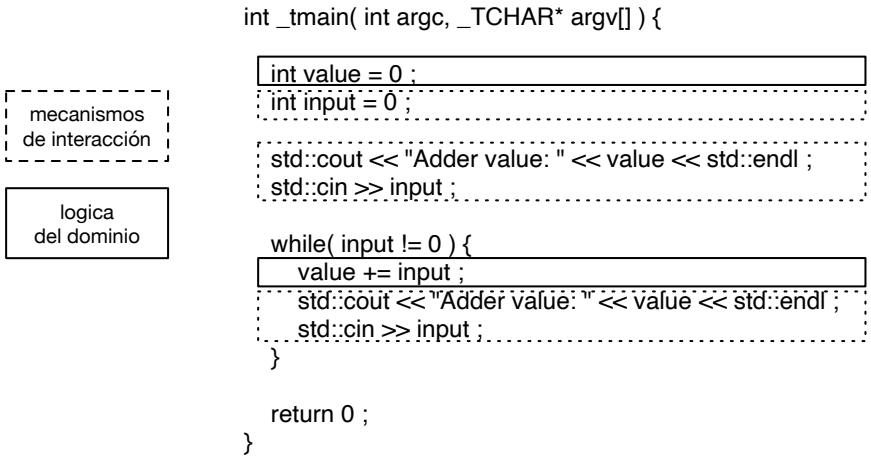
Todas estas propuestas promueven la modularización, no solo, de la lógica propia de la aplicación sino ademas, de la interfaz de usuario de modo que tanto las abstracciones que soportan los mecanismos de interacción como aquellas que modelan el dominio, puedan ser modificadas, extendidas y reutilizadas con facilidad.

A efectos de evaluar las decisiones de diseño que condujeron a los modelos de interacción actuales, comenzaremos analizando la modularizacion de una sencilla aplicación interactiva y sus sucesivos refinamientos. La aplicación a considerar trata de un sumador (Adder) que a partir de un valor inicial, acumula la suma de los valores ingresados por el usuario.

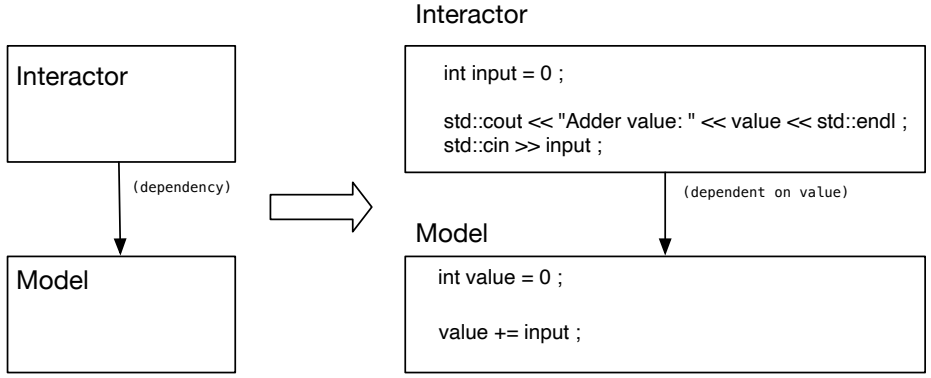
El fragmento de código que se muestra a continuación, representa una de las posibles realizaciones de lo especificado precedentemente. Cómo puede observarse inicialmente se muestra el valor inicial del sumador, luego se solicita el ingreso de un valor para ser sumado, y por ultimo se muestra el valor del sumador.

```
int _tmain( int argc, _TCHAR* argv[] ) {

    int value = 0 ;
    int input = 0 ;

    std::cout << "Adder value: " << value << std::endl ;
    std::cin >> input ;

    while( input != 0 ) {
        value += input ;
        std::cout << "Adder value: " << value << std::endl ;
        std::cin >> input ;
    }

    return 0 ;
}
```

Si bien esta versión cumple con la especificación inicial esta lejos de ser una solución que favorezca el reuso. En el fragmento de código anterior, es fácil observar qué ciertas lineas del mismo se corresponden con la interfaz de usuario (linea punteada) mientras que otras tienen que ver con la implementación de la lógica del dominio de la aplicacion (linea llena), como se indica a continuación.

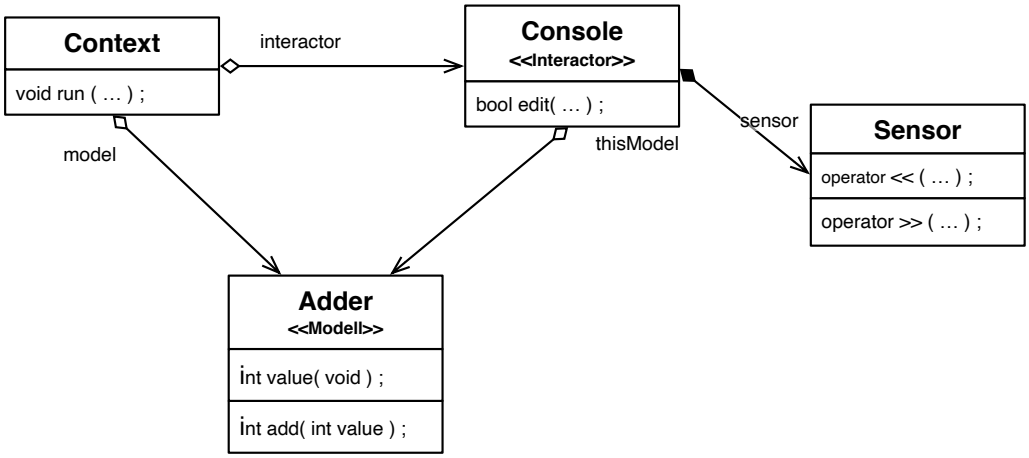



Visto esto, resulta lógico mantener el código correspondiente a la interfaz separado de aquel que implementa la lógica de la aplicación. Por aplicación del principio de separación de intereses, propuesto por Dijsktra, podemos modelar básicamente dos abstracciones, como se muestra a continuación.

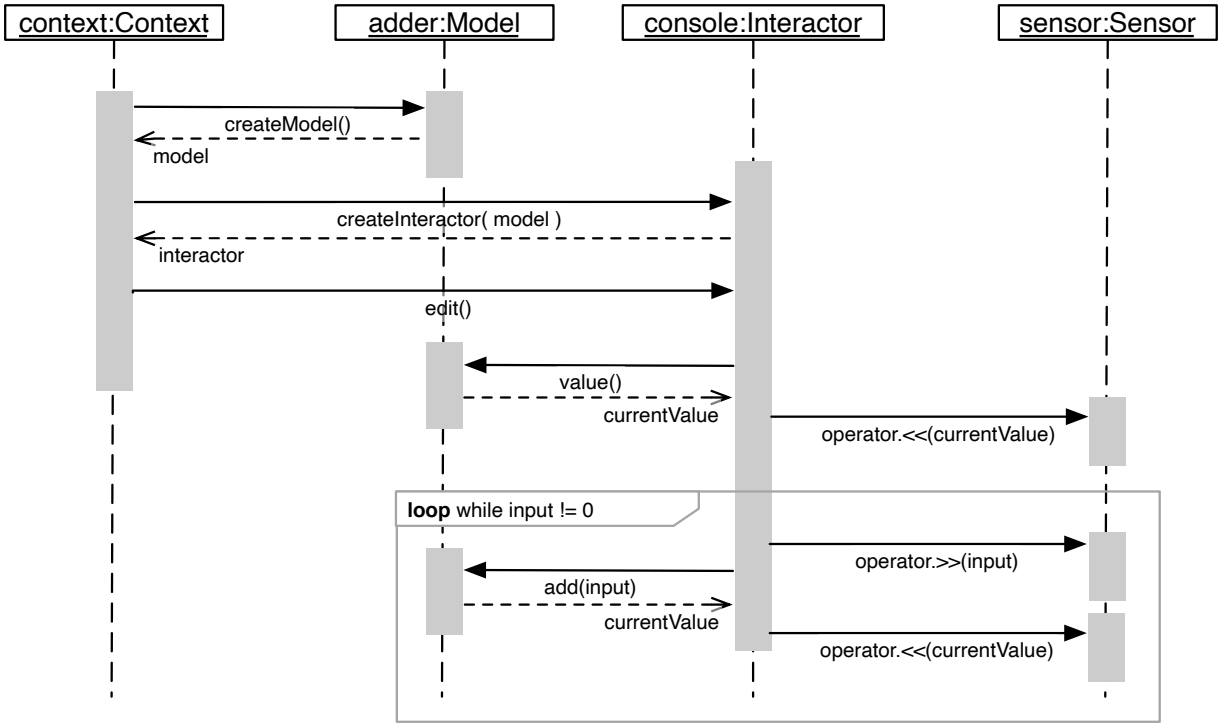


Una cuyo propósito es modelar los mecanismos de interacción, que denominaremos "Interactor", otra que modela la lógica propia de la aplicación "Model".

El diagrama de clase que se muestra a continuación, formaliza el diagrama anterior y agrega una abstracción (podrían ser dos, en este punto no es significativo) que encapsula los mecanismos de entrada/salida provistos por la plataforma (en nuestro caso el iostream), mostrando ademas el esquema de relaciones entre las abstracciones que hemos planteado hasta aquí. :




El diagrama de secuencia que se presenta a continuación, plantea la dinámica de la estructura propuesta:



Señalemos que a este nivel (iteración 1) y a efectos de facilitar la comprensión de los criterios de diseño que guiaron la evolución de los modelos de interacción, decidimos plantear la solución en termino de tipos concretos.

De acuerdo a los modelos presentados, podemos reformular  el código correspondiente a la aplicación de la siguiente manera.

En primer lugar consideremos el modelo, en nuestro caso el sumador (Adder), como planteamos al inicio de esta discusión, este tipo concreto expone una interfaz de programación simple. Por una lado permite sumar el valor ingresado por el usuario y por otro obtener el valor actual del sumador (acumulado), tal como puede observarse en el fragmento de código que se presenta a continuación:

```
class Adder {
public:
    Adder( int value = 0 ): itsValue( value ) {
    }

    virtual ~Adder( void ) {
    }

    int value( void ) {
        return itsValue ;
    }

    int add( int value ) {
        itsValue += value ;
    }
private:
    int itsValue ;

    Adder( const Adder& ) ;
    Adder& operator =( const Adder& ) ;
} ;
```

A priori, como se observa el sumador (Adder) no muestra ninguna dependencia con los mecanismos de interacción, por tanto podría ser reutilizado en el contexto de alguna otra interfaz gráfico u otra aplicación.

En segundo lugar consideremos los mecanismos de interacción, en nuestro ejemplo el tipo concreto “Console”. El fragmento de código que se muestra a continuación, refleja la implementación de la interfaz gráfica de acuerdo al modelo que venimos tratando, en nuestro caso los mecanismos de interacción para nuestra aplicación se instancia a través de la consola.

Cómo muestran los modelos presentados existe una dependencia entre la consola y el sumador, en este punto del proceso hemos elegido resolver esta dependencia de manera estática, vía el constructor de la consola. Obviamente este enfoque esta lejos de ser la solución mas flexible, pero como plantearemos anteriormente resulta conveniente en este punto del análisis.

```
// forward declaration ...
class Adder ;

class Console {
public:
    Console( Adder* model ): thisModel( model ) {
    }

    virtual ~Console( void ) {
    }

    bool edit( void ) {
        int input = 0 ;
        std::cout << "Adder value: " << thisModel->value() << std::endl() ;
    looper:
        std::cin >> input ;
        if ( input != 0 ) {
            std::cout << "" << thisModel->add( input ) << std::endl ;
            goto looper ;
        }
    }
private:
    Adder* thisModel ;

    Console( const Console& ) ;
    Console& operator =( const Console& ) ;
} ;
```

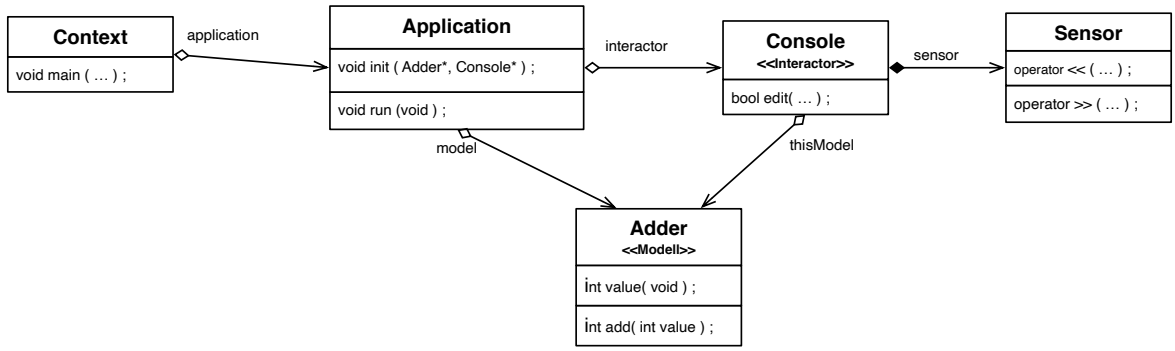
Por ultimo, en virtud de los tipos introducidos, mostramos a continuación la estructura de nuestra aplicación:

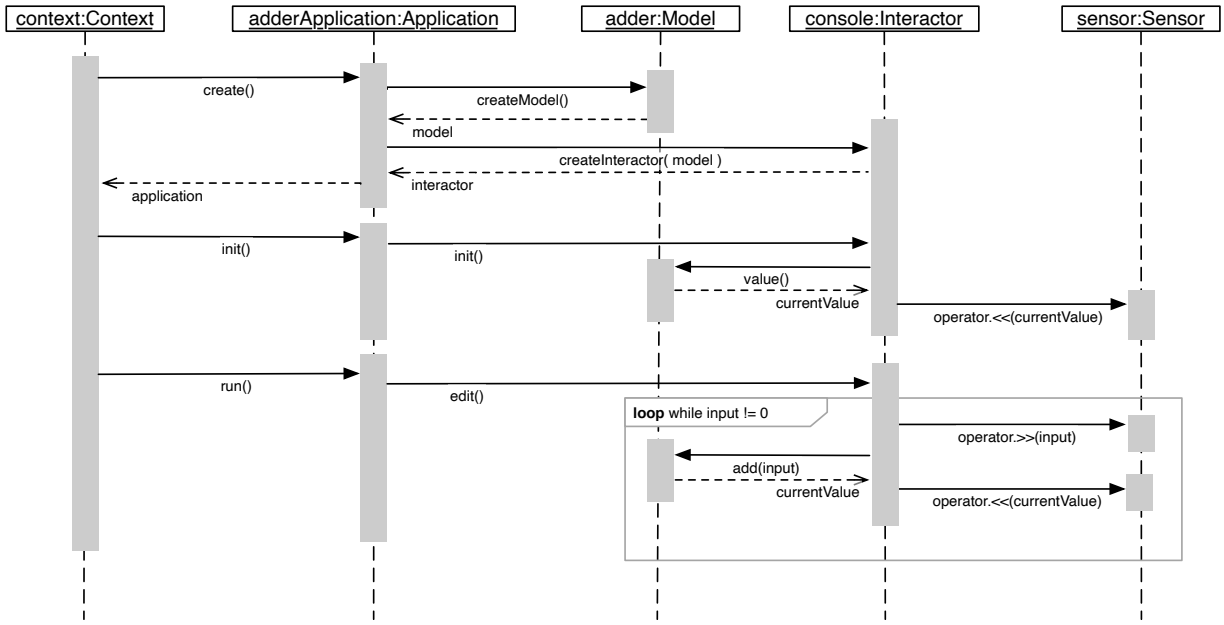
```
    int _tmain( int argc, _TCHAR* argv[] ) {

        // acquire resources ...
        Adder* model = new Adder ;
        Console* console = new console( model ) ;

        console->edit() ;

        //
        // release resources ...
        //

        return 0 ;
    }
```

Si bien aceptamos que el código anterior puede ser interpretado como nuestra aplicación, formalmente, no sé introdujo hasta aquí una abstracción que represente el tipo de aplicaciones interactivas de las que venimos hablando.

Introducir esta nueva abstracción requiere, en principio, modificar los modelos planteados con anterioridad, tanto la estructura como la dinámica se ven impactados por este cambio, lo interesante aquí es comenzar a pensar acerca de la magnitud del impacto.

A continuación puede observarse los nuevos diagramas de estructura y dinámica que dan cuenta de la nueva abstracción:




Cómo puede observarse el impacto de este refinamiento, no resulta demasiado significativo, teniendo en cuenta que hasta aquí hemos estado tratando con tipos concretos.

El fragmento de código que se presenta a continuación representa a la aplicación, puede observar la presencia de algunos tipos de dependencias severas, cuya resolución dejaremos para próximas iteraciones :

```
class AdderApplication {
public:
    AdderApplication( void ) {
        model = new Adder() ;
        interactor = new Console( model ) ;
    }

    virtual ~AdderApplicarion( void ) {
        delete interactor ;
        delete model ;
    }

    void init( void ) {
        interactor->init() ;
    }

    void run( void ) {
        interactor->edit() ;
    }
private:
    Adder* model ;
    Console* interactor ;

    AdderApplication( const AdderApplication& ) ;
    AdderApplication& operator =( const AdderApplication& ) ;
} ;
```

Las modificaciones efectuadas también requieren modificar el código de la "Console" tal como se muestra en el siguiente fragmento de código :

```
// forward declaration ...
class Adder ;

class Console {
public:
    Console( Adder* model ): thisModel( model ) {
    }

    virtual ~Console( void ) {
    }

    void init( void ) {
       render() ;
    }

    bool edit( void ) {
        int input = 0 ;
    looper:
        std::cin >> input ;
        if ( input != 0 ) {
            render() ;
            goto looper ;
        }
    }
private:
    Adder* thisModel ;

    void render( void ) {
        std::cout << "" << thisModel->add( input ) << std::endl ;
    }

    Console( const Console& ) ;
    Console& operator =( const Console& ) ;
} ;
```

Con estas modificaciones, podemos replantear la función "main" (context), como se muestra a continuación:

```
int _tmain( int argc, _TCHAR* argv[] ) {

    AdderApplication* application = new AdderApplication() ;
    try {

        application->init() ;
        application->run()  ;

    } catch( … ) {
        ;
    }

    delete application ;


    return 0 ;
}
```

Mas allá de lo frágil de nuestro diseño, resulta significativa la similitud con el código generado para una aplicación interactiva por algunas herramientas de desarrollo rápido, tal como se muestra a continuación (codigo generado por el C++ Builder Community Edition):

```
int WINAPI _tWinMain( HINSTANCE, HINSTANCE, LPTSTR, int ) {

   try {

        Application->Initialize() ;
        Application->MainFormOnTaskBar = true ;
        Application->CreateForm( __classid( TForm1, &Form1 );
        Application->Run()  ;

    } catch( Exception& exception ) {
        Application->ShowException( exception ) ;
    }
    // otras lineas …

    return 0 ;
}
```

Si bien la arquitectura planteada se encuentra en un nivel totalmente concreto, su valor radica en que muestra, a pesar de su fragilidad, que la modularizacion de los mecanismos de interacción en una aplicación interactiva satisface las necesidades planteadas inicialmente. Obviamente, subsiguientes iteraciones, deberían concluir con el desarrollo de un framework (por el momento una colección de interfaces, clases abstractas y sus relaciones) para el desarrollo de aplicaciones interactivas dentro de este modelo de interacción.

```
 1: //
 2: //  User Iterface Models
 3: //  Iteration 1
 4: //
 5: //  UserInterfaceModelsContext_Iteration_1.cpp
 6: //
 7: //  Created by Gabriel Pimentel.
 8: //  Copyright (c) 2008, 2019 Gabriel Pimentel. All rights reserved.
 9: //
10: #include <vcl.h>
11: #include <windows.h>
12:
13: #pragma hdrstop
14: #pragma argsused
15:
16: #include <tchar.h>
17:
18: #include <stdio.h>
19: #include <conio.h>
20:
21: #include "concrete_interactor.h"
22:
23: int _tmain(int argc, _TCHAR* argv[])
24: {
25:     Adder* adder = new Adder( 10 ) ;
26:     Interactor* interactor = new Interactor( adder ) ;
27:
28:     interactor->edit() ;
29:
30:     delete interactor ;
31:     delete adder ;
32:
33:     if ( TEST_DESTRUCTOR ) getch() ;
34:     return 0;
35: }
```

```
//
//  User Iterface Models
//  Iteration 1
//
//  concrete_iterator.h
//
//  Created by Gabriel Pimentel.
//  Copyright (c) 2008, 2019 Gabriel Pimentel. All rights reserved.
//
#ifndef concrete_interactorH
#define concrete_interactorH

#include <iostream>
#include "concrete_adder.h"

class Interactor {
public:
    Interactor( Adder* model ): myModel( model ) {
    }

    virtual ~Interactor( void ) {
        if ( TEST_DESTRUCTOR ) std::cout << "Interactor::~Interactor( void )" << std::endl ;
    }

    void edit( void ) {
        int input = 0 ;
        std::cout << "Adder value: " << myModel->value() << std::endl ;
    looper:
        std::cin >> input ;
        if ( input != 0 ) {
            std::cout << "Adder value: " << myModel->add( input ) << std::endl ;
            goto looper ;
        }
    }

private:
    Adder* myModel ;

    Interactor( const Interactor& ) ;
    Interactor& operator =( const Interactor& ) ;
};


#endif
```

```
 1: //
 2: //  User Iterface Models
 3: //  Iteration 1
 4: //
 5: //  concrete_adder.h
 6: //
 7: //  Created by Gabriel Pimentel.
 8: //  Copyright (c) 2008, 2019 Gabriel Pimentel. All rights reserved.
 9: //
10: #ifndef concrete_adderH
11: #define concrete_adderH
12:
13: #include <iostream>
14:
15: #define TEST_DESTRUCTOR     1
16:
17: class Adder {
18: public:
19:     Adder( int value = 0 ): adderValue( value ) {
20:     }
21:
22:     virtual ~Adder( void ) {
23:         if ( TEST_DESTRUCTOR ) std::cout << "Adder::~Adder( void )" << std::endl ;
24:     }
25:
26:     int add( int value ) {
27:         adderValue += value ;
28:         return adderValue ;
29:     }
30:
31:     int value( void ) {
32:         return adderValue ;
33:     }
34: private:
35:     int adderValue ;
36:
37:     Adder( const Adder& ) ;
38:     Adder& operator =( const Adder& ) ;
39: };
40:
41:
42: #endif
```