Introducción a 2MIL
Lenguaje de Interconexión de Módulos

Laura Pomponio

Ingeniería de Software II
Facultad de Ciencias Exactas Ingeniería y Agrimensura
Universidad Nacional de Rosario

2025

Índice

1. Introducción

Diseñar software requiere documentar con claridad y sin ambigüedades, pero a la vez sin detalles respecto a su implementación. En [GJM03] los autores definen el lenguaje TDN (Textual Design Notation) para documentar de un modo estructurado y formal el diseño de software. 2MIL, propuesto por Maximiliano Crisitá en [Cri22], es es una adaptación de este lenguaje.

Este documento extiende y modifica algunas de las descripciones de 2MIL propuestas en [Cri22] y oficia como guía rápida para usarlo.

1.1. Módulos y sus interfaces

De acuerdo a [GJM03], un módulo es un fragmento de software bien definido que provee recursos o servicios computacionales a otras partes del sistema. Por tanto, los módulos interactúan entre sí a través de sus interfaces. Es decir; la *interfaz* de un módulo es el conjunto de recursos o servicios que éste provee a otros módulos clientes, es aquello que puede ser accedido desde fuera del módulo. Se dice entonces, que dichos recursos o servicios son exportados por el módulo e importados por los módulos clientes. La forma en la cual el módulo lleva a cabo aquello que exporta es el secreto del módulo y está oculto; esto es, está inmerso en su *implementación*.

En principio, un módulo puede exportar por medio de su interfaz cualquier tipo de recurso o servicio: una variable, un tipo, un método o cualquier otro tipo de elemento que pueda tener su correlato en un lenguaje de programación. Sin embargo—si se sigue la técnica de abstracción y encapsulamiento—es probable que la mayoría de los elementos de una interfaz sean métodos.

2MIL provee un modo claro y formal de describir las interfaces de los módulos de un sistema. Sin embargo, el lenguaje no provee una descripción formal para describir la semántica de los elementos de una interfaz. Es decir, para describir formalmente en qué consiste un recurso o cómo funciona un servicio provisto se debe recurrir a lenguajes de especificación formal—por ej. CSP, Statecharts, Z, TLA, etc. Lo que sí provee 2MIL es una sintaxis precisa para definir las interfaces de los módulos.

2. Algunas convenciones sugeridas

Se sugiere que el nombre de los módulos comience con mayúscula y que los métodos y las variables—si eventualmente estas aparecen—comiencen y se escriban mayormente con minúscula.

Un *módulo lógico* es un módulo que no se implementa, se define para agrupar porciones del sistema con fines organizativos. Los módulos lógicos se utilizan para describir la estructura de módulos de acuerdo a un agrupamiento de estos basado en cierto criterio.

Un *módulo físico* requiere implementación. Este módulo puede ser *abstracto* o *concreto*. En el primer caso será un módulo donde no todos sus métodos son implementados y por tanto no puede crearse una instancia de él. En el segundo caso, todos sus métodos son implementados y por tanto puede crearse una instancia del mismo.

Para mejor legibilidad, en las descripciones se usará distinta tipografía según el tipo de módulo que se presente en los ejemplos. Sin embargo, esto no pertenece a la definición del lenguaje.

Tipografía en el nombre	Descripción
NombreModulo	Módulo físico-concreto. Se implementa y puede crearse una instancia del mismo.
NombreModulo	Módulo físico-abstracto. Se implementa pero no puede crearse una instancia del mismo.
NOMBREMODULO	Módulo lógico. No se implementa

3. Métodos y sus parámetros formales

Un método en la interfaz de un módulo se especifica con el nombre y, entre paréntesis, los tipos de datos de sus parámetros, indicando además sus roles; es decir, cómo deben ser usados por el método. Un parámetro puede ser usado como únicamente de entrada, como únicamente de salida y como de entrada-salida. Para indicar estos se antepone al tipo del parámetro los siguientes indicadores: **i** (entrada), **o** (salida) e **io** (entrada-salida). Por ejemplo, **i** String declara un parámetro de entrada de tipo String. El significado de estos roles es el siguiente.

i (entrada) indica que el parámetro solo será leído por el método pero no será modificado.

o (salida) indica que el parámetro solo será modificado por el método pero no será leído.

io (entrada-salida) indica que el parámetro será leído y modificado por el método.

Si el método no tiene parámetros, los paréntesis quedarán vacíos. Si el método devuelve un valor, esto se indicará con dos puntos seguidos del tipo de dato de dicho valor.

Algunos ejemplos de cómo especificar un método son los siguientes.

- metodoA1() método llamado metodoA1 que no tiene parámetros ni devuelve un valor.
- metodoA2(): Boolean método llamado metodoA2 que no tiene parámetros pero devuelve un valor de tipo booleano.
- metodoA3(**i** String, **io** Boolean) método llamado metodoA3 que recibe como parámetro de entrada un string y como parámetro de entrada-salida un booleano. El último caso podía ser que el método recibe la referencia a una variable booleana y dependiendo de su valor (rol de entrada), modifica el valor de la misma (rol de salida).
- metodoA4(**o** String) método llamado metodoA4 no devuelve ningún valor y recibe un parámetro de tipo String que modificará.

Cuando se presente ambigüedad sobre a qué módulo pertenece cierto recurso y se quiera explicitar esto; se antepondrá al nombre del recurso, el nombre del módulo seguido de doble dos puntos (::) o de un punto (.). Por ejemplo, para explicitar que metodoA1 corresponde a la interfaz de *ModuloA*, se escribirá *ModuloA*::metodoA1() o *ModuloA*.metodoA1().

4. Cajas 2MIL para definir interfaces de módulos

La interfaz de un módulo se especifica con una caja 2MIL donde se indica el tipo del módulo, su nombre y cláusulas—o palabras reservadas—seguidas de sus valores correspondientes. Informalmente tiene la siguiente estructura.

```
<tipo-módulo>     <nombre-de-módulo y demás>
<cláusula>        <especificación>
   ⋮                    ⋮
<cláusula>        <especificación>
```

<tipo-módulo> representa una palabra reservada que identificará de qué tipo es el módulo para el cual se define la interfaz. Aunque se detallarán más adelante, los diferentes tipos de módulos se identifican con las palabras reservadas **Module** y **Generic Module**.

Debido a que la interfaz de un módulo es aquello visible y accesible desde fuera del módulo, la cláusula **exports** estará explicita o implícitamente en todas las definiciones las de una interfaz[1].

[1]Si un módulo no exporta nada, no existe interacción de este con el resto del sistema. Su interfaz es vacía y por tanto pierde sentido 2MIL.

En las secciones que continúan se describirá cómo especificar los siguientes tipos de interfaces de módulos.

- Interfaz de módulo simple o básico. Es físico (abstracto o concreto).
- Interfaz de módulo heredero. Es físico (abstracto o concreto) y hereda la interfaz de otro.
- Interfaz de módulo genérico. Es físico-abstracto.
- Interfaz de instancia de módulo genérico. Es físico-concreto.
- Interfaz de módulo lógico.

5. Cláusula **exports** - Interfaz de un módulo básico

La interfaz de un módulo definida en 2MIL requiere el nombre del módulo y los recursos o servicios accesibles desde fuera del módulo. Se presenta más abajo una caja 2MIL que define la interfaz de un módulo llamado *ModuloA*.

- Un módulo se identifica con la cláusula **Module** seguida del nombre del módulo (por ej. *ModuloA*).
- Para especificar cuáles son los recursos del módulo que son visibles al resto del sistema se usa la cláusula **exports** seguida de la lista de estos recursos, presentada en forma vertical (por ej. los métodos *metodoA1* y *metodoA2* de *ModuloA*).

```
Module      ModuloA

exports     metodoA1()
            metodoA2() : Boolean
```

6. Cláusula **imports** - Usar recursos/servicios de un módulo

Cuando un módulo usa recursos de otro módulo debe importar la interfaz de este último para poder usarla. Hay tres formas de importar recursos de un módulo: importar toda la interfaz en forma *explícita*, importar toda la interfaz en forma *implícita* o importar de forma *selectiva* solo ciertos recursos de la interfaz y dejar otros afuera.

6.1. **imports** explícito

Para declarar que un módulo usa recursos de otros módulos se utiliza la cláusula **imports** seguida de la lista de estos. Más abajo se define la interfaz de ModuloB y se especifica que este usa recursos del *ModuloA*.

- Cuando un módulo importa por medio de **imports** a otro, se está indicando que **todos** los recursos o servicios de la interfaz del último pueden ser accedidos por el primero.

```
Module      ModuloB

imports     ModuloA

exports     metodoB1()
            metodoB2()
            metodoB3():Real
```

En el ejemplo, ModuloB exporta tres métodos e importa mediante la cláusula **imports** todos los métodos de la interfaz de *ModuloA*—para usar uno o más de estos. Observar que si la cláusula **imports** no hubiera sido incluida, la definición de la interfaz de ModuloB no daría ningún indicio de que este módulo usa recursos de *ModuloA*. Esta cláusula es la que permite declarar esto.

6.2. imports implícito

Cuando en la interfaz de un módulo se declara un método—por medio de **exports**—con un argumento o un valor de salida que es del tipo de otro módulo, implícitamente se está indicando que toda la interfaz del último es importada.

- La aparición del nombre de un módulo en la interfaz de otro implica un **imports** implícito, donde el último puede acceder a los recursos del primero. En este caso no se requiere usar **imports**.

Observar abajo la definición de la interfaz de ModuloC. Este módulo exporta dos métodos. Uno que devuelve un valor de tipo *ModuloA* y otro que recibe como argumento de entrada un elemento de tipo ModuloB. La presencia de estos módulos en la interfaz de ModuloC indican que este último puede acceder a las interfaces de *ModuloA* y ModuloB. Por lo tanto, no se requiere usar la cláusula **imports** para indicar esto.

Module	ModuloC
exports	metodoC1():*ModuloA* metodoC2(**i** ModuloB)

6.3. imports selectivo

En algunos casos puede requerirse declarar que un módulo solo puede acceder a algunos de los recursos que exporta otro y no a toda la interfaz. En este caso, se especifica en la cláusula **imports** qué recursos específicamente se importan.

- Si a la cláusula **imports** se le asocia un subconjunto de los recursos de un módulo, se está declarando que únicamente esos recursos del módulo son importados y los otros no pueden ser accedidos.
- **imports** tiene mayor jerarquía que **exports** para determinar qué se importa. Esto significa que si un módulo importa selectivamente un recurso de otro módulo mediante **imports** y además el nombre de este último aparece en su interfaz; solo importará aquello declarado en **imports**.

Observar abajo la especificación del módulo ModuloD. Este módulo importa implícitamente todos los métodos de *ModuloA*, explícitamente todos los métodos de ModuloC pero solo importa un método de ModuloB aunque este sea un valor de retorno de metodoD2.

Module	ModuloD
imports	ModuloB::metodoB1, ModuloC
exports	metodoD1():*ModuloA* metodoD2():ModuloB

7. Cláusulas **protocol, private** y **comments** - Más información

Algunas cláusulas pueden agregarse a la definición de la interfaz, para documentar otras propiedades o características, que complementen la definición y ayuden en la comprensión del diseño. A continuación se presentan tres cláusulas.

- protocol se utiliza para describir un acuerdo que debe ser respetado por los módulos clientes para usar de forma adecuada los recursos de la interfaz. Por tanto, en el protocolo solo pueden aparecer los recursos exportados en la interfaz. Se sugiere usar una especificación formal aunque también esto podría ser expresado de modo semi-formal o informal.
- private permite declarar secretos del módulo. Esta cláusula no es necesaria para definir un módulo 2MIL—-ya que lo oculto no está en la interfaz—pero puede ser de ayuda para agregar cierta información que ayude a la comprensión del diseño.
- comments permite realizar algunos comentarios sobre el módulo. Los comentarios se expresan en lenguaje natural y son usados para proveer información semántica respecto de los servicios exportados por el módulo.

Se agregan a continuación estas cláusulas a ModuloB como ejemplo.

Module	ModuloB
imports	*ModuloA*
exports	metodoB1() metodoB2() metodoB3():Real
protocol	metodoB1 → **Module**
private	_estadoInterno : *ModuloA*
comments	Este es un módulo concreto. Todos los métodos de la interfaz son implementados. La variable _estadoInterno de tipo *ModuloA*, es privada y guarda el estado interno del módulo.

En este ejemplo, el protocolo es expresado en una extensión de CSP y especifica que para usar la interfaz, el cliente debe invocar metodoB1 antes que cualquier otro método del módulo.

En los exámenes no es necesario usar la cláusula protocol.

8. Cláusula **inherits from** - Heredar interfaz

Un módulo puede heredar la interfaz de otro módulo y agregar o no más recursos a su interfaz. Para especificar esto se usa la cláusula **inherits from**.

Se definen a continuación dos módulos como herederos de *ModuloA* (ver su definición más arriba o seguir el enlace).

Module	ModuloE **inherits from** *ModuloA*

ModuloE es un módulo que hereda la interfaz de *ModuloA* y por tanto exporta todos los métodos; esto es, ModuloE::metodoA1 y ModuloE::metodoA2 constituyen la interfaz.

Module	ModuloF **inherits from** *ModuloA*
exports	metodoF1(**i** String)

ModuloF es un módulo que hereda la interfaz de *ModuloA* y además define un método más en su interfaz. Por tanto, los métodos que exporta este módulo son ModuloF::metodoA1, ModuloF::metodoA2 y ModuloF::metodoF1.

9. Cláusula **Generic Module** - Interfaz de módulo genérico

Un módulo genérico es un módulo que puede ser parametrizado respecto a uno o varios tipos de datos. Estos módulos no pueden ser directamente usados por un cliente, son físicos-abstractos. Para que esto sea posible, primero deben ser instanciados con tipos de datos particulares, como se describe más adelante en sección 10.

Un ejemplo de módulo genérico es una cola o fila de elementos (FIFO). Cómo se almacena una estructura de cola y cómo se manipulan internamente sus elementos debería quedar oculto para los clientes detrás de una interfaz. Esta interfaz debería ser útil tanto para encolar String como para cualquier otro tipo de elemento, ya que la lógica del manejo de estos es la misma (encolar-desencolar).

La especificación de un módulo genérico se hace por medio de la cláusula **Generic Module** a la que se le asocia el nombre del módulo, seguido de una lista de parámetros genéricos separados por coma y entre paréntesis.

A continuación se presentan como ejemplo la interfaz de una cola y la de un par ordenado especificados en 2MIL.

Generic Module	*Cola(X)*
exports	encolar(X) desencolar() : X primero() : X vacia?() : Boolean

Esta interfaz permite que clientes pongan en la cola un elemento (encolar) y tomen el primero eliminándolo de la cola (desencolar). Además provee métodos para saber si la cola está vacía (vacia?) y acceder al primer elemento de la cola (primero). El tipo de dato manipulado por la cola es indicado con X como tipo genérico. Esto quiere decir que cuando X sea instanciado en un tipo particular, se tendrá una cola de dicho tipo.

Generic Module	*Par(X,Y)*
exports	comp1() : X comp2() : Y

La interfaz de *Par* permite el acceso a las componentes de un par ordenado, por medio de los métodos comp1 y comp2. Los tipos de datos de las componentes (X e Y) son genéricos. Como en el ejemplo anterior, cuando X e Y se instancien en tipos específicos, se contará con un módulo de pares ordenados de dichos tipos.

10. Cláusula is - Instancia de módulo genérico

Para definir un módulo concreto a partir de instanciar—en ciertos tipos de datos—un módulo genérico, se usa **Module** seguido del nombre del módulo y luego **is** seguido del nombre del módulo genérico donde sus argumentos fueron reemplazados por tipos de datos concretos.

Siguiendo los ejemplos presentados previamente se define abajo el módulo PtoPlano para definir un punto en el plano como un par ordenado *Par* de dos números reales.

Module	PtoPlano **is** *Par*(Real,Real)
comments	Especifica un punto en el plano.

PtoPlano es una instancia del módulo genérico *Par* donde los tipos genéricos X e Y fueron instanciados por Real en ambos casos. Con esta definición, módulos clientes podrían acceder a los métodos PtoPlano ::comp1() y PtoPlano ::comp2(), los cuales devolverían los valores reales que constituyen el par.

Otro ejemplo podría ser un registro de datos como se presenta a continuación. RegistroDato es un módulo que especifica un registro, de cierta información, como un par ordenado de un String y un elemento de tipo ModuloF.

Module	RegistroDato **is** *Par*(String,ModuloF)
comments	Define la interfaz de un registro de dos componentes que contienen información.

Continuando con el ejemplo, podría requerirse una cola de registros RegistroDato para poder ir procesándolos en orden. Para esto, se define la interfaz de dicha cola como sigue.

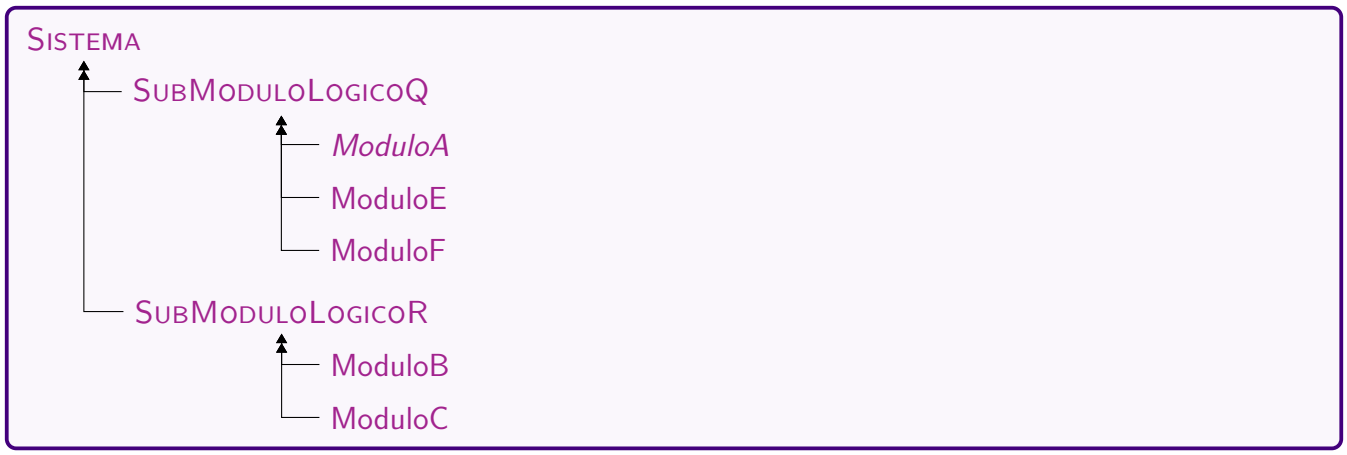
Module	ColaSolicitudes **is** *Cola*(RegistroDato)
comments	Define la interfaz de una cola de solicitudes a procesar.

ColaSolicitudes es una *Cola* de elementos de tipo RegistroDato. Por tanto, ColaSolicitudes::encolar(RegistroDato), como los otros métodos de *Cola* instanciados, serán parte de su interfaz.

11. Cláusula comprises - Organizar en módulos lógicos

Al descomponer un sistema en módulos es necesario establecer la estructura modular de un modo preciso. Para especificar la estructura de módulos definida en [PCW85, GJM03] es necesario describir módulos lógicos que agrupen—o estén compuestos por— submódulos lógicos y/o módulos físicos.

Imaginar que algunos de los módulos, que se definieron en los ejemplos previos, son organizados como muestra la siguiente estructura de módulos representada en árbol (↟ establece la relación de un módulo lógico y sus compuestos).



En esta estructura hay tres módulos lógicos que organizan módulos físicos: Sistema, SubModuloLogicoQ y SubModuloLogicoR.

La especificación de un módulo lógico en 2MIL se hace por medio de la cláusula **Module** seguida del nombre del módulo y luego **comprises** seguida de la lista, en forma vertical, de los submódulos lógicos y/o físicos que lo componen.

Al declarar un módulo lógico, se establece cuáles de los recursos de sus componentes pueden ser accedidos por fuera del mismo. En principio, la cláusula **comprises** indica que el módulo exporta todo lo que sus submódulos exportan. Como veremos, si se necesita restringir esto a solo un subconjunto de recursos se usa la cláusula **exports**.

La estructura de módulos definida más arriba se puede describir en 2MIL del siguiente modo.

Module	Sistema
comprises	SubModuloLogicoQ SubModuloLogicoR

Module	SubModuloLogicoQ
comprises	*ModuloA* ModuloE ModuloF

Module	SubModuloLogicoR
comprises	ModuloB ModuloC
exports	ModuloB::metodoB1

Considerar a continuación lo que puede ser accedido desde fuera de un módulo lógico, dependiendo de la especificación en 2MIL.

- Si un módulo lógico **no incluye** la cláusula **exports** indica que el módulo exporta **todos** los recursos exportados por sus submódulos.
- Si se quiere restringir el acceso desde afuera de un módulo lógico, a solo algunos recursos de los submódulos que este agrupa, entonces se debe usar la cláusula **exports** para explicitar el subconjunto de tales recursos. Cuando se usa **exports**, el módulo **únicamente** exportará aquello que se declare en la misma.

En el ejemplo, SubModuloLogicoQ—al no incluir la cláusula **exports**—exporta todos los recursos exportados por sus submódulos. Esto es, las interfaces de *ModuloA*, ModuloE, ModuloF. En el caso de SubModuloLogicoR solo exporta el método ModuloB::metodoB1. Notar que aunque ModuloC es parte de este módulo, como no aparece en **exports**, sus métodos no pueden ser accedidos por fuera de SubModuloLogicoR. Finalmente, el módulo principal Sistema, exporta todo lo que exporta SubModuloLogicoQ (interfaces de *ModuloA*, ModuloE, ModuloF) y todo lo que exporta SubModuloLogicoR (el método metodoB1 de ModuloB).

Referencias

[Cri22] M. Cristiá. *Diseño de Software*. Apunte de estudio. Ingeniería de Software II. LCC. UNR., 2022.

[GJM03] C. Ghezzi, M. Jazayeri, and D. Mandrioli. *Fundamentals of Software Engineering*. Prentice Hall, 2003.

[PCW85] D.L. Parnas, P.C. Clements, and D.M. Weiss. The modular structure of complex systems. *IEEE Transactions on Software Engineering*, SE-11(3):259–266, 1985.