

Ministerio de Producción

Secretaría de Industria y Servicios

Subsecretaría de Servicios Tecnológicos y Productivos

y

Ministerio de Educación y Deportes

A través del



Analistas del Conocimiento

Dimensión Programador

Apunte del Módulo

Programación Orientada a Objetos

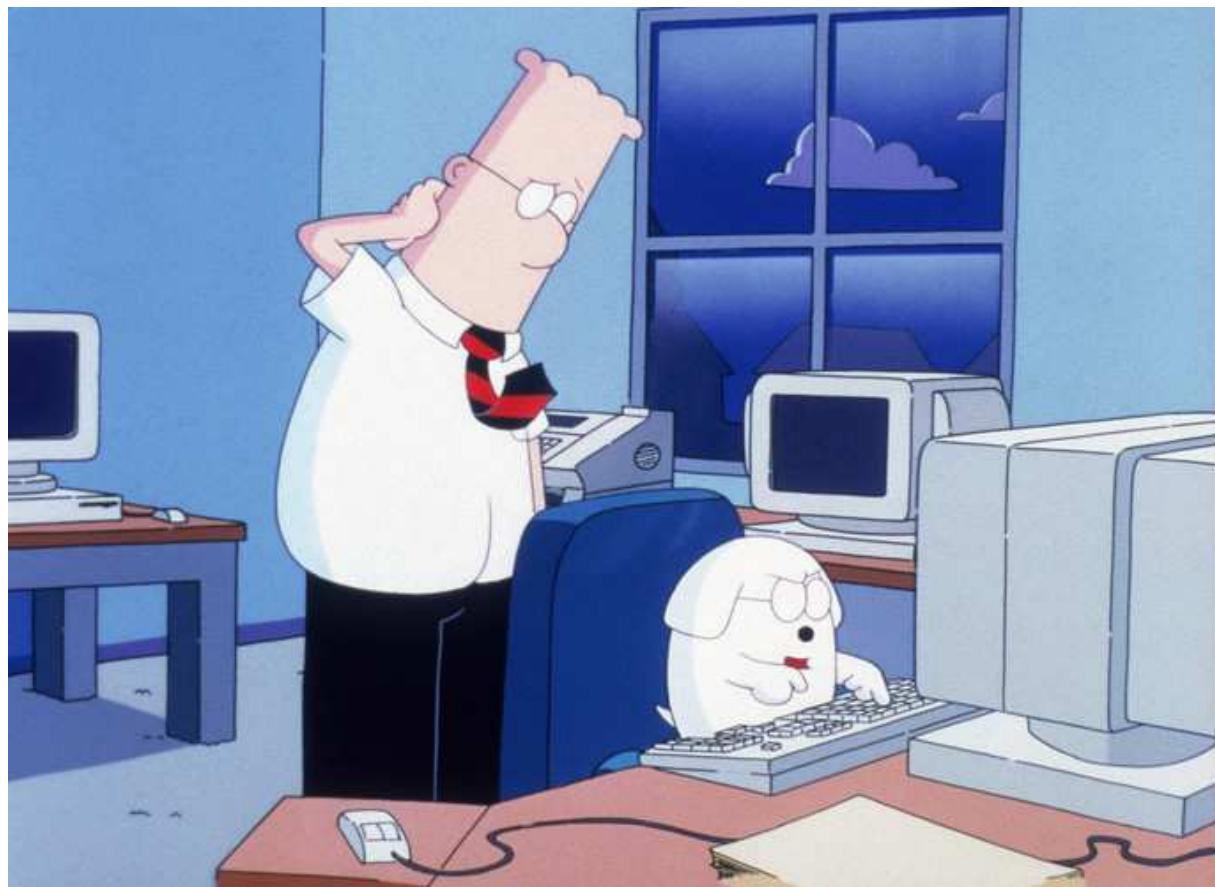


Tabla de Contenido

DEFINICIÓN DEL MÓDULO	6
PRESENTACIÓN	6
INTRODUCCIÓN	8
LENGUAJES DE PROGRAMACIÓN	10
TIPOS DE LENGUAJES DE PROGRAMACIÓN	15
NIVEL DE ABSTRACCIÓN DEL PROCESADOR	15
PARADIGMA DE PROGRAMACIÓN.....	17
FORMA DE EJECUCIÓN	17
¿QUÉ SON LOS PARADIGMAS?	19
PARADIGMAS DE PROGRAMACIÓN	19
CLASIFICACIÓN DE PARADIGMAS DE PROGRAMACIÓN	20
PARADIGMA IMPERATIVO	20
PARADIGMA DECLARATIVO	20
PARADIGMA ESTRUCTURADO	20
PARADIGMA FUNCIONAL	20
PARADIGMA LÓGICO	21
PARADIGMA ORIENTADO A OBJETOS.....	21
DESARROLLO DE SOFTWARE CON EL PARADIGMA ORIENTADO A OBJETOS.....	22
¿QUÉ ES UNA CLASE?.....	22
INTERFAZ E IMPLEMENTACIÓN.....	22
¿QUÉ ES UN OBJETO?	23
ESTADO	24
COMPORTAMIENTO.....	25
IDENTIDAD	26
RELACIONES ENTRE OBJETOS	26
ENLACES.....	26
VISIBILIDAD	27
AGREGACIÓN	27
RELACIONES ENTRE CLASES	27
ASOCIACIÓN.....	28
HERENCIA	28
EL ORNITORRINCO, QUE DERIVA DE MAMÍFERO Y OVÍPARO. LAS DOS CLASES DERIVAN DE ANIMAL.....	30
AGREGACIÓN.....	30
EL MODELO DE OBJETOS.....	31
ABSTRACCIÓN.....	31
ENCAPSULAMIENTO.....	33
MODULARIDAD.....	34
JERARQUÍA.....	36
TIPOS (TIPIFICACIÓN).....	38

CONCURRENCIA	39
PERSISTENCIA	40
CLASIFICACIÓN.....	41
<u>EL MODELADO EN EL DESARROLLO DE SOFTWARE</u>	<u>45</u>
¿QUÉ ES UN MODELO?	45
LA IMPORTANCIA DE MODELAR	45
PRINCIPIOS DE MODELADO	46
ADVERTENCIAS EN EL MODELADO	46
<u> LENGUAJE DE MODELADO UNIFICADO (UML)</u>	<u>47</u>
CONCEPTOS BÁSICOS SOBRE UML.....	47
BREVE RESEÑA HISTÓRICA.....	47
¿QUÉ ES LA OMG?	47
PRESENTACIÓN DE LOS DIAGRAMAS DE UML.....	48
DIAGRAMA DE CASO DE USO	50
COMPONENTES DEL DIAGRAMA DE CASO DE USO	50
DIAGRAMA DE CLASES.....	62
COMPONENTES DEL DIAGRAMA DE CLASES.....	62
DIAGRAMA DE MÁQUINA DE ESTADOS	70
COMPONENTES DEL DIAGRAMA DE MÁQUINA DE ESTADO	70
DIAGRAMA DE SECUENCIA.....	75
COMPONENTES DEL DIAGRAMA DE SECUENCIA	76
<u> LENGUAJE DE PROGRAMACIÓN ORIENTADA A OBJETOS</u>	<u>82</u>
SURGIMIENTO DEL LENGUAJE	82
ENTENDIENDO JAVA.....	82
EL ENTORNO DE DESARROLLO DE JAVA.....	83
EL COMPILADOR DE JAVA.....	83
LA JAVA VIRTUAL MACHINE	83
APLICACIONES DE JAVA	85
ANTES DE COMENZAR.....	85
DECLARACIÓN DE CLASES	86
TIPOS DE DATOS	87
TIPOS BÁSICOS DE DATOS.....	87
TIPOS DE DATOS REFERENCIA.....	88
CONVERSIÓN DE TIPOS DE DATOS O CASTING	89
EL RECOLECTOR DE BASURA	89
ÁMBITO DE LAS VARIABLES.....	89
OPERADORES	90
ARRAYS	94
ESTRUCTURAS DE CONTROL.....	95
ESTRUCTURAS CONDICIONALES.....	95
BUCLAS.....	97
BLOQUES PARA MANEJO DE EXCEPCIONES	98
CLASES Y OBJETOS	99

VARIABLES MIEMBROS DE OBJETO	100
MÉTODOS.....	101
HERENCIA DE CLASES	109
SOBRESCRITURA DE VARIABLES Y MÉTODOS	110
CLASES Y MÉTODOS ABSTRACTOS	112
INTERFACES.....	112
DOCUMENTACIÓN DE CLASES Y MÉTODOS	114
COLECCIONES DE TAMAÑO VARIABLE.....	116
LISTAS	117
LA CLASE JAVA.UTILSTACK.....	120
EXCEPCIONES	121
LA API DE JAVA	126
PROGRAMANDO INTERFACES GRÁFICAS	129
<u>REFERENCIA DE FIGURAS Y TABLAS.....</u>	<u>132</u>
FIGURAS	132
TABLAS	134
FUENTES DE INFORMACIÓN	135

Definición del Módulo

Denominación de Módulo: **Programación Orientada a Objetos**

Presentación

El módulo Programación orientada a Objetos tiene, como propósito general, contribuir a que los estudiantes desarrollen capacidades técnicas de programación con objetos. Profundiza y amplía las capacidades construidas en el módulo Técnicas de Programación dado que se emplean las herramientas adquiridas en este último, en una nueva modalidad de resolución de problemas.

Este módulo se constituye, así en un espacio de formación que permite a los estudiantes desarrollar saberes propios de la formación específica de la figura profesional de “Programador”.

En este contexto se entiende por orientación a objetos a un paradigma de programación que facilita la creación de software de calidad, debido a sus características específicas que potencian el mantenimiento, la extensión y la reutilización del software generado. Los programas se organizan como colecciones cooperativas de objetos, cada uno de los cuales representa una instancia de alguna clase, y cuyas clases son miembros de una jerarquía de clases vinculadas mediante relaciones de herencia.

Es de central importancia que las actividades de enseñanza de la programación orientada a objetos se asocien a prácticas cercanas (simuladas o reales) al tipo de intervención profesional del Programador.

Para la organización de la enseñanza de esta unidad curricular se han organizado los contenidos en tres bloques:

- Fundamentos de la Programación Orientada a Objetos
- Metodología de desarrollo
- Lenguaje de POO

El bloque de **Fundamentos de la Programación Orientada a Objetos** presenta los conceptos básicos de este paradigma: abstracción, encapsulamiento, modularización, jerarquía de clases y jerarquía de partes, polimorfismo y relaciones entre clases. Este bloque mantiene estrecha relación con el bloque Lenguaje de POO, ya que los conceptos que se abordan en este bloque, se implementan y desarrollan inmediatamente en el lenguaje seleccionado. Esta relación permite lograr una mayor comprensión de los conceptos y un acercamiento temprano al lenguaje y a los principios de desarrollo de software de calidad.

El bloque **Metodología de desarrollo** aborda las técnicas de resolución de problemas informáticos bajo la óptica del paradigma Orientado a Objetos utilizando un proceso de desarrollo y un lenguaje de modelado unificado (Proceso Unificado de Desarrollo / Lenguaje de Modelado Unificado).

El bloque **Lenguaje de POO** tiene, como núcleo central, la elaboración y la construcción de aplicaciones implementando los conceptos de POO y el modelado de situaciones problemáticas en un lenguaje adecuado al paradigma en un entorno de desarrollo corporativo o abierto.

La organización del programa curricular, que se presenta en este documento, sigue una secuencia que toma como punto de partida la identificación de las clases que modelan el problema, sus relaciones y

representación mediante UML y por último su codificación en un lenguaje de programación orientado a objetos, dando como resultado la aplicación.

El propósito general de esta unidad curricular es que los/as alumnos/as construyan habilidades y conocimientos para resolver problemas e implementar sus soluciones en un lenguaje de programación orientado a objetos, logrando piezas de software de calidad, siendo el abordaje de este módulo teórico-práctico.

El módulo “Programación orientada a objetos” recupera e integra conocimientos, saberes y habilidades cuyo propósito general es contribuir al desarrollo de los estudiantes de una formación especializada, integrando contenidos, desarrollando prácticas formativas y su vínculo con los problemas característicos de intervención y resolución técnica del Programador, en particular con las funciones que ejerce el profesional en relación a:

Interpretar especificaciones de diseño de las asignaciones a programar en el contexto del desarrollo de software en el que participa.

Este módulo se orienta al desarrollo de las siguientes capacidades profesionales referidas al perfil profesional en su conjunto:

- Interpretar las especificaciones formales o informales del Líder de proyecto
- Analizar el problema a resolver
- Interpretar el material recibido y clarificar eventuales interpretaciones
- Determinar el alcance del problema y convalidar su interpretación a fin de identificar aspectos faltantes
- Comprender lo especificado observando reglas del lenguaje de POO
- Comunicarse en un lenguaje preciso y adecuado con los integrantes del equipo de trabajo

En relación a las **prácticas formativas de carácter profesionalizante**, son un eje estratégico de la propuesta pedagógica para el ámbito de la Formación Profesional (FP), al situar al participante en los ámbitos reales de trabajo con las problemáticas características que efectivamente surgen en la planificación de procedimientos o secuencias de actividades asociada al desarrollo de algoritmos y la resolución de problemas de base computacional, y que se organiza a nivel de cada módulo formativo.

Para el caso del presente módulo las prácticas formativas profesionalizantes y los objetivos de aprendizajes se organizan para el desarrollo de:

Prácticas de resolución de una situación problemática, real o simulada de acuerdo a especificaciones de diseño, desarrollando aplicaciones que den solución a problemas específicos.

Introducción

El conjunto de órdenes e instrucciones que se dan a la computadora para que resuelva un problema o ejecute una determinada misión, recibe el nombre de programa. En los primeros tiempos de la informática, la programación se efectuaba en el único lenguaje que entiende el microprocesador: su propio código, también denominado lenguaje máquina o código máquina. Pero la programación en lenguaje máquina resulta muy lenta y tediosa, pues los datos e instrucciones se deben introducir en sistema binario¹ y, además, obliga a conocer las posiciones de memoria donde se almacenan los datos.

Una de las primeras formas de escritura de programas para computadoras fueron las tarjetas perforadas. Una tarjeta perforada es una pieza de cartulina que contiene información digital representada mediante la presencia o ausencia de agujeros en posiciones predeterminadas. Comenzaron a usarse en el siglo 19 para el control de telares, aunque no fue hasta mediados del siglo 20 cuando empezaron a usarse en los ordenadores para el almacenamiento de programas y datos. Actualmente, es considerado un método obsoleto de almacenamiento, pese a que aún sigue siendo utilizado en algunos artefactos, como las máquinas para emitir votaciones en los comicios electorales.

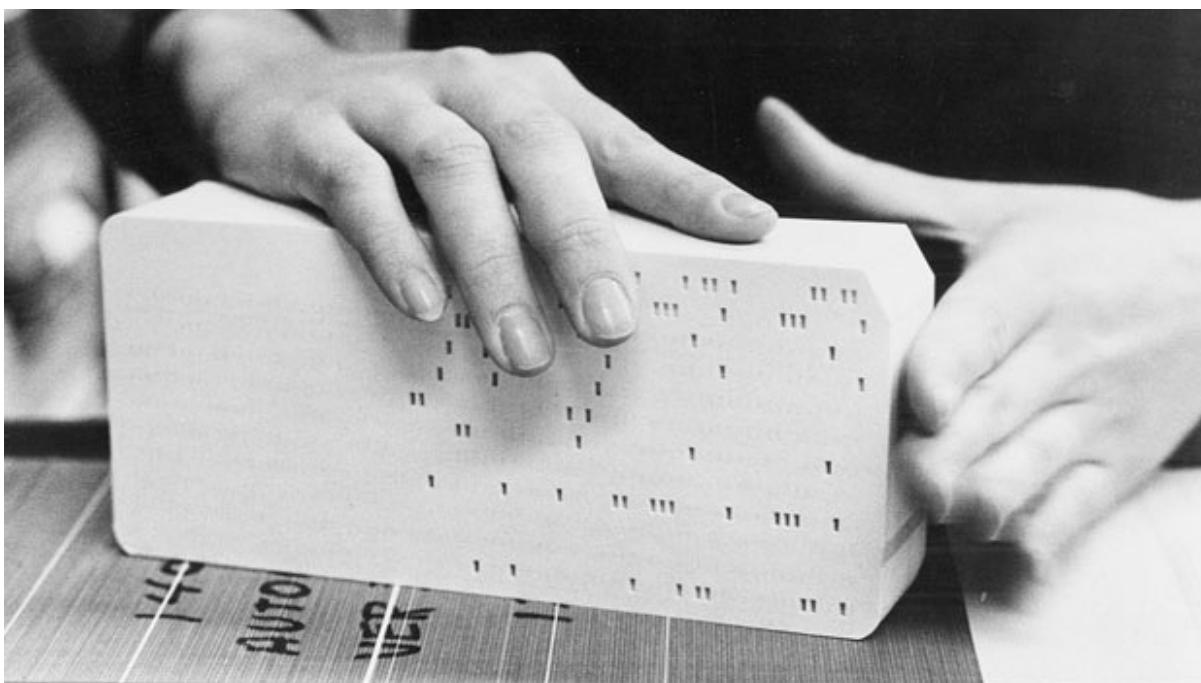


Fig. 1 – Tarjetas perforadas para programación de computadoras

La invención de las tarjetas perforadas data del 1725, cuando los franceses Basille Bouchon y Jean-Baptiste Falcon las crearon para facilitar el control de los telares mecánicos. Esta idea fue posteriormente explotada por distintos inventores como el francés Joseph Marie Jacquard que la uso

¹ Sistema Binario: es el sistema numérico usado para la representación de textos, o procesadores de instrucciones de computadora, utilizando el sistema binario (sistema numérico de dos dígitos, o bits: el "0" /cerrado/ y el "1" /abierto/)

para el control de su telar, y el británico Charles Babbage, que tuvo la idea de usarla para el control de la calculadora mecánica que había diseñado.

En 1890 fue el estadista Herman Hollerit el que usó la tecnología de las tarjetas perforadas para la máquina tabuladora encargada de realizar el censo de los estados unidos en 1890.

La tecnología siguió desarrollándose hasta que en 1950 IBM empezó a usarla como soporte de almacenamiento para sus máquinas.

Como se puede imaginar, este tipo de programación conlleva gran número de errores y la tarea de depuración exige bastante tiempo y dedicación. Por este motivo, a principios de los 50 se creó una notación simbólica, denominada código de ensamblaje (ASSEMBLY), que utiliza una serie de abreviaturas mnemotécnicas para representar las operaciones: ADD (sumar), STORE (copiar), etc. Al principio, la traducción del código de ensamblaje al código máquina se realizaba manualmente, pero enseguida se vió que la computadora también podía encargarse de esa traducción; se desarrolló así un programa traductor, llamado *ensamblador o ASSEMBLER*.

```

8 .....
9
10      ORG $4000
11 A1    = $3C
12 A2    = $3E
13 A4    = $42
14 AUXMOVE = $C311
15
16 .....
17 * SETUP - move data for VTAC
18 * and catalog to auxmem at
19 * B000-B3FF (pseudo trk 11
20 * 0-3)
21 .....
22 SETUP   LDA #<VTAC
23         STA A1
24         LDA #>VTAC
25         STA A1+1
26         LDA #<END
27         STA A2
28         LDA #>END
29         STA A2+1
30         LDA #$00
31         STA A4
32         LDA #$00
33         STA A4+1
34         SEC
35         JMP AUXMOVE
36
37         DS 4
38 .....

```

Fig. 2 – Instrucciones en Assembly en la vista de Terminator (1984)

Conforme los ordenadores fueron introduciéndose en el mundo empresarial y académico, aquellos primitivos lenguajes fueron sustituidos por otros más sencillos de aprender y más cómodos de emplear. En la sección siguiente se aborda con más detalle conceptos sobre los lenguajes de programación.

Lenguajes de Programación

Los lenguajes de programación son todos los símbolos, caracteres y reglas de uso que permiten a las personas "comunicarse" con las computadoras.

Un lenguaje de programación es un sistema estructurado y diseñado principalmente para que las computadoras se entiendan entre sí y con nosotros, los humanos. Contiene un conjunto de acciones consecutivas que el ordenador² debe ejecutar.

Estos lenguajes de programación usan diferentes normas o bases y se utilizan para controlar cómo se comporta una máquina (por ejemplo, un ordenador), también pueden usarse para crear programas informáticos que formarán productos de software.

El término "programación" se define como un proceso por medio del cual se diseña, se codifica, se escribe, se prueba y se depura un código básico para las computadoras. Ese código es el que se llama "código fuente" que caracteriza a cada lenguaje de programación. Cada lenguaje de programación tiene un "código fuente" característico y único que está diseñado para una función o un propósito determinado y que nos sirven para que una computadora se comporte de una manera deseada.

En la actualidad existe un gran número de lenguajes de programación diferentes³. Algunos, denominados lenguajes específicos de dominio (o con las siglas en inglés, DSL, Domain Specific Languages) se crean para una aplicación especial, mientras que otros son herramientas de uso general, más flexibles, que son apropiadas para muchos tipos de aplicaciones. En todo caso los lenguajes de programación deben tener instrucciones que pertenecen a las categorías ya familiares de entrada/salida, cálculo/manipulación de textos, lógica/comparación y almacenamiento / recuperación.

A continuación, presentamos un recorrido en el tiempo por los lenguajes de programación más conocidos:

1957-1959

- Fortran (Formula Translation)
- LISP (List Procesor)
- COBOL (Common Business-Oriented Language)

Considerados los lenguajes más viejos utilizados hoy en día. Son lenguajes de alto nivel que fueron creados por científicos, matemáticos y empresarios de la computación.

Principales usos: Aplicaciones científicas y de ingeniería para supercomputadoras, desarrollo de Inteligencia Artificial, software empresarial.

² En el contexto de este apunte, computadora y ordenador son sinónimos.

³ https://es.wikipedia.org/wiki/Anexo%3ALenguajes_de_programaci%C3%B3n, este sitio lista en orden alfabético los lenguajes de programación existentes, tanto de uso actual como histórico.

1970

-

Pascal (nombrado así en honor al matemático y físico Francés Blaise Pascal)



Lenguaje de alto nivel, utilizado para la enseñanza de la programación estructurada y la estructuración de datos. Las versiones comerciales de Pascal fueron ampliamente utilizadas en los años 80's.

Creador: Niklaus Wirth

Principales usos: Enseñanza de la programación. Objet Pascal, un derivado, se utiliza comúnmente para el desarrollo de aplicaciones Windows.

Usado por: Apple Lisa (1983) y Skype.

1972

-

C (Basado en un programa anterior llamado "B")



Lenguaje de propósito general, de bajo nivel. Creado por Unix Systems, en Bell Labs. Es el lenguaje más popular (precedido por Java). De él se derivan muchos lenguajes como C#, Java, Javascript, Perl, PHP y Python.

Creador: Dennis Ritchie (Laboratorios Bell)

Principales usos: Programación multiplataforma, programación de sistemas, programación en Unix y desarrollo de videojuegos.
Usado por: Unix (reescrito en C en 1973), primeros servidores y clientes de la WWW.

1983

- C++ (Originariamente “C con clases”; ++ es el operador de incremento en “C”)



Lenguaje multiparadigma. Una extensión de C con mejoras como clases, funciones virtuales y plantillas.

Creador: Bjarne Stroustrup (Laboratorios Bell)

Principales usos: Desarrollo de aplicaciones comerciales, software embebido, aplicaciones cliente-servidor en videojuegos.

Usado por: Adobe, Google Chrome, Mozilla Firefox, Microsoft Internet Explorer.

- Objective-C (Object-oriented extension de “C”)



Lenguaje de propósito general, de alto nivel. Ampliado en C, adicionaba una funcionalidad de paso de mensajes. Se hizo muy popular por ser el lenguaje preferido para el desarrollo de aplicaciones para productos de Apple en los últimos años hasta ser reemplazado por Swift.

Creador: Brad Cox y Tom Love (Stepstone)

Principales usos: Programación Apple.

Usado por: Apple OS X y sistemas operativos iOS

1987

- Perl (“Pearl” ya estaba ocupado)



Lenguaje de propósito general, de alto nivel, muy poderoso en el manejo de expresiones regulares. Creado para el procesamiento de reportes en sistemas Unix. Hoy en día es conocido por su alto poder y versatilidad.

Creador: Larry Wall (Unisys)

Principales usos: Imágenes generadas por computadora, aplicaciones de base de datos, administración de sistemas, programación web y programación de gráficos.

Usado por: IMDb, Amazon, Priceline, Ticketmaster

1991

- Python (en honor a la compañía de comedia británica Monty Python)



Lenguaje de propósito general, de alto nivel. Creado para apoyar una gran variedad de estilos de programación de manera divertida. Muchos tutoriales, ejemplos de código e instrucciones a menudo contienen referencias a Monty Python.

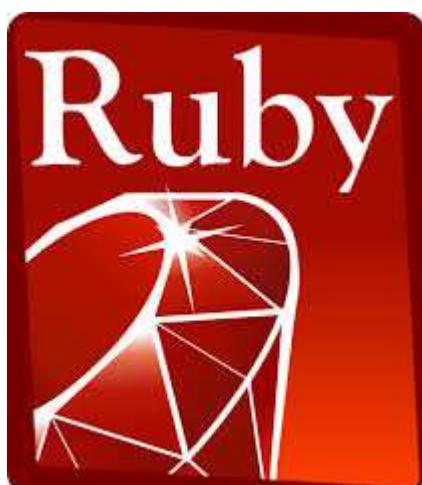
Creador: Guido Van Rossum (CWI)

Principales usos: Aplicaciones Web, desarrollo de software, seguridad informática.

Usado por: Google, Yahoo, Spotify

1993

- Ruby (La piedra del zodiaco de uno de los creadores).



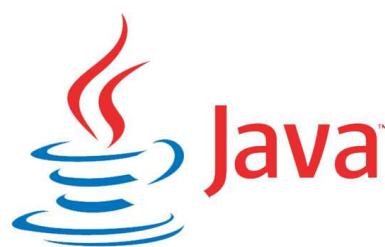
Lenguaje de propósito general, de alto nivel. Un programa de enseñanza, influenciado por Perl, Ada, Lisp, Smalltalk, entre otros. Diseñado para hacer la programación más productiva y agradable.

Creador: Yukihiro Matsumoto

Principales usos: Desarrollo de aplicaciones Web, Ruby on Rails.

Usado por: Twitter, Hulu, Groupon.

1995



Lenguaje de propósito general, de alto nivel. Hecho para un proyecto de televisión interactiva. Funcionalidad de programación multiplataforma. Es actualmente el lenguaje de programación más popular en el mundo⁴.

Creador: James Gosling (Sun Microsystems)

Principales usos: Programación Web, desarrollo de aplicaciones Web, desarrollo de software, desarrollo de interfaz gráfica de usuario.

Usado por: Android OS/Apps

- PHP (Formalmente: “Personal Home Page”, ahora es por “Hypertext Preprocessor”).



Lenguaje de código abierto, de propósito general. Se utiliza para construir páginas web dinámicas. Más ampliamente usado en software de código abierto para empresas.

Creador: Rasmus Lerdorf

Principales usos: Construcción y mantenimiento de páginas web dinámicas, desarrollo del lado del servidor.

Usado por: Facebook, Wikipedia, Digg, WordPress, Joomla.

- Javascript



JavaScript

Lenguaje de alto nivel. Creado para extender las funcionalidades de las páginas web. Usado por páginas dinámicas para el envío y validación de formularios, interactividad, animación, seguimiento de actividades de usuario, etc.

Creador: Brendan Eich (Netscape)

Principales usos: Desarrollo de web dinámica, documentos PDF, navegadores web y widgets de Escritorio.

Usado por: Gmail, Adobe Photoshop, Mozilla Firefox.

⁴ <http://www.tiobe.com/tiobe-index> sitio que muestra un ranking de lenguajes de programación.

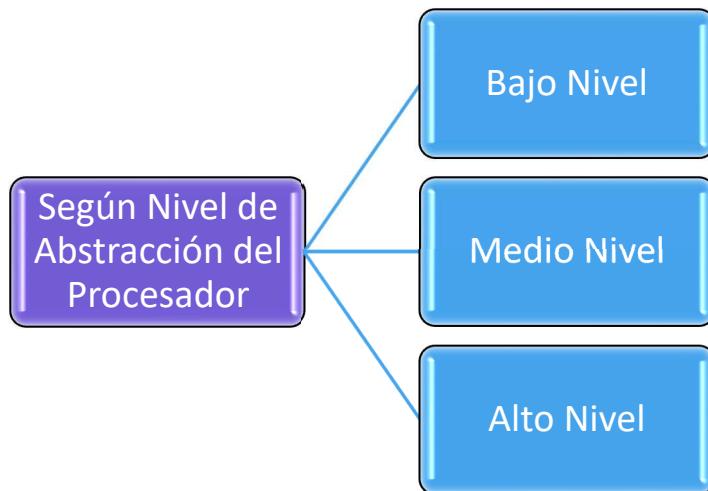
Tipos de Lenguajes de Programación

Para conocer un poco más sobre los lenguajes de programación analizaremos algunas clasificaciones posibles:



Nivel de Abstracción del Procesador

Según el nivel de abstracción del procesador, los lenguajes de programación, se clasifican en:



Lenguajes de Bajo Nivel

Es el que proporciona poca o ninguna abstracción del microprocesador de un ordenador. Consecuentemente es fácilmente trasladado a lenguaje de máquina. En general se utiliza este tipo de lenguaje para programar controladores (drivers).

Son aquellos utilizados fundamentalmente para controlar el “hardware” del ordenador y dependen totalmente de la máquina y no se pueden utilizar en otras máquinas. Están orientados exclusivamente para la máquina. Estos lenguajes son los que ordenan a la máquina operaciones fundamentales para que pueda funcionar. Utiliza básicamente ceros, unos y abreviaturas de letras. Estos lenguajes también se

llaman de código máquina. Son los más complicados, pero sólo los usan prácticamente los creadores de las máquinas.

Estos lenguajes tienen mayor adaptación al hardware y obtienen la máxima velocidad con mínimo uso de memoria. Como desventajas, estos lenguajes no pueden escribir código independiente de la máquina, son más difíciles de utilizar y comprender, exigen mayor esfuerzo a los programadores, quienes deben manejar más de un centenar de instrucciones y conocer en detalle la arquitectura de la máquina.

Lenguajes de Alto Nivel

Estos lenguajes se caracterizan por expresar los algoritmos de una manera adecuada a la capacidad cognitiva humana, en lugar de a la capacidad ejecutora de las máquinas. En los primeros lenguajes de alto nivel la limitación era que se orientaban a un área específica y sus instrucciones requerían de una sintaxis predefinida. Se clasifican como lenguajes procedimentales.

Otra limitación de los lenguajes de alto nivel es que se requiere de ciertos conocimientos de programación para realizar las secuencias de instrucciones lógicas.

Los lenguajes de muy alto nivel se crearon para que el usuario común pudiese solucionar tal problema de procesamiento de datos de una manera más fácil y rápida.

La diferencia fundamental se puede explicar con el siguiente ejemplo:

En un lenguaje de alto nivel sólo tengo que poner **sqt(x)**, que sería una función predeterminada, calcular el cuadrado de x.

Si fuera de bajo nivel, yo mismo tendría que crear la función sabiendo cómo funciona el cuadrado de un número:

$$\text{cuadrado}(x) = x * x$$

Lenguajes de Medio Nivel

Es un lenguaje de programación informática, híbrido, como el lenguaje C, que se encuentran entre los lenguajes de alto nivel y los lenguajes de bajo nivel.

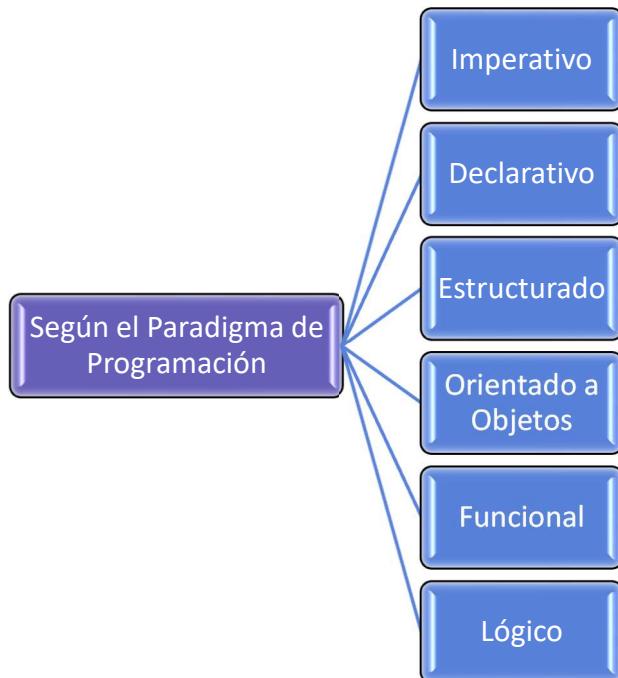
Suelen ser clasificados muchas veces de alto nivel, pero permiten ciertos manejos de bajo nivel. Son precisos para ciertas aplicaciones como la creación de sistemas operativos, ya que permiten un manejo abstracto (independiente de la máquina, a diferencia del ensamblador), pero sin perder mucho del poder y eficiencia que tienen los lenguajes de bajo nivel.

Una característica distintiva, por ejemplo, que convierte a C en un lenguaje de medio nivel es que es posible manejar las letras como si fueran números.

Una de las características más peculiares del lenguaje de programación C, es el uso de "punteros", los cuales son muy útiles en la implementación de algoritmos como Listas ligadas, Tablas Hash y algoritmos de búsqueda y ordenamiento.

Paradigma de Programación

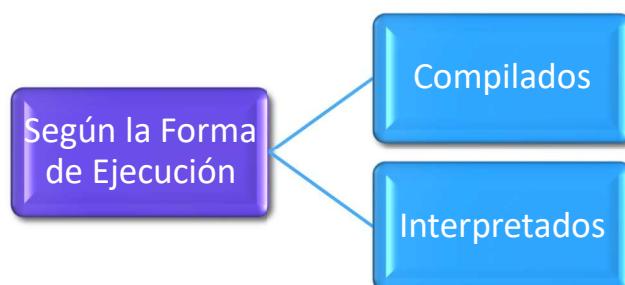
Según el Paradigma de Programación, los lenguajes se clasifican en:



La descripción de cada uno de los paradigmas tiene una sección específica, más adelante en este material.

Forma de Ejecución

Según la Forma de Ejecución, los lenguajes de programación, se clasifican en:



Lenguajes Compilados

Los compiladores son aquellos programas cuya función es traducir un programa escrito en un determinado lenguaje a un idioma que la computadora entienda (lenguaje máquina con código binario). Al usar un lenguaje compilado, el programa desarrollado es controlado previamente, por el compilador, y por eso nunca se ejecuta si tiene errores de código. Es decir, se compila y si la compilación es exitosa ese programa se puede ejecutar.

Un programa compilado es aquel cuyo código fuente, escrito en un lenguaje de alto nivel, es traducido por un compilador a un archivo ejecutable entendible para la máquina en determinada plataforma. Con ese archivo se puede ejecutar el programa cuantas veces sea necesario sin tener que repetir el proceso por lo que el tiempo de espera entre ejecución y ejecución es ínfimo.

Dentro de los lenguajes de programación que son compilados tenemos la familia C que incluye a C++, Objective C, C# y también otros como Fortran, Pascal, Haskell y Visual Basic.

Java es un caso particular ya que hace uso de una máquina virtual que se encarga de la traducción del código fuente por lo que hay veces que es denominado compilado e interpretado. Otra ventaja de la máquina virtual que usa Java, es que le permite ejecutar código Java en cualquier máquina que tenga instalada la JVM (Java Virtual Machine).

Lenguajes Interpretados

Básicamente un lenguaje interpretado es aquel en el cual sus instrucciones o más bien el código fuente, escrito por el programador en un lenguaje de alto nivel, es traducido por el intérprete a un lenguaje entendible para la máquina paso a paso, instrucción por instrucción. El proceso se repite cada vez que se ejecuta el programa el código en cuestión.

Estos lenguajes utilizan una alternativa diferente de los compiladores para traducir lenguajes de alto nivel. En vez de traducir el programa fuente y grabar en forma permanente el código objeto que se produce durante la corrida de compilación para utilizarlo en una corrida de producción futura, el programador sólo carga el programa fuente en la computadora junto con los datos que se van a procesar. A continuación, un programa intérprete, almacenado en el sistema operativo del disco, o incluido de manera permanente dentro de la máquina, convierte cada proposición del programa fuente en lenguaje de máquina conforme vaya siendo necesario durante el proceso de los datos. No se graba el código objeto para utilizarlo posteriormente

El uso de los lenguajes interpretados ha venido en crecimiento y cuyos máximos representantes son los lenguajes usados para el desarrollo web entre estos Ruby, Python, PHP, JavaScript y otros como Perl, Smalltalk, MATLAB, Mathematica.

Los lenguajes interpretados permiten el tipado dinámico de datos, es decir, no es necesario inicializar una variable con determinado tipo de dato, sino que esta puede cambiar su tipo en condición al dato que almacene entre otras características más.

También tienen por ventaja una gran independencia de la plataforma donde se ejecutan de ahí que los tres primeros mencionados arriba sean multiplataforma comparándolos con algunos lenguajes compilados como Visual Basic, y los programas escritos en lenguajes interpretados utilizan menos recursos de hardware (más livianos).

La principal desventaja de estos lenguajes es el tiempo que necesitan para ser interpretados. Al tener que ser traducido a lenguaje máquina con cada ejecución, este proceso es más lento que en los lenguajes compilados; sin embargo, algunos lenguajes poseen una máquina virtual que hace una traducción a lenguaje intermedio con lo cual el traducirlo a lenguaje de bajo nivel toma menos tiempo.

¿Qué son los Paradigmas?



Los paradigmas son poderosos porque crean los cristales o las lentes a través de los cuales vemos el mundo. El poder de un cambio de paradigma es el poder esencial de un cambio considerable, ya se trate de un proceso instantáneo o lento y pausado.

(Stephen Covey)

El concepto de paradigma se utiliza en la vida cotidiana como sinónimo de “marco teórico” o para hacer referencia a que algo se toma como “modelo a seguir”.

En términos generales, se puede definir al término paradigma como la forma de visualizar e interpretar los múltiples conceptos, esquemas o modelos del comportamiento en diversas disciplinas. A partir de la década del '60, los alcances del concepto se ampliaron y ‘paradigma’ comenzó a ser un término común en el vocabulario científico y en expresiones epistemológicas cuando se hacía necesario hablar de modelos o patrones.

Una de las primeras figuras de la historia que abordaron el concepto que ahora nos ocupa fue el gran filósofo griego Platón que realizó su propia definición de lo que él consideraba que era un paradigma. En este sentido, el citado pensador expuso que esta palabra venía a determinar *qué son las ideas o los tipos de ejemplo de una cosa en cuestión*.

El estadounidense Thomas Kuhn, un experto en Filosofía y una figura destacada del mundo de las ciencias, fue quien se encargó de renovar la definición teórica de este término para otorgarle una acepción más acorde a los tiempos actuales, al adaptarlo para describir con él a *la serie de prácticas que trazan los lineamientos de una disciplina científica a lo largo de un cierto lapso temporal*.

De esta forma, un paradigma científico establece aquello que debe ser observado; la clase de interrogantes que deben desarrollarse para obtener respuestas en torno al propósito que se persigue; qué estructura deben poseer dichos interrogantes y marca pautas que indican el camino de interpretación para los resultados obtenidos de una investigación de carácter científico.

Cuando un paradigma ya no puede satisfacer los requerimientos de una ciencia (por ejemplo, ante nuevos hallazgos que invalidan conocimientos previos), es sucedido por otro. Se dice que un cambio de paradigma es algo dramático para la ciencia, ya que éstas se perciben como estables y maduras.

En las ciencias sociales, el paradigma se encuentra relacionado al concepto de cosmovisión. El concepto se emplea para mencionar a todas aquellas experiencias, creencias, vivencias y valores que repercuten y condicionan el modo en que una persona ve la realidad y actúa en función de ello. Esto quiere decir que un **paradigma es también la forma en que se entiende el mundo**.

Paradigmas de Programación

Un paradigma de programación provee y determina la visión y métodos de un programador en la construcción de un programa o subprograma. Diferentes paradigmas resultan en diferentes estilos de programación y en diferentes formas de pensar la solución de problemas (con la solución de múltiples “problemas” se construye una aplicación o producto de software).

Los lenguajes de programación están basados en uno o más paradigmas, por ejemplo: Smalltalk y Java son lenguajes basados en el paradigma orientado a objetos. El lenguaje de programación Scheme, en

cambio, soporta sólo programación funcional. Otros lenguajes, como C++ y Python soportan múltiples paradigmas.

Clasificación de Paradigmas de Programación

Un paradigma de programación representa un enfoque particular o filosofía para diseñar soluciones. Los paradigmas difieren unos de otros, en los conceptos y la forma de abstraer los elementos involucrados en un problema, así como en los pasos que integran su solución del problema, en otras palabras, el cómputo. Además, los paradigmas dependen del contexto y el tiempo en el que surgen, ya que nacen en base a una necesidad generalizada de la comunidad de desarrolladores de software, para resolver cierto tipo de problemas de la vida real.

Un paradigma de programación está delimitado en el tiempo en cuanto a aceptación y uso, porque nuevos paradigmas aportan nuevas o mejores soluciones que la sustituyen parcial o totalmente.

De esta forma podemos encontrar los siguientes tipos de paradigmas:

Paradigma Imperativo

Describe la programación como una secuencia de instrucciones o comandos que cambian el estado de un programa. El código máquina en general está basado en el paradigma imperativo. Su contrario es el paradigma declarativo. En este paradigma se incluye el paradigma procedural.

Paradigma Declarativo

No se basa en el cómo se hace algo (cómo se logra un objetivo paso a paso), sino que describe (declara) cómo es algo. En otras palabras, se enfoca en describir las propiedades de la solución buscada, dejando indeterminado el algoritmo (conjunto de instrucciones) usado para encontrar esa solución. Es más complicado de implementar que el paradigma imperativo, tiene desventajas en la eficiencia, pero ventajas en la solución de determinados problemas.

Paradigma Estructurado

La programación se divide en bloques (procedimientos y funciones) que pueden o no comunicarse entre sí. Además, la programación se controla con secuencia, selección e iteración. Permite reutilizar código programado y otorga una mejor compresión de la programación. Es contrario al paradigma no estructurado, de poco uso, que no tiene ninguna estructura, es simplemente un “bloque”, como, por ejemplo, los archivos en lote o batch (.bat).

Paradigma Funcional

Este paradigma concibe a la computación como la evaluación de funciones matemáticas y evita declarar y cambiar datos. En otras palabras, hace hincapié en la aplicación de las funciones y composición entre ellas, más que en los cambios de estados y la ejecución secuencial de comandos (como lo hace el paradigma procedimental). Permite resolver ciertos problemas de forma elegante y los lenguajes puramente funcionales evitan los efectos secundarios comunes en otro tipo de paradigmas.

Paradigma lógico

Este paradigma se basa en los conceptos de lógica matemática; trabaja con predicados que caracterizan o relacionan a los individuos involucrados y la deducción de las posibles respuestas a una determinada consulta.

Es un tipo de paradigma de programación declarativo. La programación lógica gira en torno al concepto de predicado, o relación entre elementos. Define reglas lógicas para luego, a través de un motor de inferencias lógicas, responder preguntas planteadas al sistema y así resolver los problemas. Ej.: Prolog.

Paradigma Orientado a Objetos

La programación orientada a objetos intenta simular el mundo real a través del significado de objetos que contiene características y funciones. Está basado en la idea de encapsular estado y operaciones en objetos. En general, la programación se resuelve comunicando dichos objetos a través de mensajes. Su principal ventaja es la reutilización de código y su facilidad para pensar soluciones a determinados problemas.

El lenguaje de programación elegido para trabajar es **Java**, que pertenece al paradigma orientado a objetos, el cual desarrollaremos más adelante en este apunte.

Desarrollo de Software con el Paradigma Orientado a Objetos

¿Qué es una Clase?

Los conceptos de clase y objeto están estrechamente relacionados; sin embargo, existen diferencias entre ambos: mientras *que un objeto es una entidad concreta que existe en el espacio y en el tiempo*, una clase representa sólo una abstracción⁵, la “esencia” de un objeto.

Se puede definir una clase como un grupo, conjunto o tipo marcado por atributos comunes, una división, distinción o clasificación de grupos basada en la calidad, grado de competencia o condición. En el contexto del paradigma Orientado a Objetos se define una clase de la siguiente forma:

“Una clase describe un grupo de objetos que comparten una estructura y un comportamiento comunes”

Un objeto es una instancia de una clase. Los objetos que comparten estructura y comportamiento similares pueden agruparse en una clase. En el paradigma orientado a objetos, los objetos pertenecen siempre a una clase, de la que toman su estructura y comportamiento. Por mucho tiempo se ha referenciado a las clases llamándolas objetos, no obstante, son conceptos diferentes.

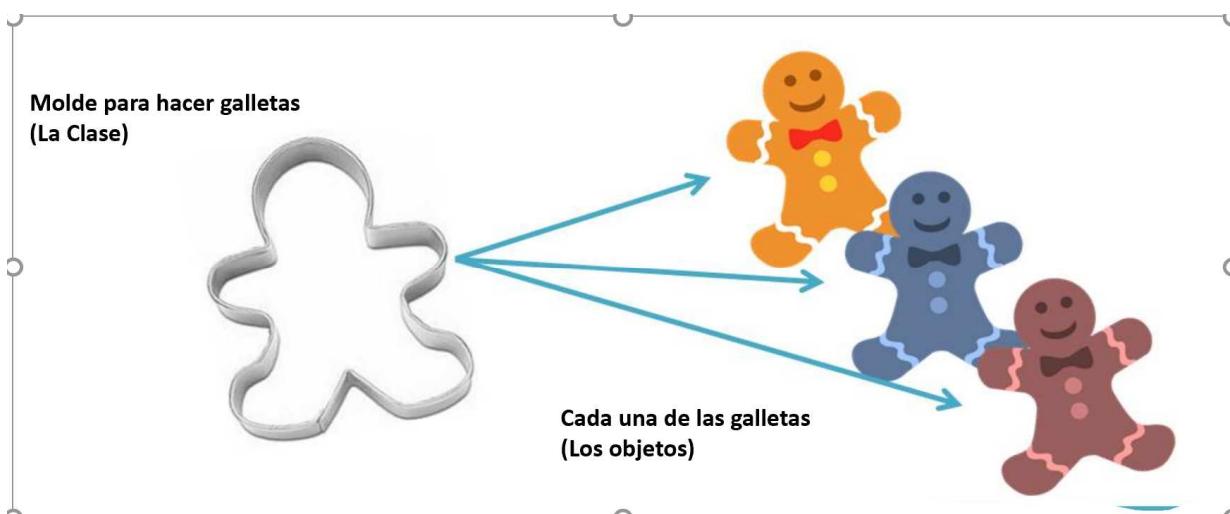


Fig. 3 - Clases y Objetos

Interfaz e implementación

La programación es en gran medida un asunto de contratos: las diversas funciones de un problema mayor se descomponen en problemas más pequeños mediante subcontratos a diferentes elementos del diseño. En ningún sitio es más evidente esta idea que en el diseño de clases.

⁵ **Abstracción:** propiedad vinculada al verbo abstraer. Es la acción de separar propiedades, en informática es muy utilizada como mecanismo de simplificación de la realidad. Se separan características esenciales de las que no lo son tanto. Se separa el “que” debe hacerse del “como” se hará.

Una clase sirve como una especie de contrato que vincula a una abstracción y todos sus clientes. Esta visión de la programación como un contrato lleva a distinguir entre la visión externa y la visión interna de una clase. La interfaz de una clase proporciona su visión externa y enfatiza la abstracción a la vez que oculta su estructura y los secretos de su comportamiento. Esta interfaz se compone principalmente de las declaraciones de todas las operaciones aplicables a instancias de la misma clase.

La implementación de una clase se compone principalmente de la implementación de todas las operaciones definidas en la interfaz de la misma. Lleva a una forma concreta las declaraciones abstractas de cómo debe ser la implementación.

Se puede dividir la interfaz de una clase en tres partes:

- **Pública:** Una declaración accesible a todos los clientes
- **Protegida:** Una declaración accesible sólo a la propia clase, sus subclases y sus clases amigas.
- **Privada:** Una declaración accesible sólo a la propia clase.

Los distintos lenguajes de programación ofrecen combinaciones de estas partes para poder elegir y establecer derechos específicos de acceso para cada parte de la interfaz de una clase y de este modo controlar lo que cada cliente puede y no puede ver.

¿Qué es un Objeto?

Desde el punto de vista de los humanos un objeto es cualquiera de estas cosas:

- Una cosa tangible y/o visible.
- Algo que puede comprenderse intelectualmente.
- Algo hacia lo que se dirige un pensamiento o una acción.

En software, el término objeto se aplicó formalmente por primera vez en el lenguaje Simula, por primera vez, donde los objetos existían para simular algún aspecto de la realidad. Los objetos del mundo real no son el único tipo de objeto de interés en el desarrollo de software. Otros tipos importantes de objetos son invenciones del proceso de diseño cuyas colaboraciones con objetos semejantes llevan a desempeñar algún comportamiento de nivel superior.

Podemos decir, entonces, que un objeto: “representa un elemento, unidad o entidad individual e identificable, ya sea real o abstracta, con un *rol bien definido* en el dominio del problema”.

En términos más generales, se define a un objeto como: “cualquier cosa que tenga una frontera definida con nitidez”, pero esto no es suficiente para servir de guía al distinguir un objeto de otro ni permite juzgar la calidad de las abstracciones.

De esta forma en el dominio de un problema relacionado con vehículos podríamos identificar como ejemplos de objetos a cada una de las ruedas, las puertas, las luces y demás elementos que lo conforman. Los casos anteriores son representaciones de cosas tangibles y reales, aunque también pueden serlo entidades abstractas como la marca, el modelo, el tipo de combustible que usa, entre otras. Además, como veremos luego, un objeto puede estar formado por otros objetos y en el ejemplo anterior el Automóvil en sí podría ser un objeto formado por los otros objetos que ya mencionamos.

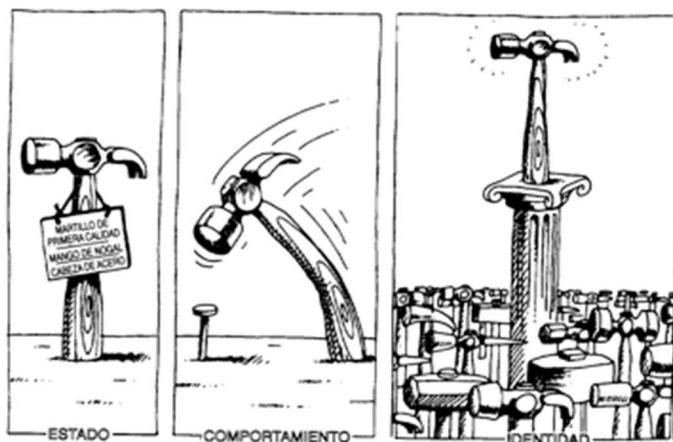


Fig. 4. Estado, comportamiento e identidad de un objeto

” Un objeto tiene un estado, comportamiento e identidad; la estructura y comportamiento de objetos similares están definidos en su clase común; los términos *instancia* y *objeto* son intercambiables”.

Estado

El estado de un objeto abarca todas las propiedades (normalmente estáticas) del mismo más los valores actuales (normalmente dinámicos) de cada una de esas propiedades.

Una propiedad es una característica inherente o distintiva, un rasgo o calidad que contribuye a hacer que ese objeto sea ese objeto y no otro. Las propiedades suelen ser estáticas, porque atributos como este suelen ser inmutables y fundamentales para la naturaleza del objeto. Se dice “suelen” porque en algunas circunstancias las propiedades de un objeto pueden cambiar.

Todas las propiedades tienen un valor. Este valor puede ser una mera cantidad o puede denotar a otro objeto. Así distinguimos a:

- Objetos: existen en el tiempo, son modificables, tienen estado, son instanciados y pueden crearse, destruirse y compartirse.
- Valores simples: son atemporales, inmutables y no instanciados.

El hecho que todo objeto tiene un estado implica que todo objeto toma cierta cantidad de espacio, ya sea del mundo físico o de la memoria de la computadora.

Cada parte de la estructura (registro de personal) denota una propiedad particular de la abstracción de un registro de personal. Esta declaración denota una clase, no un objeto, porque no representa una instancia específica.

Puede decirse que todos los objetos de un sistema encapsulan (protegen) algún estado, y que todo estado de un sistema está encapsulado en un objeto. Sin embargo, encapsular el estado de un objeto es un punto de partida, pero no es suficiente para permitir que se capturen todos los designios de las abstracciones que se descubren e inventan durante el desarrollo. Por esta razón hay que considerar también cómo se comportan los objetos.

Comportamiento

Ningún objeto existe en forma aislada, en vez de eso, los objetos reciben acciones, y ellos mismos actúan sobre otros objetos. Así puede decirse que:

“El comportamiento es cómo actúa y reacciona un objeto, en términos de sus cambios de estado y paso de mensajes”.

Dicho de otra forma: el comportamiento de un objeto representa su actividad visible y comprobable exteriormente.

Una operación en una acción que un objeto efectúa sobre otro con el fin de provocar una reacción. Las operaciones que los objetos cliente pueden realizar sobre un objeto, suelen declararse como métodos y forman parte de la declaración de la clase a la que pertenece ese objeto.

El paso de mensajes es una de las partes de la ecuación que define el comportamiento de un objeto; la definición de comportamiento también recoge que el estado de un objeto afecta asimismo a su comportamiento. Así, se puede decir, que el comportamiento de un objeto es función de su estado, así como de la operación que se realiza sobre él, teniendo algunas operaciones, el efecto colateral de modificar el estado del objeto. Refinando la definición de estado podemos decir:

“El estado de un objeto representa los resultados acumulados de su comportamiento.”

Operaciones: una operación denota un servicio que un objeto que pertenece a una clase ofrece a sus clientes. Los tres tipos más comunes de operaciones son los siguientes:

Modificador:	Una operación que altera el estado de un objeto
Selector:	Una operación que accede al estado del objeto, pero no lo altera.
Iterador:	Una operación que permite acceder a todos los objetos de una clase en algún orden, perfectamente establecido.
Hay otros dos tipos de operaciones habituales: representan la infraestructura necesaria para crear y destruir instancias de una clase, son:	
Constructor:	Una operación crea un objeto y/o inicializa su estado.
Destructor:	Una operación que libera el estado de un objeto y/o destruye el propio objeto.

Unificando las definiciones de estado y comportamiento se definen las responsabilidades de un objeto de forma tal que incluyan dos elementos clave: el conocimiento que un objeto mantiene y las acciones que puede llevar a cabo. Las responsabilidades están encaminadas a transmitir un sentido del propósito de un objeto y de su lugar en el sistema.

La responsabilidad de un objeto son todos los servicios que proporciona para todos los contratos que soporta.

En otras palabras, se puede decir que el estado y el comportamiento de un objeto definen un conjunto de roles que puede representar un objeto en el mundo, los cuales, a su vez, cumplen las responsabilidades de la abstracción.

Identidad

“La identidad es aquella propiedad de un objeto que lo distingue de todos los demás objetos”

La mayoría de los lenguajes de programación y de bases de datos utilizan nombres de variables para distinguir objetos, mezclando la posibilidad de acceder a ellos con su identidad. También utilizan claves de identificación para distinguir objetos, mezclando el valor de un dato con su identidad.

Un objeto dado puede nombrarse de más de una manera; a esto se lo denomina **compartición estructural**. En otras palabras, existen alias para el objeto que nos permiten llamarlo de diferentes formas en base a las necesidades de acceso que tengamos.

El tiempo de vida de un objeto se extiende desde el momento en que se crea por primera vez (y consume espacio por primera vez) hasta que se destruye. Para crear un objeto hay que declararlo o bien asignarle memoria dinámicamente; generalmente esto se realiza con la ayuda de los métodos **constructores y destructores** mencionados anteriormente.

La declaración e inicialización de objetos consume memoria RAM⁶, por lo que su administración es de suma importancia para garantizar el correcto funcionamiento del programa. Dependiendo del lenguaje de programación que utilicemos es posible que, para liberar memoria, una vez que un objeto deja de ser necesario, debamos destruirlo de forma explícita. En cambio, otros lenguajes de programación están provistos de herramientas que se encargan automáticamente de esta tarea, liberándonos de la responsabilidad de destruir los objetos.

Relaciones entre objetos

Los objetos contribuyen al comportamiento de un sistema colaborando con otros. La relación entre dos objetos cualesquiera abarca las suposiciones que cada uno realiza acerca del otro, incluyendo qué operaciones pueden realizarse y qué comportamiento se obtiene. Hay dos tipos de jerarquías de especial interés:

- Enlaces
- Agregación

Enlaces

Definido como “conexión física o conceptual entre objetos”. Un objeto colabora con otros a través de sus enlaces con éstos. Dicho de otra forma: un enlace denota la asociación específica por la cual un objeto (cliente) utiliza los servicios de otro objeto (servidor), o a través de la cual un objeto puede comunicarse con otro.

⁶ **Memoria RAM:** La memoria de acceso aleatorio (*Random Access Memory, RAM*), se utiliza como memoria de trabajo de computadoras para el sistema operativo, los programas y la mayor parte del software. En la RAM se cargan todas las instrucciones que ejecuta la unidad central de procesamiento (procesador) y otras unidades del computador.

Como participante de un enlace, un objeto puede desempeñar uno de tres roles:

- **Actor:** Un objeto puede operar sobre otros objetos, pidiéndole colaboración por medio de sus mensajes, pero nunca otros objetos operan sobre él, a veces, los términos *objeto activo* y *actor* son equivalentes.
- **Servidor:** Un objeto que nunca opera sobre otros, sólo otros objetos, operan sobre él.
- **Agente:** Un objeto puede operar sobre otros objetos, y además otros objetos operan sobre él. Un agente se crea normalmente para realizar algún trabajo en nombre de un actor u otro agente.

Visibilidad

La visibilidad es una propiedad que permite a un objeto operar sobre otro. Si un objeto no ve a otro, no puede enviarle un mensaje, para pedirle su colaboración.

Puede postergarse la definición de visibilidad de un objeto durante los momentos iniciales del análisis de un problema; no obstante, ni bien comienzan las actividades de diseño e implementación, hay que considerar la visibilidad a través de los enlaces porque las decisiones en este punto dictan el ámbito y acceso de los objetos a cada lado del enlace.

Agregación

Mientras que los enlaces denotan relaciones de igual a igual o de cliente/servidor, la agregación denota una jerarquía todo/parte, con la capacidad de ir desde el todo (agregado) hasta las partes. En este sentido la agregación es un tipo especial de relación.

La agregación puede o no denotar contención física. La relación todo/parte es más conceptual y por ende menos directa que la agregación física.

Existen claros pros y contras entre la agregación y los enlaces. La agregación es a veces mejor porque encapsula partes y secretos del todo. A veces, son mejores los enlaces porque permiten acoplamientos más débiles entre los objetos. Es necesario sopesar cuidadosamente ambos factores a la hora de elegir un tipo de relación entre los objetos.

Relaciones entre clases

Las clases, al igual que los objetos, no existen aisladamente. Para un dominio de problema específico, las abstracciones suelen estar relacionadas por vías muy diversas y formando una estructura de clases del diseño.

Se establecen relaciones entre dos clases por una de estas dos razones:

- una relación entre clases podría indicar algún tipo de compartición.
- una relación entre clases podría indicar algún tipo de conexión semántica.

En total existen tres tipos básicos de relaciones entre clases:

- **Generalización/Especificación:** denota un tipo de relación “**es un**” o mejor aún, “**se comporta como**”.
- **Todo/Parte:** denota una relación “**parte de**”.

- **Asociación:** que denota alguna dependencia semántica entre clases de otro modo independientes, muchas veces referenciada como hermano-hermano.

La mayoría de los lenguajes de programación orientados a objetos ofrecen soporte directo para alguna combinación de las siguientes relaciones:

- **Asociación**
- **Herencia**
- **Agregación**

De estos seis tipos de relaciones, las asociaciones son en general las más utilizadas, pero son las más débiles semánticamente. La identificación de asociaciones se realiza en el análisis y diseño inicial, momento en el que se comienzan a descubrir las dependencias generales entre las abstracciones. Al avanzar en el diseño y la implementación se refinan a menudo estas asociaciones orientándolas hacia una de las otras relaciones de clases más específicas y refinadas.

Asociación

Dependencias semánticas: una asociación denota una dependencia semántica y puede establecer o no, la dirección de esta dependencia, conocida como navegabilidad. Si se establece la dirección de la dependencia o navegabilidad, debe nombrarse el rol que desempeña la clase en relación con la otra. Esto es suficiente durante el análisis de un problema, donde sólo es necesario identificar esas dependencias.

Multiplicidad: existen tres tipos habituales de multiplicidad en una asociación:

- Uno a uno
- Uno a muchos
- Muchos a muchos

Herencia

Las personas tendemos a asimilar nuevos conceptos basándonos en lo que ya conocemos. “*La herencia es una herramienta que permite definir nuevas clases en base a otras clases*”.

La herencia es una relación entre clases, en la que una clase comparte la estructura y/o el comportamiento definidos en una (herencia simple) o más clases (herencia múltiple). La clase de la que otras heredan se denomina “superclase”, o “clase padre”. La clase que hereda de una o más clases se denomina “subclase” o “clase hija”. La herencia define una jerarquía de “tipos” entre clases, en la que una subclase hereda de una o más superclases.

Una subclase habitualmente aumenta o restringe la estructura y el comportamiento existentes en sus superclases. Una subclase que aumenta una superclase se dice que utiliza herencia por extensión. Una subclase que restringe el comportamiento de sus superclases se dice que utiliza herencia por restricción. En la práctica no es muy claro cuando se usa cada caso; en general, es habitual que hagan las dos cosas.

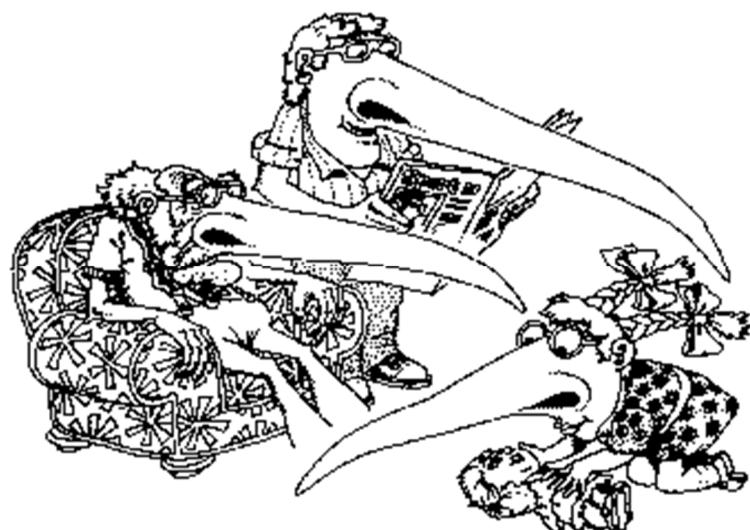


Fig. 5 - Herencia entre Clases

Algunas de las clases tendrán instancias, es decir se crearán objetos a partir de ellas, y otras no. Las clases que tienen instancias se llaman *concretas* y las clases sin instancias son *abstractas*. Una clase abstracta se crea con la idea de que sus subclases añadan elementos a su estructura y comportamiento, usualmente completando la implementación de sus métodos (habitualmente) incompletos, con la intención de favorecer la reutilización⁷.

En una estructura de clases, las clases más generalizadas de abstracciones del dominio se denominan *clases base*.

Una clase cualquiera tiene típicamente dos tipos de “clientes”:

- Instancias
- Subclases

Resulta útil definir interfaces distintas para estos dos tipos de clientes. En particular, se desea exponer a los clientes instancia sólo los comportamientos visibles exteriormente, pero se necesita exponer las funciones de asistencia y las representaciones únicamente a los clientes subclase.

El uso de la herencia expone algunos de los secretos de la clase heredada. En la práctica, esto implica que, para comprender el significado de una clase particular, muchas veces hay que estudiar todas sus superclases, a veces incluyendo sus vistas internas.

Polimorfismo

El **polimorfismo** es un concepto aplicado en el contexto de la teoría de tipos, como en las relaciones de herencia o generalización, donde el nombre de un método puede implicar comportamientos diferentes, conforme esté especificado en clases diferentes, en tanto y en cuanto estas clases estén relacionadas

⁷ **Reutilización:** volver a utilizar algo. En el contexto del software es la acción que permite evitar la duplicación y ahorrar esfuerzo de definición e implementación, ubicando propiedades y/o comportamiento en un único lugar y accediendo a él cuando se lo requiera.

por una superclase común. Un objeto puede responder de diversas formas a un mismo comportamiento, denotado por este nombre, dependiendo de la clase a la que pertenezca.

El polimorfismo es la propiedad que potencia el uso de la herencia y para poder utilizarla, las clases deben especificar los comportamientos con los mismos protocolos. La herencia sin polimorfismo es posible, pero no es muy útil.

La herencia puede utilizarse como:

- Una decisión privada del diseñador para “reusar” código porque es útil hacerlo; debería ser posible cambiar esta decisión con facilidad.
- La realización de una declaración pública de que los objetos de la clase hija obedecen a la semántica de la clase padre, de modo que la clase hija implementa, especializa o refina la clase padre.

Con herencia simple, cada subclase tiene exactamente una superclase; aunque la herencia simple es muy útil, frecuentemente fuerza al programador a derivar de una entre dos clases igualmente atractivas. Esto limita la aplicabilidad de las clases predefinidas, haciendo muchas veces necesario duplicar el código. La herencia múltiple es bastante discutida aún, y permite que una clase hija puede heredar de más de una clase padre. Podría decirse que es como un paracaídas: no siempre se necesita, pero cuando así ocurre, uno está verdaderamente feliz de tenerlo a mano. Distintos lenguajes de programación posibilitan o no este tipo de herencia.

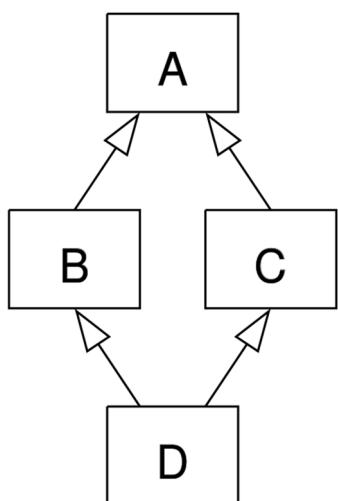


Fig. 6 Representación de la herencia de diamante

En el contexto de la herencia múltiple, se conoce como problema del diamante a una ambigüedad que surge cuando dos clases B y C heredan de A, y la clase D hereda de B y C. Si un método en D llama a un método definido en A, ¿por qué clase lo hereda, B o C?

Se llama el problema del 'diamante' por la forma del diagrama de herencia de clase en esta situación. La clase A está arriba, B y C están separadas debajo de ella, y D se une a las dos en la parte inferior consiguiendo la forma de un diamante.

El ornitorrinco, que deriva de mamífero y ovíparo. Las dos clases derivan de animal.

Agregación

Las relaciones de agregación entre clases tienen un paralelismo directo con las relaciones de agregación entre los objetos correspondientes a esas clases. La agregación establece una dirección en la relación todo/parte. Pueden identificarse dos tipos de contención.

- Contención física también conocida como composición: es una contención por valor, lo que significa que el objeto no existe independientemente de la instancia que lo encierra. El tiempo de vida de ambos objetos está en íntima conexión.

- Contención por referencia: es un tipo menos directo de agregación, en este caso se pueden crear y destruir instancias de cada objeto en forma independiente.

La contención por valor no puede ser cíclica, es decir que ambos objetos no pueden ser físicamente partes de otro. En el caso de la contención por referencia puede serlo (cada objeto puede tener un puntero apuntando al otro).

No hay que olvidar que la agregación no precisa contención física. Podemos afirmar que: si y solo si existe una relación todo/parte entre dos objetos, podremos tener una relación de agregación entre sus partes correspondientes.

El Modelo de Objetos

No hay un estilo de programación que sea el mejor para todo tipo de aplicaciones. El estilo orientado a objetos es el más adecuado para el más amplio conjunto de aplicaciones; realmente, este paradigma de programación sirve con frecuencia como el marco de referencia arquitectónico en el que se emplean otros paradigmas.

Cada uno de estos estilos de programación se basa en su propio marco de referencia conceptual. Cada uno requiere una actitud mental diferente, una forma distinta de pensar en el problema. El modelo de objetos es el marco de referencia conceptual para evaluar si se respeta el paradigma orientado a objetos. Hay cuatro elementos fundamentales en el modelo de objetos:

- Abstracción
- Encapsulamiento
- Modularidad
- Jerarquía

Al decir fundamentales, quiere decirse que un modelo que carezca de cualquiera de estos elementos no es orientado a objetos.

Hay tres elementos secundarios del Modelo de objetos:

- Tipos (tipificación)
- Concurrencia
- Persistencia

Por secundarios quiere decirse que cada uno de ellos es una parte útil del modelo de objetos, pero no esencial.

Abstracción

Es una de las vías fundamentales por la que los humanos combatimos la complejidad. La definimos de la siguiente forma:

“Una abstracción denota las características esenciales de un objeto, que lo distingue de todos los demás y proporciona así fronteras conceptuales nítidamente definidas respecto a la perspectiva del observador.”



Fig. 7 - Abstracción en el Modelo de Objetos

Una abstracción se centra en la visión externa de un objeto por lo tanto sirve para separar el comportamiento esencial de un objeto de su implementación. La decisión sobre el conjunto adecuado de abstracciones para determinado dominio es el problema central del diseño orientado a objetos.

Se puede caracterizar el comportamiento de un objeto de acuerdo a los servicios que presta a otros objetos, así como las operaciones que puede realizar sobre otros objetos. Este punto de vista obliga a concentrarse en la visión exterior del objeto que define un contrato del que pueden depender otros objetos y que a su vez debe ser llevado a cabo por la vista interior del propio objeto (a menudo en colaboración con otros objetos). Este contrato establece todas las suposiciones que puede hacer el objeto cliente del comportamiento de un objeto servidor, es decir las responsabilidades del objeto.

Todas las abstracciones tienen propiedades tanto estáticas como dinámicas. En un estilo de programación orientado a objetos ocurren cosas cuando se opera sobre un objeto. Las operaciones significativas que pueden realizarse sobre un objeto y las reacciones de este ante ellas constituyen *el comportamiento completo del objeto*.

Identificación de las abstracciones clave

Una abstracción clave es una clase u objeto que forma parte del vocabulario del dominio del problema. El valor principal que tiene la identificación de tales abstracciones es que dan unos límites al problema; enfatizan las cosas que están en el sistema y, por tanto, son relevantes para el diseño y suprimen las cosas que están fuera del sistema y, por lo tanto, son superfluas. La identificación de abstracciones clave es altamente específica de cada dominio.

La identificación de las abstracciones clave conlleva dos procesos: **descubrimiento** e **invención**. Mediante el **descubrimiento**, se llega a reconocer las abstracciones utilizadas por expertos del dominio; si el experto del dominio habla de ella, entonces la abstracción suele ser importante. Mediante la **invención**, se crean nuevas clases y objetos que no son forzosamente parte del dominio del problema, pero son artefactos útiles el diseño o la implantación, por ejemplo: bases de datos, manejadores, pantallas, manejadores de impresoras y demás. Estas abstracciones clave son artefactos del diseño particular, denominados comúnmente de “fabricación pura” y forman parte del dominio de la solución. Una vez que se identifica determinada abstracción clave como candidata, hay que evaluarla. Se debe centrar en las preguntas:

- ¿Cómo se crean los objetos de esta clase?
- ¿Pueden los objetos de esta clase copiarse y/o destruirse?
- ¿Qué operaciones pueden hacerse en esos objetos?

Si no hay buenas respuestas a tales preguntas, el concepto probablemente no estaba "limpio" desde el principio.

La definición de clases y objetos en los niveles correctos de abstracción es difícil. A veces se puede encontrar una subclase general, y elegir moverla hacia arriba en la estructura de clases, incrementando así el grado de compartición. Esto se llama promoción de clases. Análogamente, se puede apreciar que una clase es demasiado general, dificultando así la herencia por las subclases a causa de un vacío semántico grande. Esto recibe el nombre de conflicto de granularidad. En ambos casos, se intenta identificar abstracciones cohesivas y débilmente acopladas, para mitigar estas dos situaciones.

La mayoría de los desarrolladores suele tomarse a la ligera la actividad de dar un nombre correcto a las cosas -de forma que reflejen su semántica, a pesar de que es importante en la captura de la esencia de las abstracciones que se describen. El software debería escribirse tan cuidadosamente como la prosa en español, con consideración tanto hacia el lector como hacia la computadora.



Fig. 8- Las clases y objetos deberían estar al nivel de abstracción adecuado: ni demasiado alto ni demasiado bajo

Encapsulamiento

El encapsulamiento oculta los detalles de implementación de un objeto. También conocido como encapsulación, se lo define de la siguiente forma:

"El encapsulamiento es el proceso de almacenar en un mismo compartimento los elementos de una abstracción, que constituyen su estructura y su comportamiento; sirve para separar la interfaz contractual de una abstracción y su implantación".

Se llama a esos elementos encapsulados secretos de una abstracción.

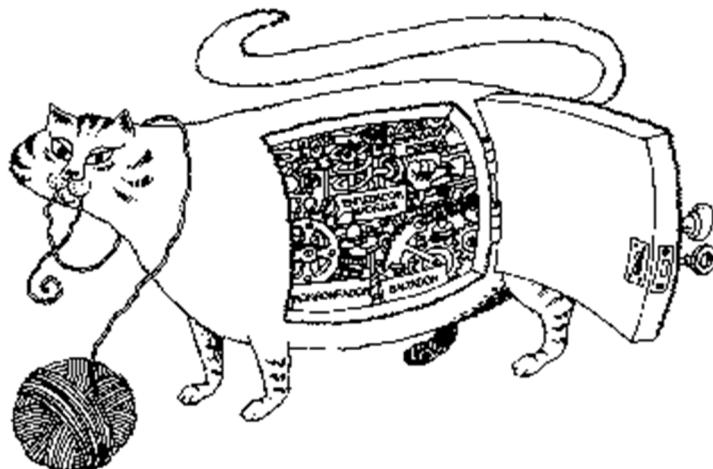


Fig. 9 - El encapsulamiento en el Modelo de Objetos

La abstracción de un objeto debe preceder a las decisiones sobre su implementación. Esta implantación debe tratarse como un secreto de la abstracción, oculto para la mayoría de los clientes.

Ninguna parte de un sistema complejo debe depender de los detalles internos de otras partes; mientras que la abstracción ayuda a las personas a pensar sobre lo que están haciendo, el encapsulamiento permite que los cambios hechos en los programas sean fiables con el menor esfuerzo.

La abstracción y el encapsulamiento son conceptos complementarios: la primera se centra en el comportamiento observable de un objeto, mientras el encapsulamiento se centra en la implementación que da lugar a este comportamiento. El encapsulamiento se consigue a menudo ocultando todos los secretos de un objeto que no constituyen sus características esenciales; típicamente: la estructura de un objeto está oculta, así como la implementación de sus métodos.

El encapsulamiento proporciona barreras explícitas entre abstracciones diferentes y por tanto conduce a una clara separación de intereses.

Una clase debe tener dos partes: una *interfaz* (captura solo su vista externa, abarcando a la abstracción que se ha hecho del comportamiento común de las instancias de la clase) y una *implementación* (comprende la representación de la abstracción, así como los mecanismos que consiguen el comportamiento deseado). La interfaz de una clase es el único lugar donde se declaran todas las suposiciones que un cliente puede hacer acerca de todas las instancias de una clase; la implementación encapsula detalles acerca de los cuales ningún cliente puede realizar suposiciones.

Modularidad

Como vimos anteriormente en la unidad de diseño de algoritmos, el acto de fragmentar un programa en componentes individuales puede reducir su complejidad en algún grado.

“La modularidad es la propiedad que tiene un sistema que ha sido descompuesto en un conjunto de módulos cohesivos y débilmente acoplados.”

En la definición anterior *cohesión* se refiere al grado de relación que tienen los componentes que conforman el sistema, es decir en qué grado comparten propiedades comunes. Lo esperado es que cada componente esté formado por elementos altamente relacionados entre sí. Por otro lado, el *acoplamiento* se refiere al grado de relaciones que tiene el componente con otros componentes o

cuanto depende de otros para poder realizar sus tareas, en este caso esperamos que los componentes del sistema tengan un grado de acoplamiento bajo, lo que hace más fácil la tarea de mantener cada componente de forma individual.



Fig. 10 - Modularidad, cohesión y acoplamiento en el Modelo de Objetos

Aunque la fragmentación es útil por reducir la complejidad, existe una razón más poderosa: es que la fragmentación crea una serie de fronteras bien definidas y documentadas dentro del programa. Estas fronteras tienen un valor incalculable a la hora de la compresión del programa.

La modularización consiste en dividir un programa en módulos que puedan compilarse separadamente, pero que tienen conexiones con otros módulos. Las conexiones entre módulos son las suposiciones que cada módulo hace acerca de todos los demás.

La mayoría de los lenguajes que soportan el concepto de módulo, también hacen la distinción entre la interfaz de un módulo y su implementación. Así es correcto decir que la modularización y el encapsulamiento van de la mano.

Los módulos sirven como contenedores físicos en los que se declaran las clases y los objetos del diseño lógico realizado. La decisión sobre el conjunto adecuado de módulos para determinado problema es un problema casi tan difícil de decidir cómo decidir sobre el conjunto adecuado de abstracciones. La modularización arbitraria puede ser a veces peor que la ausencia de modularización.

Existen varias técnicas útiles, así como líneas generales no técnicas, que pueden ayudar a conseguir una modularización inteligente de clases y objetos. El objetivo de fondo de la modularización es la reducción del costo del software al permitir que los módulos se diseñen y revisen independientemente. La estructura de un módulo debería:

- Ser lo bastante simple como para ser comprendida en su totalidad
- Ser posible cambiar la implantación de los módulos sin saber nada de los otros módulos y sin afectar su comportamiento.
- Y la facilidad de realizar un cambio debería guardar una relación razonable con la probabilidad de que ese cambio fuese necesario.

Existe un límite pragmático a estas líneas; en la práctica el costo de recompilar el cuerpo de un módulo es relativamente bajo, mientras que el costo de recompilar la interfaz es bastante alto. Por esta razón, la interfaz de un módulo debería ser tan estrecha como fuera posible, siempre y cuando se satisfagan las necesidades de todos los módulos que lo utilizan.

El desarrollador debe equilibrar dos intereses técnicos rivales:

1. El deseo de encapsular abstracciones.
2. La necesidad de hacer ciertas abstracciones visibles para otros módulos.

He aquí el siguiente consejo:

“Los detalles de un sistema, que probablemente cambien de forma independiente, deberían ser secretos en módulos separados; las únicas suposiciones que deberían darse entre módulos son aquellas cuyo cambio se considera improbable. Toda estructura de datos es privada a algún módulo; a ella pueden acceder uno o más programas del módulo, pero no programas de fuera del módulo. Cualquier otro programa que requiera información almacenada en los datos de un módulo debería obtenerla llamando a programas de éste”.

Es decir, hay que construir módulos cohesivos (agrupando abstracciones que guarden cierta relación lógica) y débilmente acoplados (minimizando la dependencia entre módulos).

Los principios de abstracción, encapsulamiento y modularidad son sinérgicos. Un objeto proporciona una frontera bien definida alrededor de una abstracción, tanto el encapsulamiento como la modularidad proporcionan barreras que rodean a esta abstracción.

Hay dos problemas técnicos más que pueden afectar la decisión de modularización:

1. Empaquetar clases y objetos en módulos de forma que su reutilización fuese conveniente, puesto que los módulos suelen ser unidades de software elementales e indivisibles.
2. Debe haber límites prácticos al tamaño de un módulo individual, por cuestiones de compilación y manejo de memoria virtual.

Es difícil conjugar todos los requisitos, pero no hay que perder de vista la cuestión más importante: encontrar las clases y objetos correctos y organizarlos después en módulos separados son decisiones de diseño ampliamente independientes. La identificación de clases y objetos es parte del diseño lógico de un sistema, pero la identificación de los módulos es parte del diseño físico del mismo. No pueden tomarse todas las decisiones del diseño lógico antes que las del físico ni viceversa: estas decisiones se dan en forma iterativa.

Jerarquía

“La jerarquía es una clasificación u ordenación de abstracciones”.

La abstracción es algo bueno, pero la mayoría de las veces hay muchas más abstracciones diferentes de las que se pueden comprender simultáneamente. El encapsulamiento ayuda a manejar esta complejidad ocultando la visión interna de estas abstracciones. La modularidad también ayuda, ofreciendo una vía para agrupar abstracciones relacionadas lógicamente. Esto no es suficiente. Frecuentemente un conjunto de abstracciones forma una jerarquía y la identificación de esas jerarquías en el diseño simplifica en gran medida la comprensión del problema.

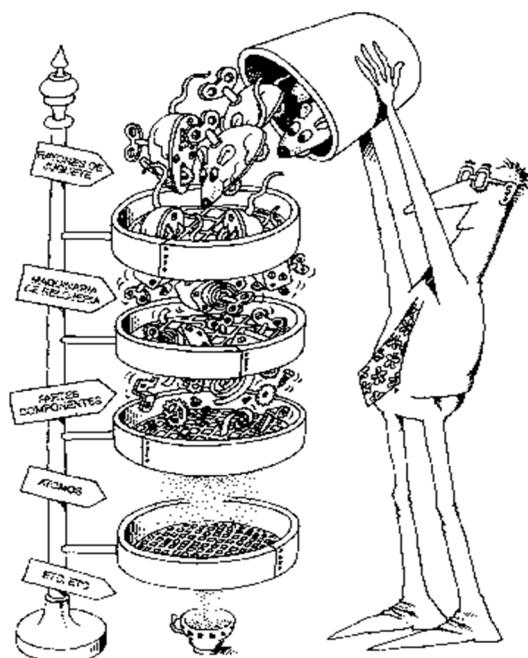


Fig. 11 - Las Abstracciones forman una Jerarquía

Las dos jerarquías más importantes en un sistema complejo son

- Su estructura de clases (la jerarquía <<de clases>>).
- Su estructura de partes (la jerarquía <<de partes>>).

Existe una conveniente tensión entre los principios de abstracción, encapsulamiento y jerarquía. “El encapsulamiento intenta proporcionar una barrera opaca tras la cual se ocultan los métodos y el estado de la abstracción; la herencia requiere abrir esta interfaz en cierto grado y permitir el acceso a los métodos y estado sin encapsulamiento.”

Existen, para una clase dada, dos tipos de clientes:

1. Objetos que invocan operaciones sobre instancias de la clase.
2. Subclases que heredan de esta clase.

Como vimos anteriormente, con la herencia se puede violar el encapsulamiento de tres formas:

1. La subclase podría acceder a una variable de instancia de su superclase.
2. La subclase podría llamar a una operación privada de su superclase.
3. La subclase podría referenciar directamente a superclases de su superclase.

Mientras las jerarquías <<es un>> denotan relaciones de generalización/especialización, las jerarquías <<parte de>> describen relaciones de agregación. La combinación de herencia con agregación es muy potente: la agregación permite el agrupamiento físico de estructuras relacionadas lógicamente, y la herencia permite que estos grupos de aparición frecuente se reutilicen con facilidad en diferentes abstracciones.

Tipos (Tipificación)

“Los tipos son la puesta en vigor de la clase de objetos, de modo que los objetos de tipos distintos no pueden intercambiarse o, como mucho, pueden intercambiarse sólo de formas muy restringidas”.

El concepto de tipos se deriva de las teorías sobre tipos abstractos de datos. “Un tipo es una caracterización precisa de propiedades estructurales o de comportamiento que comparten una serie de entidades”.

Para nuestro propósito se usarán los términos **tipo** y **clase** en forma intercambiable. Los tipos permiten expresar las abstracciones de manera que el lenguaje de programación en el que se implantan puede utilizarse para apoyar las decisiones de diseño.

La idea de congruencia es central a la noción de tipos. (ej. Las unidades de medida en física: al dividir distancia con tiempo, se espera algún valor que denote velocidad, no peso). Este es ejemplo de comprobación estricta de tipos, en los que las reglas del dominio dictan y refuerzan ciertas combinaciones correctas entre las abstracciones. Los lenguajes de programación con comprobación estricta de tipos detectan cualquier violación en tiempo de compilación como en Java, mientras que en lenguajes como Smalltalk (sin tipos) estos errores suelen manifestarse en tiempo de ejecución.

La comprobación estricta de tipos es particularmente relevante a medida que aumenta la complejidad del sistema. Sin embargo, tiene un lado oscuro, en la práctica, introduce dependencias semánticas tales que incluso cambios pequeños en la interfaz requieren recopilaciones de todas las subclases.



Fig. 12- La comprobación estricta de tipos impide que se mezclen abstracciones

Existen varios beneficios importantes que se derivan del uso de lenguajes de tipos estrictos:

- Sin la comprobación de tipos un lenguaje puede “estallar” de forma misteriosa en ejecución en la mayoría de los lenguajes.
- En la mayoría de los sistemas, el ciclo **editar-compilar-depurar** es tan tedioso que la detección temprana de errores es indispensable.
- La declaración de tipos ayuda a documentar programas.
- La mayoría de los compiladores pueden generar código más eficiente si se han declarado los tipos.

- Los lenguajes sin tipos ofrecen mayor flexibilidad, pero la seguridad que ofrecen los lenguajes con tipos estrictos suele compensar la flexibilidad que se pierde.

Como mencionamos anteriormente, el polimorfismo representa un concepto de la teoría de tipos en el que un solo nombre puede denotar objetos de muchas clases diferentes que se relacionan por una superclase común. Cualquier objeto denotado por ese nombre es capaz de responder a algún conjunto común de operaciones definidas por su interfaz. El polimorfismo es quizás la característica más potente de los lenguajes orientados a objetos después de la capacidad de soportar la abstracción.

Concurrencia

“La concurrencia es la propiedad que distingue un objeto activo de uno que no está activo”.

Un sistema automatizado puede tener que manejar muchos eventos diferentes simultáneamente. Para esos casos es útil considerar utilizar un conjunto de procesadores o procesadores que soporten la multitarea. Un solo proceso denominado hilo de control es la raíz a partir de la cual se realizan acciones dinámicas independientes dentro del sistema. Todo programa tiene al menos un hilo de control, pero un sistema que implique concurrencia puede tener muchos hilos: algunos son transitorios y otros permanentes.

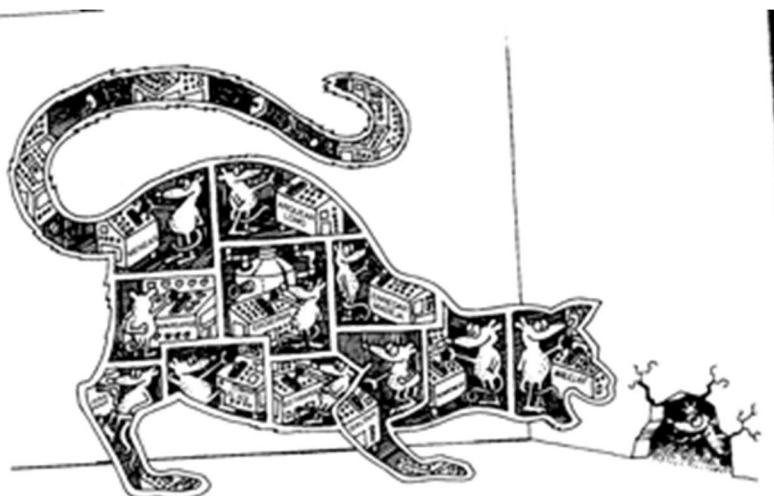


Fig. 13 - La concurrencia permite a dos objetos actuar al mismo tiempo

Mientras que la Programación Orientada a Objetos (POO) se centra en la abstracción de datos, encapsulamiento y herencia, la concurrencia se centra en la abstracción de procesos y la sincronización. Cada objeto puede representar un hilo separado de control (una abstracción de un proceso). Tales objetos se llaman **activos**. En un sistema basado en diseño orientado a objetos, se puede conceptualizar al mundo con un conjunto de objetos cooperativos, algunos de los cuales son **activos** y sirven, así como centros de actividad independiente.

En general existen tres enfoques para la concurrencia en el diseño orientado a objetos:

1. La concurrencia es una característica intrínseca de ciertos lenguajes de programación.

2. Se puede usar una biblioteca de clases que soporte alguna forma de procesos ligeros. La implementación de esta biblioteca es altamente dependiente de la plataforma, aunque la interfaz es relativamente transportable.
3. Pueden utilizarse interrupciones para dar la ilusión de concurrencia. Esto exige conocer detalles de bajo nivel del hardware.

No importa cuál de ellas se utilice; una vez que se introduce la concurrencia en un sistema hay que considerar como los objetos activos **sincronizan** sus actividades con otros. Este es el punto en el que las ideas de abstracción, encapsulamiento y concurrencia interactúan. En presencia de la concurrencia no basta con definir los métodos de un objeto; hay que asegurarse que la semántica de estos métodos se mantiene a pesar de la existencia de múltiples hilos de control.

Persistencia

“La persistencia es la propiedad de un objeto por la que su existencia trasciende el tiempo (es decir, el objeto continúa existiendo después de que su creador deja de existir) y/o el espacio (es decir, la posición del objeto varía con respecto al espacio de direcciones en el que fue creado)”.

Un objeto de software ocupa una cierta cantidad de espacio en memoria y existe durante cierta cantidad de tiempo. Este espectro de persistencia de objetos abarca lo siguiente:

- Resultados transitorios en la evaluación de expresiones.
- Variables locales en la activación de procedimientos.
- Variables propias, variables locales y elementos del montículo cuya duración difiere de su ámbito.
- Datos que existen entre ejecuciones de un programa.
- Datos que existen entre varias versiones de un programa.
- Datos que sobreviven al programa.



Fig. 14 - Conserva el estado de un objeto en el tiempo y el espacio

Los lenguajes de programación tradicionales suelen tratar los tres primeros tipos de persistencia de objetos; la persistencia de los tres últimos tipos pertenece al dominio de la tecnología de bases de datos, lo que veremos en mayor profundidad en el módulo “Bases de Datos”.

Muy pocos lenguajes ofrecen soporte para la persistencia (Smalltalk es una excepción). Frecuentemente se consigue a través de bases de datos orientadas a objetos disponibles comercialmente.

La persistencia abarca algo más que la mera duración de los datos, no sólo persiste el estado de un objeto, sino que su clase debe trascender también a cualquier programa individual, de forma que todos los programas interpreten de la misma manera el estado almacenado.

La discusión hasta aquí afecta la persistencia en el tiempo; sin embargo, para sistemas que se ejecutan en un conjunto distribuido de procesadores, a veces, hay que preocuparse de la persistencia en el espacio. En estos casos es útil pensar en aquellos objetos que puedan llevarse de una máquina a otra, y que incluso puedan tener representaciones diferentes en máquinas diferentes.

Clasificación

La clasificación es el medio por el que ordenamos, el conocimiento ubicado en las abstracciones.

En el diseño orientado a objetos, el reconocimiento de la similitud entre las cosas nos permite exponer lo que tienen en común abstracciones clave y mecanismos; y eventualmente nos lleva a arquitecturas más pequeñas y simples. Desgraciadamente, no existe un camino trillado hacia la clasificación. Para el lector acostumbrado a encontrar respuestas en forma de receta, afirmamos inequívocamente que no hay recetas fáciles para identificar clases y objetos. No existe algo que podamos llamar una estructura de clases <>perfecta<>, ni el conjunto de objetos «correcto». Al igual que en cualquier disciplina de ingeniería, nuestras elecciones de diseño son un compromiso conformado por muchos factores que compiten.



Fig.15- La clasificación es el medio por el cual ordenamos el conocimiento

No es de extrañar que la clasificación sea relevante para todos los aspectos del diseño orientado a objetos:

- La clasificación ayuda a identificar jerarquías de **generalización-especialización** y **agregación** entre clases.

- La clasificación también proporciona una guía para tomar decisiones sobre **modularización**. Se puede decidir ubicar ciertas clases y objetos juntos o en módulos diferentes, dependiendo de su similitud. El *acoplamiento* y la *cohesión* son simplemente medidas de esta similitud.
- La clasificación desempeña un papel en la asignación de **procesos** a los procesadores. Se ubican ciertos procesos juntos en el mismo procesador o en diferentes procesadores, dependiendo de intereses de empaquetamiento, eficacia o fiabilidad.

Además, criterios diferentes para clasificar los mismos objetos arrojan resultados distintos. La clasificación es altamente dependiente de la razón por la que se clasifica.

La lección aquí es simple: como afirma Descartes, «el descubrimiento de un orden no es tarea fácil; sin embargo, una vez que se ha descubierto el orden, no hay dificultad alguna en comprenderlo». Los mejores diseños de software parecen simples, pero, como muestra la experiencia, exige gran cantidad de esfuerzo el diseño de una arquitectura simple.

¿Por qué, entonces, es tan difícil la clasificación? Existen dos razones importantes:

Primero: no existe algo que pueda llamarse una clasificación “perfecta”, aunque por supuesto, algunas clasificaciones son mejores que otras. Cualquier clasificación es relativa a la perspectiva del observador que la realiza.

Segundo: la clasificación inteligente requiere una tremenda cantidad de perspicacia creativa. “A veces la respuesta es evidente, a veces es una cuestión de gustos, y otras veces, la selección de componentes adecuados es un punto crucial del análisis”.



Fig. 16 - Diferentes observadores pueden clasificar el mismo objeto de distintas formas

Algunos enfoques de Clasificación

Existen varias clasificaciones sugeridas por autores que proponen fuentes de obtención de clases y objetos, derivadas de los requerimientos del dominio del problema.

Por ejemplo, un enfoque sugiere que, para identificar clases del dominio del problema, las candidatas, provienen habitualmente de una de las siguientes fuentes:

Roles desempeñados por Personas	Humanos que llevan a cabo alguna función. Ejemplo: Madre, profesor, político.
Lugares	Denotan espacios geográficos o elementos para contención de casas. Ejemplo: País, Barrio, Estantería.
Cosas	Objetos físicos, o grupos de objetos, que son tangibles. Ejemplo: Vehículos, libros, sensores de temperatura
Roles desempeñados Organizaciones	Colecciones formalmente organizadas de personas, recursos, instalaciones y posibilidades que tienen una misión definida, cuya existencia es, en gran medida, independiente de los individuos. Ejemplo: Banco, Ministerio, Administradora de Tarjeta de Crédito
Conceptos	Principios o ideas no tangibles, utilizados para organizar o llevar cuenta de actividades de negocios y/o comunicaciones. Ejemplo: Materia, Carrera, Tratamiento Médico.
Eventos	Cosas que suceden, que habitualmente vinculan a clases de alguna de las otras categorías, en una fecha y/u horas concretas, o como pasos dentro de una secuencia ordenada. Ejemplo: Aterrizaje, Inscripción, Venta, Reserva, Donación.

Tabla 1 – Ejemplo de Clasificación de clases del dominio del problema

Identificación de mecanismos

Un mecanismo es cualquier estructura mediante la cual los objetos colaboran para proporcionar algún comportamiento que satisface un requerimiento del problema. Mientras el diseño de una clase incorpora el conocimiento de cómo se comportan los objetos individuales, un mecanismo es una decisión de diseño sobre cómo cooperan colecciones de objetos. Los mecanismos representan así patrones de comportamiento.

El mecanismo que elige un desarrollador entre un conjunto de alternativas es frecuentemente el resultado de otros de factores, como costo, fiabilidad, facilidad de fabricación y seguridad.

Del mismo modo que es una falta de educación el que un cliente viole la interfaz con otro objeto, tampoco es aceptable que los objetos transgredan los límites de las reglas de comportamiento dictadas por un mecanismo particular.

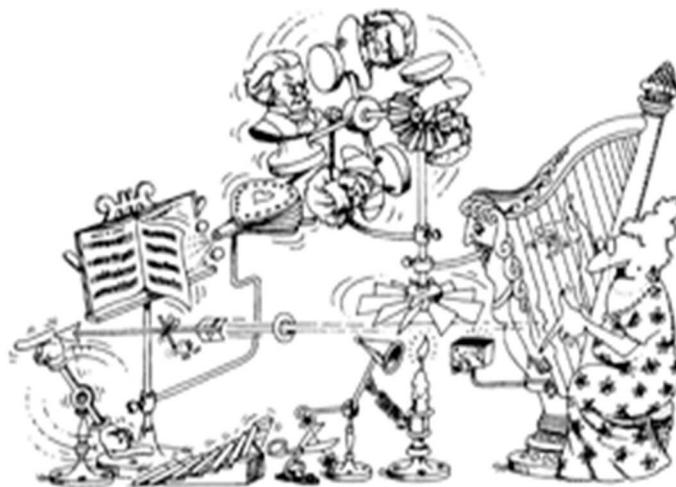


Fig. 17 - Los mecanismos son los medios por los cuales los objetos colaboran para proporcionar algún comportamiento de nivel superior

Mientras las **abstracciones clave** vistas anteriormente reflejan el vocabulario del dominio del problema, los **mecanismos** son el alma del diseño. Durante el proceso de diseño, el desarrollador debe considerar no sólo el diseño de clases individuales, sino también cómo trabajan juntas las instancias de esas clases. Una vez más, el uso de escenarios dirige este proceso de análisis. Una vez que un desarrollador decide sobre un patrón concreto de colaboración, se distribuye el trabajo entre muchos objetos definiendo métodos convenientes en las clases apropiadas. En última instancia, el protocolo de una clase individual abarca todas las operaciones que se requieren para implantar todo el comportamiento y todos los mecanismos asociados con cada una de sus instancias. Los mecanismos representan así decisiones de diseño estratégicas, como el diseño de una estructura de clases.

Los mecanismos son realmente uno de entre la variedad de patrones que se encuentran en los sistemas de software bien estructurados.

El modelado en el Desarrollo de Software

El éxito en el desarrollo de software está dado en la medida que se pueda satisfacer a las necesidades de los usuarios con un producto de calidad. Este mensaje conlleva un importante mensaje: el producto principal del trabajo de un equipo de desarrollo no son bonitos documentos, ni reuniones importantes, ni reportes de avance. Más bien, es un buen software que satisface las necesidades cambiantes de sus usuarios y demás afectados, todo lo demás es secundario.

No obstante, no se debe confundir “secundario” con “irrelevante”. Para producir software de calidad, es necesario involucrar a los usuarios para poder obtener un conjunto de requerimientos. Para desarrollar software de calidad, debemos definir una arquitectura sólida que le de soporte y sea permeable a los cambios. Para desarrollar software rápida, efectiva y eficientemente, con el mínimo de desperdicios y re-trabajo, hay que contar con el equipo de desarrollo adecuado, herramientas y el enfoque apropiado.

¿Qué es un modelo?

Un modelo es una simplificación de la realidad

Un modelo proporciona los planos del sistema. Estos planos pueden ser detallados o generales. Un buen modelo incluye aquellos elementos que tienen una gran influencia y omite aquellos elementos menores, que no son relevantes para el nivel de abstracción que se desea mostrar.

Todo sistema debe ser descripto desde distintas perspectivas utilizando distintos modelos. En el software, un modelo puede de *estructura*, destacando los elementos que organizan el sistema o de *comportamiento*, resaltando aspectos funcionales y/o de interacción dinámica.

La importancia de modelar

Construimos modelos para comprender mejor el sistema que estamos desarrollando. Por medio del modelado conseguimos cuatro objetivos:

1. Los modelos nos ayudan a visualizar cómo es o queremos que sea un sistema.
2. Los modelos nos permiten especificar la estructura o el comportamiento de un sistema.
3. Los modelos proporcionan plantillas que nos guían en la construcción de un sistema.
4. Los modelos documentan las decisiones que hemos adoptado.

Los modelos son útiles siempre, no sólo para sistemas grandes. No obstante, es cierto que mientras más grande y complejo sea el sistema el modelado se hace más importante, puesto que *construimos modelos de sistemas complejos porque no podemos comprender el sistema en su totalidad*. Esto es así porque la capacidad humana de comprender la realidad tiene límites y con el modelado se reduce el problema, centrándose en un aspecto a la vez.

Principios de modelado

El uso de modelos tiene una historia interesante en todas las disciplinas de ingeniería. Esa experiencia sugiere 4 principios básicos de modelado:

1. La elección de qué modelos crear tiene una profunda influencia sobre cómo se acomete un problema y cómo se da forma a su solución.
2. Todo modelo puede ser expresado a diferentes niveles de precisión.
3. Los mejores modelos están ligados a la realidad.
4. Un único modelo no es suficiente. Cualquier sistema no trivial se aborda mejor a través de un pequeño conjunto de modelos casi independientes.

Advertencias en el Modelado

A pesar de las ventajas obvias del modelado existe un factor importante que ejerce influencia en el valor de un modelo: la calidad del modelo en sí mismo. Si la abstracción es incorrecta, entonces obviamente la realidad creada eventualmente fuera de la abstracción, probablemente será incorrecta. Una abstracción incorrecta, no refleja o representa verdaderamente la realidad. Por lo tanto, la calidad del modelo es de inmensa importancia para derivar beneficios de calidad.

El modelado está limitado por las siguientes advertencias:

- **Un modelo es una abstracción de la realidad.** El modelador, dependiendo de sus necesidades, muestra partes que son importantes para una situación particular y deja fuera otras que considera menos importantes. Por lo tanto, el modelo no es una representación completa de la realidad. Esto lleva a la posibilidad que el modelo esté sujeto a diferentes interpretaciones.
- **A menos que el modelo sea dinámico, no provee el correcto sentido de la evolución en función del tiempo.** Dado que la realidad es cambiante, es imperativo que el modelo cambie en consecuencia. De otra forma, no podrá transmitir el significado real al usuario.
- **Un modelo puede crearse para una situación específica o para manejar un problema particular.** No hace falta decir que una vez que la situación ha cambiado, el modelo no será relevante.
- **Un modelo es una representación única de posibles elementos múltiples de la realidad.** Por ejemplo: una clase en software es una representación de múltiples objetos, sin embargo, no provee información respecto a volumen o performance.
- **El usuario del modelo puede no estar informado de las notaciones y el lenguaje utilizado para expresar el modelo.** El no comprender las notaciones o el proceso puede volver inútil al modelo.
- **Modelar informalmente, sin el debido cuidado y consideración por la naturaleza de los modelos en sí mismos, usualmente provoca confusión en los proyectos y puede reducir productividad.** Por lo tanto, la formalidad en el modelado es necesaria para su éxito.
- **Los modelos deben cambiar con el cambio de la realidad.** Conforme las aplicaciones y sistemas cambian los modelos deben hacer lo mismo para continuar siendo relevantes. Los modelos desactualizados pueden ser engañosos.
- **Los procesos juegan un rol significativo en dirigir las actividades de modelado.** El modelado sin considerar procesos es una práctica potencial riesgosa y debería evitarse.

Lenguaje de Modelado Unificado (UML)

Conceptos básicos sobre UML

UML son las siglas para Unified Modeling Language, que en castellano quiere decir: Lenguaje de Modelado Unificado.

Es un lenguaje de modelado, de propósito general, usado para la visualización, especificación, construcción y documentación de sistemas Orientados a Objetos.

Para comprender qué es el UML, basta con analizar cada una de las palabras que lo componen, por separado.

- **Lenguaje:** el UML es, precisamente, un lenguaje, lo que implica que éste cuenta con una sintaxis y una semántica. Por lo tanto, al modelar un concepto en UML, existen reglas sobre cómo deben agruparse los elementos del lenguaje y el significado de esta agrupación.
- **Modelado:** el UML es visual. Mediante su sintaxis se modelan distintos aspectos del mundo real, que permiten una mejor interpretación y entendimiento de éste.
- **Unificado:** unifica varias técnicas de modelado en una única.

Breve Reseña Histórica

Las raíces del UML provienen de tres métodos distintos: el método de Grady Booch, la Técnica de Modelado de Objetos de James Rumbaugh y “Objetory”, de Ivar Jacobson. Estas tres personalidades son conocidas como “los tres amigos”. En 1994 Booch, Rumbaugh y Jacobson dieron forma a la primera versión del UML y en 1997 fue aceptado por la OMG, fecha en la que fue lanzada la versión v1.1 del UML. Desde entonces, UML atravesó varias revisiones y refinamientos hasta llegar a la versión actual: UML 2.0.

¿Qué es la OMG?

La OMG (Object Management Group) es una asociación sin fines de lucro formada por grandes corporaciones, muchas de ellas de la industria del software, como, por ejemplo: IBM, Apple Computer, Oracle y Hewlett-Packard. Esta asociación se encarga de la definición y mantenimiento de estándares para aplicaciones de la industria de la computación. Podemos encontrar más información sobre la OMG en su sitio oficial: <http://www.omg.org/> (external link)

UML ha madurado considerablemente desde UML 1.1. Varias revisiones menores (UML 1.3, 1.4 y 1.5) han corregido defectos y errores de la primera versión de UML. A estas le ha seguido la revisión mayor UML 2.0 que fue adoptada por el OMG en 2005.

A partir de la versión 2.0 de UML, se modificó el lenguaje, de manera tal que permitiera capturar mucho más comportamiento (Behavior). De esta forma, se permitió la creación de herramientas que soporten la automatización y generación de código ejecutable, a partir de modelos UML.

Dado que es un modelo en constante revisión y evolución, la última versión formalmente liberada es el UML 2.5.

Se puede aplicar en el desarrollo de software gran variedad de formas para dar soporte a una metodología de desarrollo de software (tal como el Proceso Unificado Racional o RUP), pero no especifica en sí mismo qué metodología o proceso usar.

UML cuenta con varios tipos de diagramas, los cuales muestran diferentes aspectos de los sistemas que se modelan.

Presentación de los diagramas de UML

En el siguiente cuadro se muestra una breve descripción de cada uno de los diagramas que integran la versión 2.0 de UML, así como su clasificación.

Clasificación	Diagrama	Descripción
Diagrama de Estructura	Diagrama de Clases	Muestra una colección de elementos de modelado, tales como clases, tipos y sus contenidos y relaciones.
<i>Estos diagramas modelan estructura del sistema del sistema, por consiguiente, son diagramas estáticos</i>	Diagrama de Componentes	Representa los componentes que componen una aplicación, sistema o empresa. Los componentes, sus relaciones, interacciones y sus interfaces públicas. Un componente es una pieza de código de cualquier tipo.
	Diagrama de Estructura Compuesta	Representa la estructura interna de un clasificador (tal como una clase, un componente o un caso de uso), incluyendo los puntos de interacción de clasificador con otras partes del sistema.
	Diagrama de Despliegue	Un diagrama de despliegue muestra cómo y dónde se desplegará el sistema. Las máquinas físicas y los procesadores se representan como nodos. Como los componentes se ubican en los nodos para modelar el despliegue del sistema, la ubicación es guiada por el uso de las especificaciones de despliegue.
	Diagrama de Objetos	Un diagrama que presenta los objetos y sus relaciones en momento del tiempo. Un diagrama de objetos se puede considerar como un caso especial, un ejemplo de instanciación de un diagrama de clases.
	Diagrama de Paquetes	Un diagrama que presenta cómo se organizan los elementos de modelado en paquetes y las dependencias entre ellos.

Diagramas de Comportamiento	Diagrama de Casos de Uso	Un diagrama que muestra las relaciones entre los actores (que representan el ambiente de interés para el sistema), y los casos de uso.
<i>Diagramas que modelan comportamiento dinámico del sistema a nivel de clases.</i> <i>La excepción es el diagrama de casos de uso que modela comportamiento estático del sistema.</i>	Diagrama de Máquinas de Estado	Un diagrama que ilustra cómo los objetos de una clase, se puede mover entre estados que clasifican su comportamiento, de acuerdo con eventos disparadores de transiciones.
	Diagrama de Actividades	Representa los procesos de negocios de alto nivel, incluidos el flujo de datos. También puede utilizarse para modelar lógica compleja y/o paralela dentro de un sistema.
Diagramas de Interacción.	Diagrama de Comunicación	Es un diagrama que enfoca la interacción entre líneas de vida, donde es central la arquitectura de la estructura interna y cómo ella se corresponde con el pasaje de mensajes. La secuencia de los mensajes se da a través de un esquema de numerado de la secuencia.
<i>Tipo especial de diagramas de comportamiento que modelan interacción entre objetos, por consecuencia modelan el comportamiento dinámico del sistema.</i>	Diagrama de Secuencias	Un diagrama que representa una interacción, poniendo el foco en la secuencia de los mensajes que se intercambian, junto con sus correspondientes ocurrencias de eventos en las Líneas de Vida.
	Diagrama de Tiempos	El propósito primario del diagrama de tiempos es mostrar los cambios en el estado o la condición de una línea de vida (representando una Instancia de un Clasificador o un Rol de un clasificador) a lo largo del tiempo lineal. El uso más común es mostrar el cambio de estado de un objeto a lo largo del tiempo, en respuesta a los eventos o estímulos aceptados. Los eventos que se reciben se anotan, a medida que muestran cuándo se desea mostrar el evento que causa el cambio en la condición o en el estado.
	Diagrama de Descripción de Interacción	Los Diagramas de Revisión de la Interacción enfocan la revisión del flujo de control, donde los nodos son Interacciones u Ocurrencias de Interacciones. Las Líneas de Vida los Mensajes no aparecen en este nivel de revisión

Tabla 2 – Diagramas de UML 2.0

Es importante recordar que la elección de los modelos a construir varía en cada situación en particular y cada uno de los 13 diagramas de UML 2.0 sirve para modelar aspectos diferentes del software. En este material se trabajará en forma simplificada sobre:

- ↳ Modelado de Clases con Diagrama de Clases
- ↳ Modelado de Requerimientos funcionales con Diagramas de Casos de Uso
- ↳ Modelado de Objetos con Diagrama de Secuencia
- ↳ Modelado de estados de una clase con Diagrama de Máquina de Estados

Diagrama de Caso de uso

Un diagrama de casos de uso muestra actores, caso de uso, paquetes de casos de uso y sus relaciones.

Los diagramas de casos de uso pueden organizarse en paquetes de casos de uso, mostrando sólo lo que es relevante con un paquete particular.

Componentes del Diagrama de Caso de uso

Caso de uso	Nombre del Caso de Uso	Un caso de uso registra un contrato entre los involucrados del sistema acerca de su comportamiento, en varias circunstancias, organizándolo por los objetivos de los actores seleccionados.
	Según UML, el nombre del caso de uso debe colocarse debajo de la elipse que lo representa, sin embargo, no todas las herramientas que modelan UML, respetan esta convención, colocando el nombre dentro de la elipse.	El nombre del caso de uso deberá tener un nombre que indique lo que se alcanza por su interacción con el actor. El nombre es una frase verbal, que inicia con un verbo en infinitivo, puede tener varias palabras, debe ser entendible. No puede haber dos casos de uso con el mismo nombre. Ejemplos de nombres podrían ser:
Actor	Nombre del Actor	<ul style="list-style-type: none"> • Registrar venta • Generar comprobante de inscripción.
	Dado que un actor representa un rol, el nombre debe ser representativo de ese rol. Para comprender completamente el propósito del sistema, se debe saber para quién es el sistema.	<p>Un actor es cualquier cosa que intercambia datos con el sistema, puede ser un usuario, hardware externo u otro sistema (software).</p> <p>La diferencia entre un actor y una instancia, es que el actor representa una clase más que un usuario real. Varios usuarios pueden jugar el mismo rol, lo que significa que pueden ser uno y el mismo actor. En tal caso, cada usuario constituye una instancia de actor.</p> <p>El mismo usuario puede también actuar como varios actores, es decir, la misma persona puede tomar diferentes roles</p>
Asociación		<p>La asociación es una relación que vincula a un actor con un caso de uso.</p> <p>La asociación es una relación que puede plantearse según las siguientes opciones:</p> <ul style="list-style-type: none"> • Actor-Caso de Uso 1: esta navegabilidad se utiliza para indicar que es el actor el que inicia la relación de interacción con el caso de uso. • Actor-Caso de Uso 2: esta navegabilidad se utiliza para indicar que es el Caso de Uso el que inicia la relación de interacción con el Actor. • Actor-Caso de uso 3. Se utiliza cuando no está definido aún quien de los dos iniciar la relación de interacción.

Tabla 3 – Elementos básicos de modelado para un diagramas de Casos de Uso en UML 2.0

La funcionalidad del sistema es definida por diferentes casos de uso, cada uno de los cuales representa un flujo de eventos específico. La descripción de un caso de uso define qué ocurre en el sistema cuando el caso de uso es ejecutado.

La colección de casos de uso constituye todas las maneras posibles de usar el sistema. Se puede obtener una idea de la tarea del caso de uso simplemente observando su nombre.

¿Cómo Encontrar Casos de uso?

El siguiente conjunto de preguntas es útil para identificar casos de uso:

- Para cada actor identificado: ¿cuáles son las tareas en las cuales el sistema debería estar involucrado?
- ¿Necesita el actor ser informado acerca de ciertas ocurrencias en el sistema?
- ¿Necesita el actor informar acerca de cambios externos, repentinos?
- ¿El sistema brinda al negocio el comportamiento correcto?
- ¿Pueden ejecutarse todos los aspectos por los casos de uso que se han identificado?
- ¿Qué casos de uso soportarán y mantendrán el sistema?
- ¿Qué información debe ser modificada o creada en el sistema?

También son casos de uso, aunque no representan las funciones principales del sistema, los siguientes:

- Inicio y finalización del sistema
- Mantenimiento del Sistema. Por ejemplo: agregar nuevos usuarios, definir perfiles de usuarios.
- Mantenimiento de los datos almacenados en el sistema, ejemplo: el sistema debe trabajar en paralelo con un sistema anterior, heredado y los datos necesitan sincronizarse entre los dos.
- Funcionalidad necesaria para modificar el comportamiento del sistema.

¿Todos los Casos de uso se describen en detalle?

El UML define lineamientos respecto a cómo construir diagramas de casos de uso y una recomendación de complementar los diagramas con descripciones para los casos de uso. No obstante, no define estándares respecto de cómo deben ser esas descripciones.

Frecuentemente habrá en los modelos de casos de uso algunos casos de uso que son tan simples que no necesitan una descripción detallada del flujo de eventos, una descripción general es suficiente. El criterio para tomar esa decisión es que no se vean desacuerdos entre quienes lo leerán o en lo que el caso de uso significa.

Descripción del Caso de Uso

El propósito de la descripción o narrativa de un caso de uso es contar la historia de cómo el sistema y sus actores trabajan en conjunto para alcanzar una meta específica. Las narrativas de los casos de uso:

- Describen una secuencia de acciones, incluyendo variantes, que un sistema y sus actores pueden ejecutar para alcanzar una meta.
- Se representan como un conjunto de flujos que describen cómo un actor emplea un sistema para alcanzar una meta y lo que el sistema hace para ayudar al actor a alcanzar esa meta.
- Capturan la información de los requisitos necesarios para apoyar las otras actividades del desarrollo.

Las narrativas de los casos de uso se pueden capturar de muchas maneras, incluyendo una Wiki, tarjetas indexadas, documentos de MS Word o dentro de una de las muchas herramientas de administración, de manejo de requisitos o de modelado de requisitos comercialmente disponibles.

Las narrativas de los casos de uso se pueden desarrollar a distintos niveles de detalle, desde un simple resumen, identificando el flujo básico y las variantes más importantes, hasta una especificación completa, altamente detallada, que defina las acciones, entradas y salidas involucradas en la ejecución del caso de uso. Las narrativas de los casos de uso se pueden preparar en los siguientes niveles de detalle:

Brevemente Descrita	El nivel más liviano de detalle que sólo captura la meta del caso de uso y el actor que lo inicia. Este nivel de detalle es apropiado para aquellos casos de uso que usted decide no implementar. Se necesitará más detalle si el caso de uso se va a dividir para su implementación.
Con Resumen Básico	El caso de uso se debe resumir para entender su tamaño y complejidad. Este nivel de detalle también posibilita un manejo efectivo del alcance, puesto que el resumen permite que las distintas partes del caso de uso se prioricen unas con otras y, si es necesario, se destinen a distintas versiones. Este es el nivel más liviano de detalle que posibilita que el caso de uso se divida y desarrolle. Es ajustable para aquellos equipos que trabajan en estrecha colaboración con sus usuarios y que son capaces de completar cualquier detalle faltante vía conversaciones y la terminación de los casos de prueba acompañantes.
Con Resumen Esencial	Algunas veces es necesario aclarar las responsabilidades del sistema y sus actores mientras se trabaja en el caso de uso. Un resumen básico captura sus responsabilidades, pero no define claramente qué partes del caso de uso son responsabilidad del caso de uso y qué partes son responsabilidad de los actores. En este nivel de detalle, la narrativa se convierte en una descripción del diálogo entre el sistema y sus actores. Es particularmente útil cuando se establece la arquitectura de un nuevo sistema o se intenta establecer una nueva experiencia de usuario.
Completamente Descrita	Las narrativas de los casos de uso se pueden usar para proporcionar una especificación bastante detallada de los requisitos, haciéndolas evolucionar hasta su nivel de detalle más completo: completamente descritas. El detalle extra puede ser necesario para cubrir la falta de experiencia dentro del equipo, la falta de acceso a los interesados o para comunicar efectivamente requisitos complejos. Este nivel de detalle es particularmente útil para aquellos casos de uso donde un malentendido del contenido podría tener consecuencias severas de seguridad, finanzas o asuntos legales. También puede ser útil cuando hacemos desarrollo de software externo o por subcontratación.

Tabla 4 – Tipos de Narrativas para la descripción de los Casos de Uso

La narrativa del caso de uso es un producto de trabajo bastante flexible que se puede expandir para capturar la cantidad de detalle necesario para cualquier circunstancia.

Si usted es parte de un equipo pequeño que trabaja de manera colaborativa con el cliente o de un proyecto exploratorio, entonces los resúmenes básicos proporcionarán una forma liviana de descubrir los requisitos. Si usted está trabajando en un entorno más rígido donde hay poco acceso a los expertos, entonces las narrativas con resúmenes esenciales o las completamente descritas se pueden usar para cubrir los vacíos en el conocimiento del equipo. No todos los casos de uso se deben especificar al mismo nivel de detalle; no es extraño que los casos de uso más importantes y riesgosos tengan más detalle que

los otros. Lo mismo va para las secciones de la narrativa del caso de uso: las partes más importantes, complejas o riesgosas de un caso de uso, frecuentemente, se describen con más detalle que las otras.

Un caso de uso tiene muchas instancias posibles

Un caso de uso puede seguir un número ilimitado, pero numerable de caminos. Estos caminos representan escenarios que están dentro del caso de uso, porque responden al mismo objetivo, aunque con algunas variaciones.

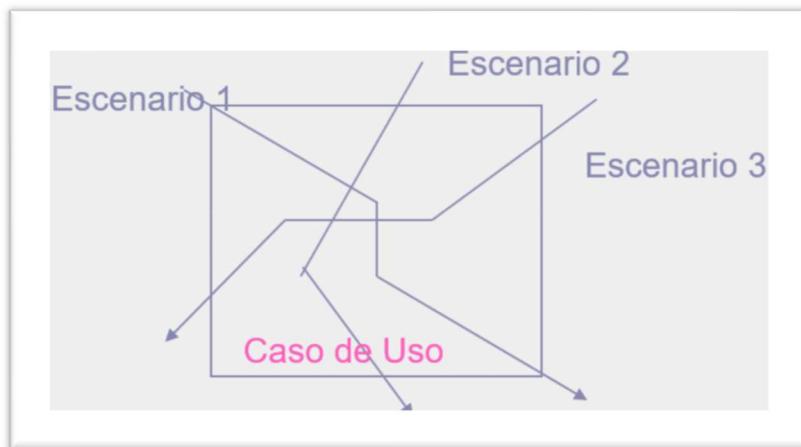


Fig. 18- Un caso de uso y los escenarios que lo conforman

El escenario elegido depende de los eventos. Los tipos de eventos, incluyen:

- **Entrada desde un actor:** un actor puede decidir, de varias opciones, que hacer después. (Ejemplo: al actor puede decidir ingresar otra botella en una maquina recicladora de botellas o pedir el recibo para finalizar la transacción).
- **Un control de valores o tipos de un objeto o atributo interno:** el flujo de eventos puede diferir si un valor es más o menos grande que un cierto valor. (Ejemplo: en un caso de uso de extracción en un cajero automático, el flujo puede diferir si el actor pide más dinero del que puede extraer de su cuenta).

Los escenarios pueden agruparse, según si cumplen con el objetivo o no en **escenarios o flujos de éxito** y **escenarios o flujos de fracaso**, respectivamente. Dentro de los escenarios de éxito, suele destacarse el que se conoce como “Escenario Feliz”, que es el que describe el flujo de eventos que conforma la manera más sencilla de cumplir con el objetivo.

¿Cómo sabe usted cuándo dejar de escribir un escenario o un caso de uso?

Hay dos cláusulas que tienen que ser hechas explícitamente para conseguir los límites apropiados en un caso de uso y un escenario. Para ambos casos las cláusulas son las mismas:

Cláusula 1. Todas las interacciones se relacionan al mismo objetivo.

Cláusula 2. Las interacciones empiezan activando al evento y terminan cuando el objetivo se alcanza o se abandona, y el sistema completa sus responsabilidades con respecto a la interacción.

La Cláusula 2 trae a colación la noción de tener una responsabilidad con respecto a la interacción. Muchos sistemas tienen que anotar cada interacción con un cliente (Ej. los sistemas de banco). Si la cláusula no incluyera este requisito, el guion acabaría sin esa responsabilidad.

Descripción de los casos de uso

El flujo de eventos de un caso de uso contiene la información más importante derivada del trabajo de modelado de casos de uso. El flujo debería presentar lo que el sistema hace, no cómo está diseñado el sistema para ejecutar el comportamiento requerido.

LINEAMIENTOS PARA EL CONTENIDO DEL FLUJO DE EVENTOS:

- Describir como inicia y termina el caso de uso
- Describir que datos se intercambian entre el actor y el caso de uso.
- No describir detalles de la interfaz del usuario, las descripciones deben ser con tecnología neutra, es decir sin hacer referencia a cómo se va a resolver el comportamiento.
- Describir el flujo de eventos, reforzando interacción entre actor y caso de uso.
- Describir sólo los eventos que pertenecen a ese caso de uso, y no lo que pasa en otros casos de uso o fuera del sistema.
- Evitar terminología vaga tal como “por ejemplo” “etc.” “información”.
- Detalle en el flujo de eventos todos los “que” que deberían responderse, recuerde que los diseñadores de pruebas usarán ese texto para identificar casos de prueba.
- Respete la terminología del dominio del problema, si ha utilizado ciertos términos en otros casos de uso, asegúrese que se usen los mismos términos en el caso de uso que está describiendo y que su significado sea el mismo. Para manejar terminología común, utilice glosarios.

¿Por qué definir actores?

Los roles que los actores juegan proveen una perspectiva importante de porqué es necesario el caso de uso y su salida. La razón por la cual un actor particular inicia un caso de uso es casi siempre la razón principal de porqué se necesita el caso de uso en primer lugar. Las diferencias sutiles en el rol que un actor juega pueden producir variaciones en las interacciones del caso de uso. Por ejemplo, determinar que el sistema de procesamiento de préstamos debe responder tanto a clientes individuales como corporativos identifica la necesidad de modelar casos de uso únicos para las necesidades específicas de ambos.

Focalizando en los actores, se puede concentrar en cómo será usado en sistema en lugar de pensar cómo será construido o implementado. También ayuda a refinar y definir los límites del sistema. Finalmente, los actores pueden ser un factor para decidir cómo se llevará a cabo el desarrollo en función de entregas de valor para el negocio.

La definición de actores ayuda a identificar potenciales usuarios que necesitan estar involucrados en el esfuerzo de modelado de caso de uso.

¿Cómo encontrar actores?

Revisar las siguientes fuentes de información permitirá encontrar potenciales actores del sistema:

- Diagramas de Contexto u otros modelos existentes que describen el límite del sistema con su entorno. Buscar entidades externas que tienen interacciones con el sistema.
- El análisis de involucrados que determina que grupo de personas interactuarán con el sistema.
- Especificaciones escritas y otra documentación del proyecto, tal como memos acerca de reuniones con usuarios, ayuda a identificar usuarios que podrán ser actores potenciales.

- Minutas de reuniones de requerimientos. Los participantes de estas sesiones pueden ser importantes, debido a que los roles que ellos representan en la organización pueden interactuar con el sistema.
- Guías de Capacitación y Manuales de Usuarios para procesos y sistemas actuales. Estos manuales y guías están dirigidos a menudo a actores potenciales.

El siguiente grupo de preguntas es útil para tener en mente cuando se identifica actores:

- ¿Quién o qué inicia eventos con el sistema?
- ¿Quién proveerá, usará o quitará información?
- ¿Quién usará esta funcionalidad?
- ¿Quién está interesado en cierto requerimiento?
- ¿En qué parte de la organización será usado el sistema?
- ¿Quién dará soporte y mantendrá el sistema?
- ¿Cuáles son los recursos externos del sistema?
- ¿Qué otros sistemas necesitarán interactuar con este sistema?

Existen algunos aspectos diferentes de los alrededores del sistema que deben representarse como actores separados:

- Usuarios que ejecutan las funciones principales del sistema
- Usuarios que ejecutan las funciones secundarias del sistema, tal como la administración del sistema.
- Hardware externo que el sistema usa.
- Otros sistemas que interactúan con el sistema.

Los actores ayudan a definir los límites del sistema

Encontrar actores también significa que se establecen los límites del sistema, lo que ayuda a entender el propósito y alcance del sistema. Solo aquellos que se comunican directamente con el sistema necesitan ser considerados actores.

Si se están incluyendo más roles que los que están alrededor del sistema, está intentando modelar el negocio en el cual el sistema será utilizado, no el sistema en sí mismo.

Ejemplo: Si está construyendo un sistema para reservas aéreas, para ser usado por un agente de viajes, el actor será el agente de viajes. El pasajero no interactúa directamente con el sistema y por lo tanto no es un actor. Si está construyendo un sistema para reservas aéreas que permitirá a los usuarios conectarse vía Internet, el pasajero interactuará directamente con el sistema y por lo tanto será un actor.

Breve descripción de los actores

Una breve descripción del actor debería incluir información acerca de:

- A qué o quién representa el actor
- Porqué es necesario el actor
- Qué interés tiene el actor en el sistema.
- Características generales de los actores, tales como nivel de experticia (educación), implicaciones sociales (lenguaje), y edad. Estas características pueden influenciar detalles de la interface de usuario, tal como fuente y lenguaje.

Estas características son utilizadas principalmente cuando se identifican las clases de interfaz y el prototipo, para asegurar la mejor usabilidad entre la comunidad de usuarios y el diseño de interfaz gráfica.

Generalización de Actores

Una generalización de actor de un tipo de actor (descendiente) a otro tipo de actor (ancestro) indica que el descendiente hereda el rol que el ancestro puede jugar en un caso de uso.

Varios actores pueden jugar el mismo rol en un caso de uso particular., por ejemplo, los actores cobrador y contador heredan las propiedades del Supervisor de balance, dado que ambos controlan el balance de una cuenta.

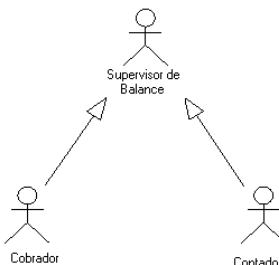


Fig. 19- Generalización entre actores

Un usuario puede jugar varios roles en relación con el sistema, lo que significa que un usuario puede corresponder a varios actores. Para hacer más claro el modelo se puede representar al usuario por un actor que hereda a algunos actores. Cada actor heredado representa uno de los roles del usuario relativo al sistema.

Uso del Diagrama de Caso de uso

No hay reglas estrictas acerca de cómo dibujar un diagrama de caso de uso, pero los siguientes diagramas pueden ser de interés:

- Actores que pertenecen al mismo paquete de caso de uso.
- Un actor y todos los casos de uso con los que interactúa.
- Casos de uso que manejan la misma información.
- Casos de uso usados por el mismo grupo de actores.
- Casos de uso que se ejecutan a menudo con la misma secuencia.
- Casos de uso que pertenecen al mismo paquete de caso de uso.
- Los casos de uso más importantes. Un diagrama de este tipo puede servir como un resumen del modelo.
- Los casos de uso desarrollados junto, en el mismo incremento.
- Un caso de uso específico y sus relaciones con actores y otros casos de uso.

Es recomendable incluir cada actor, caso de uso y relación en al menos uno de los diagramas. Si hace al modelo de caso de uso más claro, puede formar parte de varios diagramas y/o mostrarlos varias veces en el mismo diagrama.

Casos de uso Concretos y Abstractos

Un caso de uso **concreto** es iniciado por un actor y constituye un flujo de eventos completo. "Completo" significa que una instancia de un caso de uso ejecuta una operación entera llamada por un actor.

Un caso de uso **abstracto** nunca se inicia por sí mismo. Los casos de uso abstractos son incluidos en, extienden o generalizan otros casos de uso. Cuando un caso de uso concreto es iniciado, se crea una instancia de caso de uso. Por lo tanto, no se crean instancias separadas desde los casos de uso abstractos.

La distinción entre los dos es importante, debido a que en los casos de uso concretos los actores verán e iniciarán el sistema.

Estructurando el Modelo de Caso de uso

Las razones principales para estructurar el modelo de casos de uso:

- Para hacer a los casos de uso más fáciles de entender.
- Para particionar comportamiento común descripto en muchos casos de uso.
- Para hacer el modelo de caso de uso más fácil de mantener.

No hay forma de estructurar los casos de uso hasta que no se sabe un poco más de su comportamiento, además de una breve descripción. Al menos se deberá tener una descripción un poco más amplia para asegurar que las decisiones están basadas en una comprensión suficientemente adecuada del comportamiento.

Para estructurar caso de uso se tienen tres clases de relaciones. Las describimos brevemente a continuación:

Tipo de Relación	Definición y características
<p style="text-align: center;">Extensión</p>	<ul style="list-style-type: none"> • Especifica cómo un caso de uso puede insertarse y así extender la funcionalidad de otro. • El caso de uso donde se insertará la extensión debe ser un caso de uso completo en sí mismo. • Se usan para modelar partes optativas, alternativas de un caso de uso, que a su vez tienen un objetivo propio. • Se dibuja con una flecha de línea cortada, cuya dirección va desde el caso de uso de extensión (adicional) al caso de uso base.

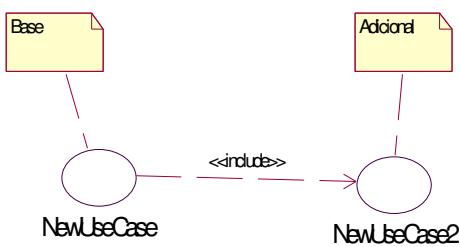
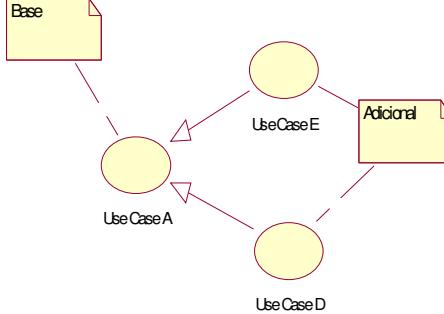
Inclusión 	<ul style="list-style-type: none"> • Especifica y agrupa comportamiento similar de varios casos de usos, en un caso de uso abstracto, que otros podrán usar. • Se usan cuando su intervención es necesaria para completar un curso completo de eventos. • Se dibuja con una flecha desde el caso de uso concreto o base al caso de uso abstracto (adicional).
Generalización 	<ul style="list-style-type: none"> • Se usa cuando los hijos tienen el mismo objetivo y comportamiento que el padre, aunque con variaciones. • Un caso de uso más específico puede especializar a un caso de uso más general. • Una relación de generalización entre casos de usos implica que el caso de uso hijo contiene todos los atributos y secuencias de comportamiento y puntos de extensión definidos para el padre. • Se dibuja con una flecha de línea entera y punta cerrada, desde el caso de uso hijo al padre. • Los casos de usos hijos pueden redefinir el comportamiento heredado del padre.

Tabla 5 – Tipos de relaciones utilizadas para estructurar un diagrama de Casos de Uso

¿Están los casos de uso relacionados siempre a Actores?

La ejecución de cada caso de uso incluye la comunicación con uno o más actores. Una instancia de caso de uso siempre es iniciada con un actor que le pide al sistema que haga algo. Esto implica que cada caso de uso debería tener asociaciones de comunicación con actores. La razón de esta regla es forzar al sistema a proveer solo la funcionalidad que los usuarios necesitan y nada más. Tener casos de uso que nadie requiere es un indicador de que hay algo erróneo en el modelo de caso de uso o en los requerimientos.

Sin embargo, existen algunas excepciones a esta regla:

- Si un caso de uso es abstracto (no instanciable separadamente), su comportamiento puede no incluir interacción con un actor.
- Un caso de uso hijo en una relación de generalización, puede no tener actor asociado, si el caso de uso padre describe toda la comunicación con el actor.
- Un caso de uso base en una relación de inclusión puede no tener un actor asociado, si el caso de uso de inclusión describe toda la comunicación con el actor.
- Un caso de uso que es iniciado de acuerdo a una planificación, lo que significa que el reloj es el iniciador. El reloj del sistema es un evento interno y el caso de uso no es iniciado por un actor. Sin embargo, por claridad, se puede usar un actor ficticio “Tiempo” para mostrar cómo es iniciado el caso de uso en los diagramas de caso de uso.

Ejemplo de uso del Diagrama de Casos de Uso

El ejemplo presenta una vista parcial de un *Sistema de Ventas y Reservas de Entradas* para un *Complejo de Cines*. Este complejo de cines cuenta con varios cines distribuidos en la ciudad. Cada cine tiene varias salas. Las salas no tienen el mismo tamaño, por consiguiente, la capacidad en términos de cantidad de butacas disponibles, no es la misma para todas las salas.

Las funciones de películas que se proyectarán en cada una de las salas de cada cine del complejo, las define el Responsable de Programación en forma centralizada para cada cine.

Esta programación de funciones se usa para reservar lugares y para vender entradas, para una función habilitada para ello. La reserva de entradas puede hacerse vía telefónica o vía web.

La vista que se presenta a continuación muestra cada uno de los recursos de modelado de la técnica:

- Casos de uso concretos y abstractos (concretos son todos menos el caso de uso 1 y el 6 que son abstractos).
- Actores concretos y un actor abstracto (concretos son todos menos el Usuario que es abstracto).
- Ejemplos de uso de las relaciones entre casos de uso (inclusión (en inglés *include*), extensión (en inglés *extend*), y la generalización entre los casos de uso 7 y 8 que heredan del caso de uso 6).

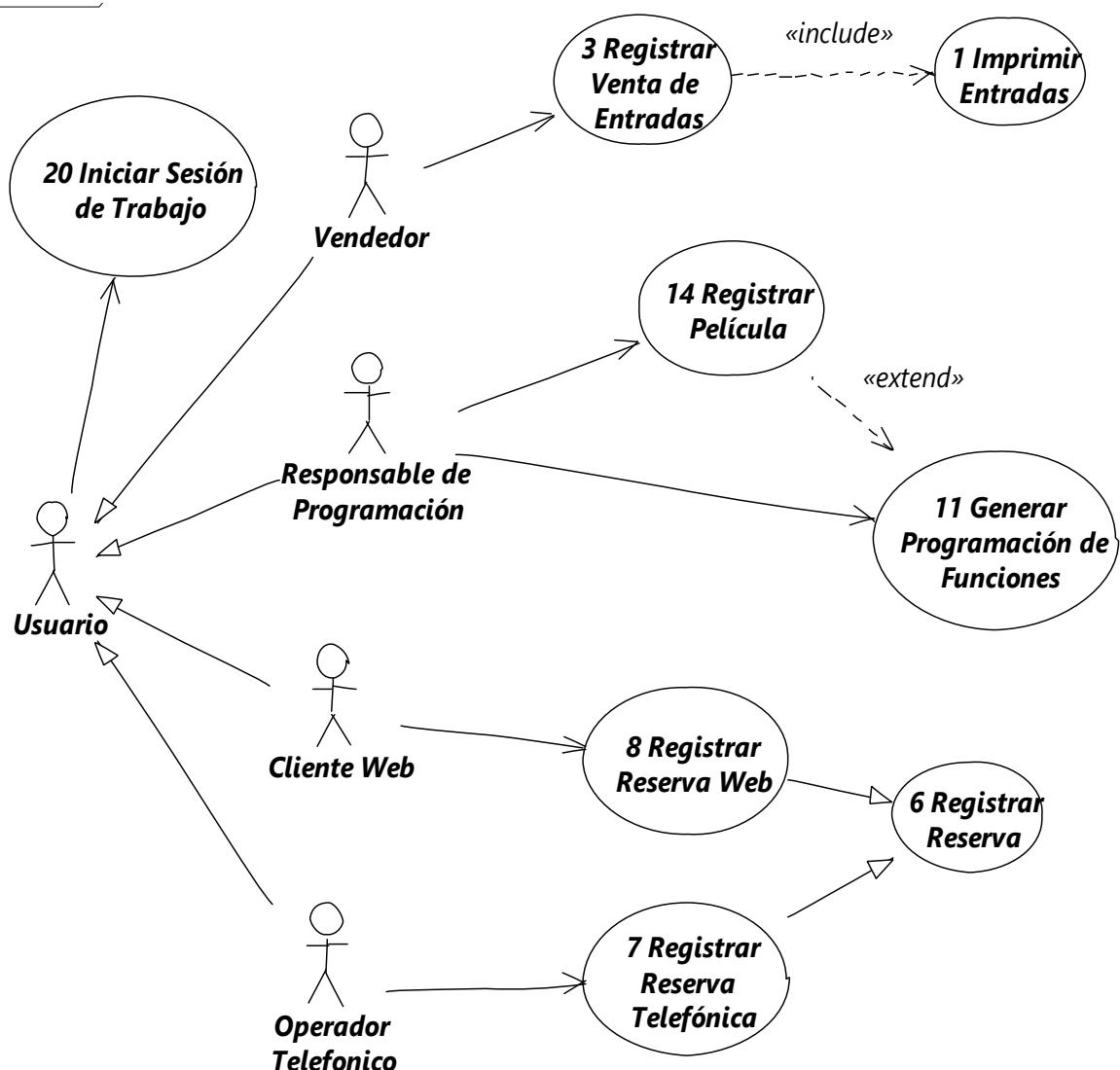


Fig. 20 – Ejemplo de una vista de un diagrama de casos de uso para un Complejo de Cines

Descripción de Actores

Nombre del Actor	Descripción
Cliente Web	Persona que tiene acceso a una computadora con conexión a Internet y puede entrar a la página Web del Complejo de Cines para realizar consultas de programación y reservas para funciones de los diferentes cines del complejo.
Responsable de Programación	Persona que administra toda la funcionalidad vinculada con la obtención de la programación de funciones y la registración de nuevas películas que integrarán las programaciones del complejo.
Vendedor	Persona responsable de la registración y anulación de ventas de entradas para las funciones habilitadas en los cines del complejo.
Operador Telefónico	Persona responsable de la creación consulta y modificación de reservas de lugares para funciones de los cines del complejo
Operador de Reserva	Responsable de la registración y /o modificación de reservas de lugares en una o más funciones de cine del complejo.
Usuario	Persona que está definida como usuario de la aplicación y va a registrarse en la misma para realizar alguna actividad.

Tabla 6 – Ejemplo de descripción de los actores que participan en la vista del diagrama de casos de uso presentado en la Figura 14.

Ejemplo de Descripción de un Caso de Uso con la narrativa de resumen esencial

Nombre del Caso de uso: Registrar Película	Nro. de Orden: 14
Actor Principal: Responsable de Programación (RP)	Actor Secundario: no aplica
Objetivo: registrar cada una de las películas que integrarán las distintas funciones que se programarán para los cines del complejo.	
Flujo Básico	
1. RP: selecciona la opción de ingresar nueva película . 2. Sistema: solicita se ingrese nombre de película. 3. RP: ingresa el nombre de la película que desea registrar. 4. Sistema valida que no exista una película registrada con el mismo nombre y no hay. 5. Sistema: busca y muestra los Géneros, Calificaciones y Países de Origen de la película y solicita se seleccionen 6. RP: selecciona los datos correspondientes a la película. 7. Sistema: solicita se ingresen los demás datos para la película. 8. RP ingresa: duración, título original, año de estreno. 9. Sistema: asigna el estado de la película como “NO DISPONIBLE” y el actor no desea modificarlo 10. Sistema: solicita confirmación para registrar película ingresada. 11. RP confirma. 12. Sistema: valida que se hayan especificado los datos mínimos (nombre, duración, año de estreno, calificación, género, país de origen, elenco), requeridos para realizar la registración de la película y han sido especificados. 13. Sistema registra la película.	
Flujos Alternativos	
A1: Existe una película registrada con el mismo nombre. A2: Elenco no pudo registrarse. A3: El RP desea ingresar premios de películas A4: El RP desea ingresar comentarios de películas A5: El sistema asigna el estado de la película como “NO DISPONIBLE” y el actor <u>desea</u> modificarlo. A6: No se han ingresado los datos mínimos requeridos para efectuar la registración de la película. A7: El RP no confirma registración de la película.	
Observaciones:	
1. El RP puede cancelar la ejecución del caso de uso en cualquier momento.	

Tabla 7 – Ejemplo de narrativa de los actores que participan en la vista del diagrama de casos de uso presentado en la Figura 14.

Diagrama de Clases

Los diagramas de clases muestran una vista estática de la estructura del sistema, o de una parte de éste, describiendo qué atributos y comportamiento debe desarrollar el sistema, detallando los métodos necesarios para llevar a cabo las operaciones del sistema. El diagrama de clases muestra un conjunto de interfaces, colaboraciones y sus relaciones.

Componentes del Diagrama de Clases

Clase

Descripción de un conjunto de objetos que comparten los mismos atributos, operaciones, relaciones y semántica. Se pueden utilizar para capturar el vocabulario del sistema que se está desarrollando. Pueden incluir abstracciones que formen parte del dominio del problema o abstracciones que formen parte del dominio de la solución. El comportamiento de una clase se describe mediante los mensajes que la clase es capaz de comprender junto con las operaciones que son apropiadas para cada mensaje.



Fig.21- Notación para una clase en UML

- **Visibilidad:** normalmente se desea ocultar los detalles de implementación y mostrar sólo las características necesarias para llevar a cabo las responsabilidades de la abstracción. Hay tres niveles disponibles:
 - *Pública*: Cualquier clasificador externo con visibilidad hacia el clasificador dado puede utilizar la característica. Se representa con el símbolo “+”.
 - *Protegida*: Cualquier descendiente del clasificador puede utilizar la característica. Se representa con el símbolo “#”.
 - *Privada*: Sólo el propio clasificador puede utilizar la característica. Se representa con el símbolo “-”
 - *Paquete*: Visible para clasificadores del mismo paquete. Se representa con el símbolo “~”.

Clases de Análisis

UML, ha incorporado como parte del lenguaje el uso de tres tipos de clases, denominadas *clases de análisis*, cuyo propósito es lograr una estructura más estable y mantenible del sistema que será robusta para el ciclo de vida entero del sistema. Se ha asignado una representación icónica para referirnos a cada uno de los tres tipos de clases de análisis, según se muestra a continuación

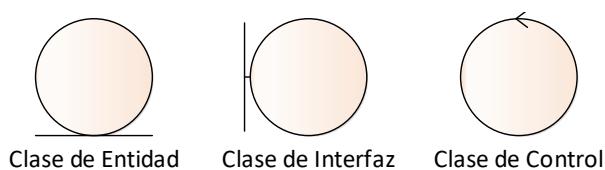


Fig. 22- Representación icónica de los tres tipos de clases de UML

Clase de Entidad

La clase de entidad modela información en el sistema que debería guardarse por mucho tiempo y, comúnmente, debería sobrevivir la ejecución del caso de uso del sistema en la que se creó. Todo el comportamiento naturalmente acoplado a esta información debería también ser ubicado en la clase de entidad. Las clases de entidad se utilizan para modelar abstracciones del dominio del problema. Las clases que se identifican en el modelo del dominio del problema son ejemplos de este tipo de clases. Un ejemplo de una clase de entidad es la clase que guarda comportamiento y datos relevantes de una **Película** para el complejo de cines.

Las clases de entidad pueden identificarse desde los casos de uso. La mayoría de las clases de entidad se encuentran pronto y son obvias. Estas clases de entidad “obvias” se identifican frecuentemente en el modelo de objetos del dominio del problema. Los otros pueden ser más difíciles de encontrar. Las entidades comúnmente corresponden a algún concepto en la vida real.

Clase de Interfaz

La clase de interface modela el comportamiento e información que es dependiente de la frontera del sistema con el ambiente. Así todo lo que concierne a cualquier vínculo entre el sistema y los actores, se ubica en un objeto de interfaz. Un ejemplo de un objeto de interfaz, sería la clase que representa la pantalla que vincula un actor con un caso de uso, para pedirle los datos de una película: **PantallaAdmPelicula**.

Toda la funcionalidad especificada en las descripciones de los casos de uso, que es directamente dependiente del ambiente del sistema se pone en los objetos de interface. Es mediante estos objetos que los actores se comunican con el sistema. La tarea de un objeto de interface es trasladar la entrada del actor al sistema en eventos del sistema, y traducir esos eventos del sistema en los que el actor está interesado en algo que será presentado al actor. Las clases de interface pueden, en otras palabras, describir comunicación bidireccional entre el sistema y sus usuarios. Cada actor concreto necesita su propia interfaz para comunicarse con el sistema.

Así, para identificar que parte del flujo de un caso de uso debería destinarse a una clase de interfaz, enfocamos las interacciones entre los actores y los casos de uso. Esto significa que deberíamos buscar unidades que muestran una o más de las características siguientes:

- Presentan información al actor o piden información de él.
- Su funcionalidad cambia conforme cambie el comportamiento del actor.
- Su curso es dependiente de un tipo particular de interfaz.

Clase de Control

La clase de control modela funcionalidad que implica operar sobre varios objetos diferentes de entidad, haciendo algunos cálculos y retornando el resultado al objeto de interface. Contiene comportamiento de la lógica de negocio definida en un caso de uso. Tiene responsabilidades de coordinación de la ejecución de un caso de uso y funciona como intermediario entre las clases de interfaz y las de control. Un ejemplo de una clase de control es el **GestorPelicula**, utilizada para coordinar la realización del caso de uso registrar película.

La clase de control actúa como vínculo, que une los otros tipos de clase. Son comúnmente las más efímeras de todos los tipos de objeto y comúnmente duran mientras dura la ejecución de un caso de uso. Es, sin embargo, difícil lograr un equilibrio razonable entre qué se pone en los objetos de entidad, en los objetos de control y en los objetos de interface. Daremos aquí algunas heurísticas con respecto a

cómo encontrarlos y especificarlos. Los objetos de control se encuentran directamente desde los use cases. En una primera vista asignaremos un objeto de control para cada use case.

Los tipos típicos de funcionalidad ubicados en las clases de control son comportamiento relacionado a la transacción, o secuencias específicas de control de uno o varios use cases, o la funcionalidad que separa los objetos de entidad de los objetos de interface. Los objetos de control conectan cursos de eventos y así llevan adelante la comunicación con otros objetos.

Estos dos últimos tipos de clases de análisis suelen llamarse de **fabricación pura**, y eso se debe a que son creadas para modelar el dominio de la solución y no tienen un vínculo directo con el dominio del problema.

¿Por qué modelar utilizando los tres tipos de clases de análisis?

La suposición básica es que todos los sistemas cambiarán. Por lo tanto, la estabilidad ocurrirá en el sentido que todos los cambios sean locales, esto es, que afecten (preferentemente) a una única clase del sistema. Déjenos primero considerar qué tipos de cambios son comunes al sistema. Los cambios más comunes al sistema son a su funcionalidad y a su interface. Los cambios a la interface deberían afectar únicamente a las clases de interface. Los cambios en la funcionalidad son más difíciles.

La funcionalidad puede ubicarse sobre todos los tipos de clases, por eso ¿cómo logramos la localización de estos cambios? Si es la funcionalidad que está asociada a la información retenida por el sistema, por ejemplo, cómo calcular la edad de una persona, tales cambios afectan a las clases de entidad que representa esa información. Los cambios de funcionalidad de una interface, por ejemplo, cómo recibir y recolectar información acerca de una persona, deberían afectar a la clase de interfaz correspondiente. Los cambios de funcionalidad entre clases, por ejemplo, cómo calcular impuestos con varios criterios diferentes, deberían ser locales a la clase de control. La funcionalidad atribuible a uno o varios casos de uso se asigna a una clase de control.

En síntesis, asignamos el comportamiento descripto en un caso de uso, según los siguientes principios:

- Aquellas funcionalidades del caso de uso, que son directamente dependientes del ambiente del sistema se ponen en las clases de interfaz.
- Aquellas funcionalidades que tratan con el almacenamiento y manipulación de información que no está naturalmente ubicada en ninguna clase de interfaz y que representa al dominio del problema, se ubica en las clases de entidad.
- Las funcionalidades específicas a uno o varios casos de uso y que cumplen roles de coordinación de un caso de uso, son modeladas con clases de control.

Interface

Una interface es un tipo especial de clase que agrupa una colección de operaciones que especifican un servicio de una clase o componente.

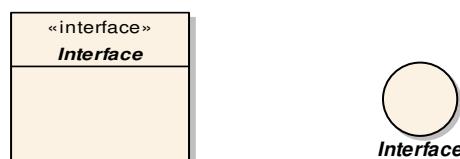


Fig. 23- Notaciones para una Interface en UML, representación etiquetada e icónica

Existen dos tipos de interfaces:

- *Interfaces Requeridas*: Muestran lo que el clasificar al que pertenecen puede requerir del ambiente a través de un puerto dado.
- *Interfaces Provistas*: Muestra la funcionalidad que expone un clasificador a su ambiente.

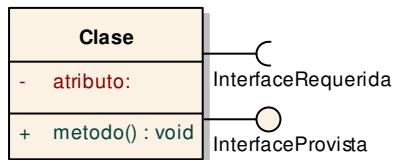


Fig. 24- Visualización de una clase con interfaces asociadas

Relaciones

Una relación es una conexión entre dos elementos. En el contexto del diagrama de clases, los elementos que se relacionan son clases e interfaces. Los tipos de relaciones son:

- ↳ **Asociación**: Relación estructural que especifica que los objetos de un elemento se conectan a los objetos de otro elemento. Este tipo de relación se suele implementar con una referencia en uno de los dos elementos que participan de la relación hacia el otro elemento. Estas relaciones pueden incluir multiplicidad, navegabilidad y el nombre del rol que se establece en la relación. Al implementarse esta relación el rol de la asociación suele ser del tipo de uno de los elementos en la relación.

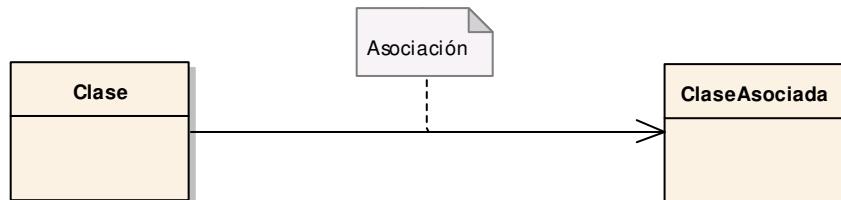


Fig. 25- Notación para representar la relación de asociación entre clases en UML

- ↳ **Agregación**: Es un tipo especial de asociación, que representa una relación completamente conceptual entre un “Todo” y sus “Partes”.



Fig. 26- Notación para representar la relación de agregación entre clases en UML

- ↳ **Composición**: Es una variación de la agregación que muestra una relación más fuerte entre el todo y sus partes. Se utiliza cuando existe una relación de contenedor-contenido entre dos

elementos. Usualmente una instancia de una parte suele pertenecer sólo a una instancia del todo. Esta relación en general implica que al borrarse el todo se borran todas sus partes.

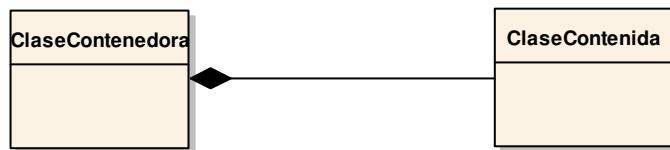


Fig. 27- Notación para representar la relación de composición entre clases en UML

- ☞ **Generalización:** Es una relación entre un elemento general (superclase o padre) y un tipo más específico de ese elemento (subclase o hijo). El hijo hereda los métodos y atributos del padre, luego puede añadir nueva estructura y comportamiento, o modificar el comportamiento del padre. Existen dos tipos de herencia:
 - Herencia Simple: Un hijo hereda de un único padre.

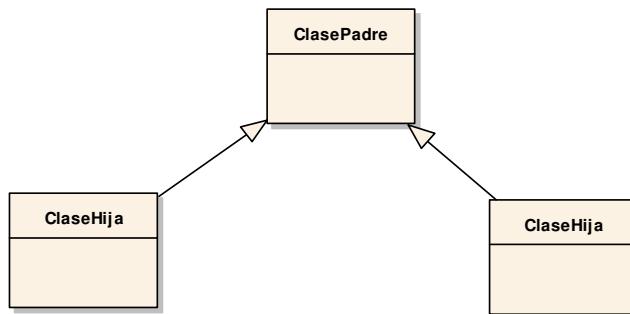


Fig. 28. Notación para representar la relación de generalización entre clases en UML, herencia simple

- **Herencia Múltiple:** Un hijo hereda de más de un padre. Este tipo de herencia ha sido prácticamente reemplazada por la realización de clases de interfaz, ya que las buenas prácticas recomiendan no utilizar herencia múltiple y muchos lenguajes de programación orientados a objetos no lo soportan.

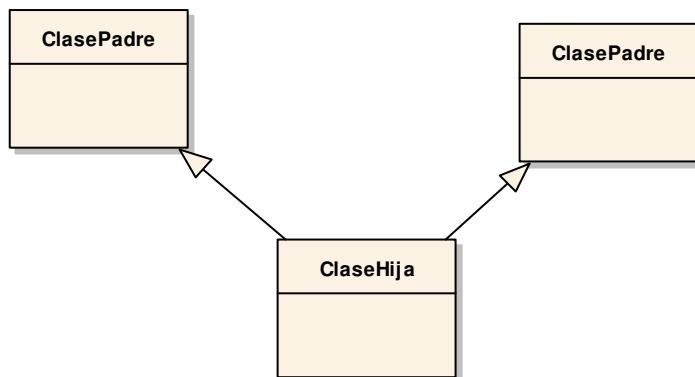


Fig. 29- Notación para representar la relación de generalización entre clases en UML, herencia múltiple

- Dependencia: Es una asociación de uso, la cual especifica que un cambio en la especificación de un elemento puede afectar a otro elemento que lo utiliza, pero no necesariamente a la inversa.

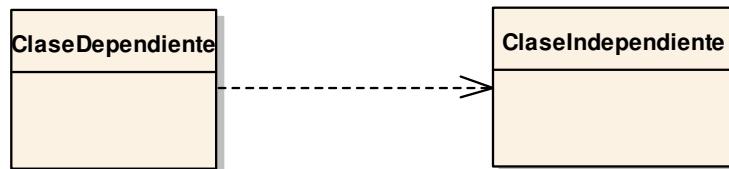


Fig. 30- Notación para representar la relación de dependencia entre clases en UML

- Realización: Esta relación implica que el elemento de donde parte la relación implementa el comportamiento definido en el otro elemento relacionado. La realización indica trazabilidad entre los elementos. Esta relación se suele utilizar entre una clase y una interfaz, cuando la clase realiza o implementa el comportamiento definido por la interfaz.

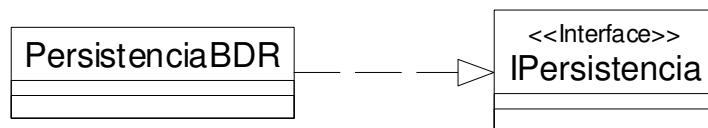


Fig. 31. Notación para representar la relación de realización entre clases en UML

- Contención: Esta relación muestra cómo el elemento fuente está contenida dentro del elemento objetivo.



Fig. 32- Notación para representar la relación de contención entre clases en UML

Lineamientos Generales:

Los diagramas de clases son los diagramas más comunes en el modelado de sistemas orientados a objetos. Se utilizan para describir la vista estática del sistema y soportan una ventaja fundamental del paradigma orientado a objetos: la rastreabilidad. Un diagrama de clases puede comenzarse a construir en el comienzo del desarrollo de software en la etapa de Requerimientos, estructurar en la etapa de Análisis, refinarse en la etapa de Diseño y finalizar al momento de realizar la implementación. Estos diagramas están compuestos por clases y sus relaciones que permiten a los objetos de las clases colaborar entre sí.

Uso del Diagrama:

- ↳ Para modelar el modelo de objetos de dominio del problema o la solución.
- ↳ Para modelar las clases que luego se mapearán en componentes de código, para la implementación del sistema.
- ↳ Para mantener rastreabilidad de la vista de la estructura a través de los diferentes modelos construidos en el proceso de desarrollo.
- ↳ Para mostrar estructuras de clases como las siguientes:
 - Las clases más importantes y sus relaciones.
 - Clases relacionadas funcionalmente.
 - Clases que pertenecen al mismo paquete.
 - Jerarquías de agregación importantes.
 - Estructuras importantes de objetos de entidad, incluyendo estructuras de clases con relaciones de asociación, agregación y generalización.
 - Paquetes y sus dependencias.
 - Clases que participan en una realización de caso de uso específica.
 - Una clase, sus atributos, operaciones y relaciones con otras clases.
- ↳ Para modelar el vocabulario de un sistema: Implica tomar decisiones sobre qué abstracciones son parte del sistema en consideración y cuales caen fuera de los límites. Se construye el Modelo de Objetos del Dominio.
- ↳ Para modelar colaboraciones simples: Sociedad de clases, interfaces y otros elementos que colaboran para proporcionar un comportamiento cooperativo mayor que la suma de todos los elementos.
- ↳ Para modelar la estructura del sistema: Incluye las clases del dominio del problema y las de los diagramas de interacción. Representa una vista estática del sistema.
- ↳ Para modelar un esquema lógico de base de datos: Un plano para el diseño conceptual de una base de datos. Muestra aquellas clases que deben ser persistentes.

Ejemplo:

A continuación, se muestra un ejemplo de un diagrama de clases para modelar el dominio. El mismo es una simplificación de la realidad, elaborada con fines didácticos.

Es una vista parcial de un Sistema de Gestión de Ventas y Reservas de Entradas para un Complejo de Cines; en él se muestran algunas clases que son representaciones del dominio del problema, que serán requeridas para desarrollar la funcionalidad que se presenta en el diagrama de casos de uso presentado anteriormente, en la Figura 16, en la sección de modelado con casos de uso. Se muestra las abstracciones más relevantes (clases) con los atributos y métodos que se consideran necesarios para satisfacer los requerimientos identificados. También se muestran las relaciones entre las clases con su navegabilidad, multiplicidad y el nombre de la relación.

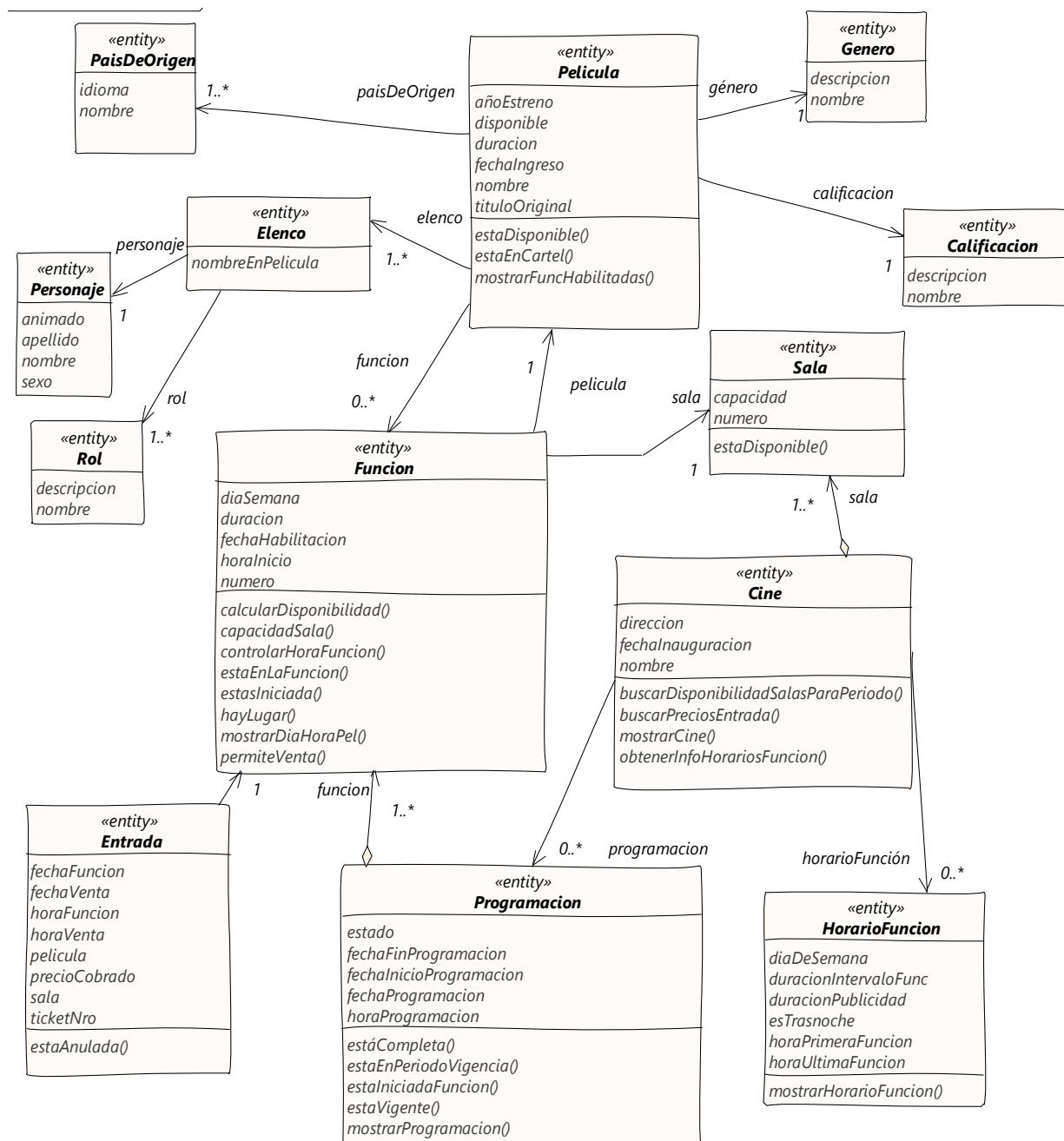


Fig. 33- Vista parcial del Modelo de Dominio para el caso de estudio del Complejo de Cines.

Diagrama de Máquina de Estados

Se denomina máquina de estados a un modelo de comportamiento de un sistema con entradas y salidas, en donde las salidas dependen no sólo de las señales de entradas actuales sino también de las anteriores. Las máquinas de estados se definen como un conjunto de estados que sirve de intermediario en esta relación de entradas y salidas, haciendo que el historial de señales de entrada determine, para cada instante, un estado para la máquina, de forma tal que la salida depende únicamente del estado y las entradas actuales.

Un diagrama que muestra una máquina de estados incluyendo estados simples, transiciones y estados anidados compuestos. Especifica la secuencia de estados en los que un objeto instancia de un clasificador puede estar, los eventos y condiciones que causan que los objetos alcancen esos estados, y las acciones que ocurren cuando esos estados son alcanzados.

Componentes del Diagrama de Máquina de Estado

Estados

Es una condición de un objeto en la cual el mismo ejecuta alguna actividad o espera un evento. El estado de un objeto varía a través del tiempo, pero en un punto particular está determinado por:

- El valor de los atributos del objeto;
- Las relaciones que tiene con otros objetos;
- Las actividades que está ejecutando.

Es importante identificar los estados que hacen una *diferencia* en el sistema que se está modelando. Debe existir una **diferencia semántica** que haga la “diferencia” entre los estados que justifique la molestia de modelarlos en una máquina de estados. Debe agregar valor al modelo.



Fig. 34-. Notaciones para representar estados en UML.

Los estados contienen:

- **Nombre:** es una cadena de texto que lo distingue de otros estados. Un estado puede ser anónimo.
- **Efectos de Entrada-Salida:** Acciones que se ejecutan en la entrada y salida del estado respectivamente.
- **Transiciones Internas:** Son manejadas sin causar un cambio de estado.

Estados Iniciales y Finales: son dos estados especiales que pueden definirse en una máquina de estados. El **estado inicial**, que indica el punto de inicio por defecto para una máquina de estado o subestado. Se representa con un circulo negro relleno. El **estado final**, es representado como un circulo negro relleno con un circulo de linea negra alrededor e indica que la ejecución de la máquina de estado o del estado asociado ha sido completada.

Estado de Historia: Es muy común que se encuentre la siguiente situación cuando se está modelando una máquina de estado:

- Está en un subestado A de un estado compuesto.
- Se realiza una transición fuera del estado compuesto (por lo tanto, fuera del subestado A).
- Se va a uno o más estados externos.
- Se vuelve con una transición al estado compuesto, pero se desea continuar en el subestado donde se encontraba al momento de salir previamente.

Claramente el estado compuesto necesita una forma de recordar en cual subestado estaba al momento de dejar el estado compuesto. UML utiliza el pseudo estado **historia**, para manejar esto. A menos que se especifique de otra manera, cuando una transición entra a un estado compuesto, la acción de la máquina de estados anidada comienza nuevamente en el estado inicial (a menos que la transición apunte a un subestado directamente). Los estados de historia permiten a la máquina de estados re ingresar al último subestado que estuvo activo antes de abandonar el estado compuesto.

Transición

Es una relación entre dos estados que indica que un objeto en el primer estado ejecutará ciertas acciones y entrará en el segundo estado cuando ocurra un evento especificado y las condiciones indicadas sean satisfechas.

Una transición puede tener:

- **Estado Origen:** Es el estado afectado por la transición; si un objeto está en este estado, una transición saliente puede ser disparada cuando un objeto recibe el evento disparador de la transición y si la condición de control (si existe alguna), se cumple.
- **Evento Disparador:** El evento que hace a la transición apta para ser disparada (si la condición de control es satisfecha) cuando es recibida por el objeto en el estado origen.
- **Condición de Control:** Es una expresión booleana que se evalúa cuando la transición es disparada tras la recepción de un evento disparador.
- **Efecto:** Un comportamiento ejecutable, tal como una acción, que puede actuar sobre el objeto que posee la máquina de estado, e indirectamente sobre otros objetos que son visibles a él.
- **Estado Destino:** El estado activo luego de completarse la transición.

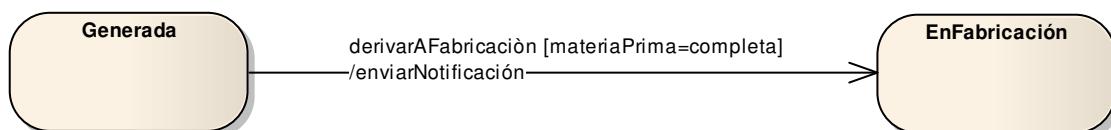


Fig. 35- Transición entre estados en un Diagrama de Máquina de Estados.

¿Cómo se construye?:

Se describen a continuación algunos lineamientos referidos a la construcción de los diagramas de máquina de estado:

- Identificar el o los estados de creación (estados iniciales) y los estados finales.
- Buscar que otros estados podrán pasar los objetos de la clase en su vida.
- Se puede encontrar estados en los valores límite de los atributos, por ejemplo: ¿Qué pasa si se llena el cupo de un seminario? Qué esté lleno hace que apliquen reglas diferentes para el objeto.

- Luego de haber encontrado todos los estados posibles, se comienza con las transiciones, para cada estado, preguntarse como el objeto puede salir de ese estado, si es que puede (no es un estado final).
- Como todas las transiciones tienen estado inicial y uno final, evaluar si no es necesaria la incorporación de un estado nuevo.
- Considerar los estados recursivos, transiciones cuyo estado inicial y final es el mismo.
- Luego analizar los métodos identificados en el diagrama de clases para la clase que se está modelando, algunos de ellos corresponderán con una transición en el diagrama de transición de estados.
- Si bien los diagramas de máquina de estado se pueden heredar, considerar que, si bien las subclases son similares a las superclases, *aún son diferentes*, lo que significa que se debe reconsiderar el diagrama de máquina de estado en la subclase. Lo más probable es que aparezcan nuevos estados y transiciones en las clases hijas o que sea necesario re-definir algunos.

Lineamientos Generales:

En el apartado siguiente, se presentan algunos lineamientos generales a considerar en el momento de modelar con diagramas de máquina estados:

- **Crear un diagrama de máquina de estado cuando el comportamiento de los objetos de una clase, difiere en función de los estados:** Si el elemento clase o componente tiene un comportamiento que no varía conforme cambien los estados entonces no tiene sentido la construcción de una máquina de estado.
- **Considerar que, en las aplicaciones de negocio, un porcentaje reducido de las clases requieren un diagrama de máquina de estado:** Estos diagramas son mucho más comunes en sistemas de tiempo real.
- **Ubicar el estado inicial en el borde superior izquierdo:** Esto responde al enfoque de lectura de izquierda a derecha, de arriba hacia abajo que las personas en la cultura occidental utilizan.
- **Ubicar el estado final en el borde inferior derecho:** Esto responde al enfoque de lectura de izquierda a derecha, de arriba hacia abajo que las personas en la cultura occidental utilizan.
- **El nombre del estado debería ser simple pero descriptivo:** Esto incrementa el valor de la comunicación. Idealmente el nombre de los estados debe estar en presente o en participio, por ejemplo: *Generado* o *Registrado*.
- **Cuestionarse los Estados “Agujero Negro”:** son los estados a los que les llegan transiciones, una o más, pero ninguna transición sale de él. Esta es una pauta de que faltan una o más transiciones. Esto es válido para los puntos finales.
- **Cuestionarse los Estados “Milagro”:** son los estados de los que salen transiciones, una o más, pero ninguna entra a él. Esto es válido únicamente para los puntos de inicio. Esta es una pauta de que faltan una o más transiciones.
- **Modelar Subestados para manejar la complejidad:** Los subestados tienen sentido cuando los estados existentes exhiben un comportamiento complejo, por ello motiva a explorar sus subestados. Introducir un super estado tiene sentido cuando algunos estados existentes comparten una condición de entrada o salida común.
- **Agregar transiciones comunes de los subestados:** Para simplificar el diagrama se debe utilizar la misma transición. Para poder unir las transiciones, es necesario que todas tengan las mismas acciones y las mismas condiciones de guarda.

- **Crear Niveles de Diagramas de Máquina de Estado para entidades muy complejas:** Cuando algún diagrama de máquina de estado tenga varios estados complejos, que necesiten modelarse con subestados, es más conveniente plantear varios niveles de diagramas con una vista general y otros con vistas más detalladas.
- **Siempre incluir estados iniciales y finales en los Diagramas de Máquina de Estados:** Es necesario incluir todos los estados completos del ciclo de vida de la “entidad”, su “nacimiento” y su eventual “muerte” en los diagramas de alto nivel. En los diagramas de nivel más bajo puede no siempre incluir estados iniciales y finales, particularmente los diagramas que modelan los estados “del medio” del ciclo de vida de la entidad.
- **Modelar transiciones recursivas únicamente cuando se desea salir y reingresar al mismo estado:** En este caso las operaciones invocadas para las acciones de entrada o salida podrían ser invocadas automáticamente.

Uso del Diagrama:

- Modela el comportamiento de un objeto a través de su tiempo de vida, para esto se debe:
- Establecer el contexto de la máquina de estado: una clase, un caso de uso, el sistema completo.
- Establecer los estados de inicio y fin para el objeto. Si hay pre o post condiciones para estos estados también deben definirse.
- Determinar los eventos a los cuales el estado debe responder.
- Disponer los estados de alto nivel en los que pueda estar el objeto comenzando por el estado de inicio al estado final.
- Conectar esos estados con las transiciones disparadas por los eventos apropiados.
- Identificar las acciones de entrada o salida.
- Expandir o simplificar la máquina de estados usando subestados.
- Controlar que todos los eventos disparados por las transiciones concuerden con los esperados por las interfaces o protocolos realizados por el objeto y que todos los eventos esperados por las interfaces o protocolos del objeto sean manejados por la máquina de estados.
- Controlar que todas las acciones en la máquina de estados sean soportadas por relaciones, métodos, y operaciones del objeto que las encierra.
- Realizar un seguimiento de la máquina de estados, comparándolo con las secuencias esperadas de eventos y sus respuestas. Buscar estados inalcanzables y estados en los que la máquina se atasque.

Ejemplos:

Si continuamos con el caso de estudio del complejo de cines, una de las clases que se identificaron como parte del dominio del problema es la **Sala**: del análisis realizado se detectó que los objetos de la clase Sala tienen comportamientos que varían de acuerdo al estado en el que se encuentra. Los estados relevantes identificados para esta clase y las transiciones permitidas entre ellos, se muestran en el siguiente diagrama de máquina estados:

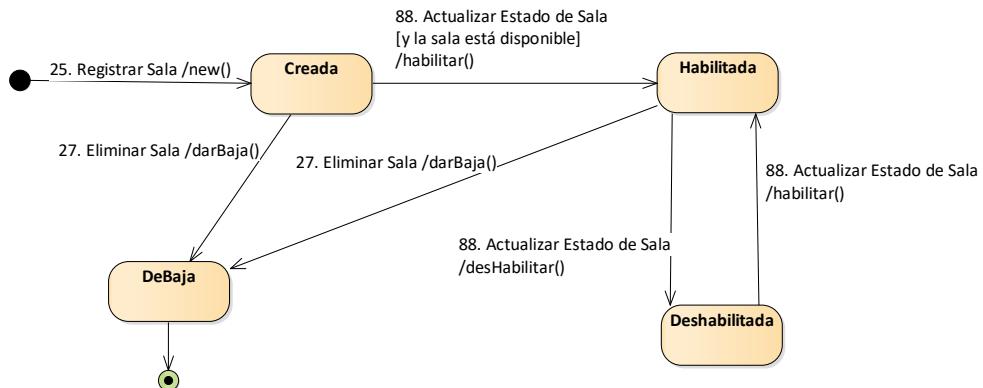


Fig. 36-. Diagrama de Máquina de estados para la clase Sala, utilizada en el caso del Complejo de Cines

Para exemplificar los conceptos de Estado Compuesto e historia, utilizaremos como ejemplo la clase Estructura, que muestra el ciclo de vida para los objetos de esa clase, desde que se genera el pedido de fabricación hasta finaliza. El estado “EnFabricacionDeEstructura”, es un estado compuesto y dada su complejidad, requiere una máquina de estado interna, que modela todos los estados por los que transita durante la fabricación; estos estados son los que necesitamos que tengan historia, dado que si el proceso de fabricación se interrumpe, necesitamos al continuar luego de la interrupción que vuelva al estado en el que estaba justo antes de la interrupción, esto es lo que se indica con la “H”.

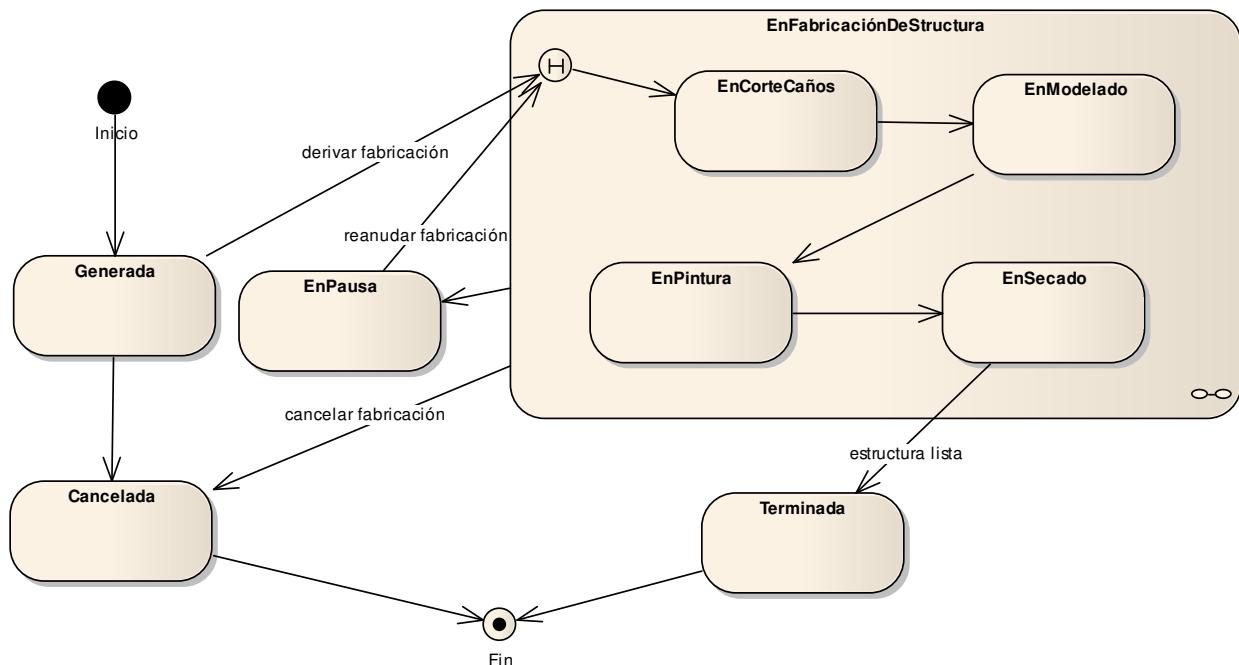


Fig. 37- Diagrama de Máquina de estados que muestra el uso de una sub-máquina y historia para los estados

Diagrama de Secuencia

Un diagrama de secuencia muestra la interacción de un conjunto de instancias de un clasificador a través del tiempo como una representación estructurada del comportamiento a través de una serie de pasos secuenciales.

En síntesis, muestra un conjunto de líneas de vida y los mensajes enviados y recibidos por estas, enfatizando el orden de los mensajes en función del tiempo.

Se modela como colaboran e interactúan las líneas de vida mediante el paso de mensajes a lo largo del tiempo para conseguir un resultado.

Al ser un diagrama de interacción, éste demuestra cómo los objetos cooperan para realizar los diferentes comportamientos definidos en un caso de uso. La secuencia de los mensajes queda representada por el orden de estos.

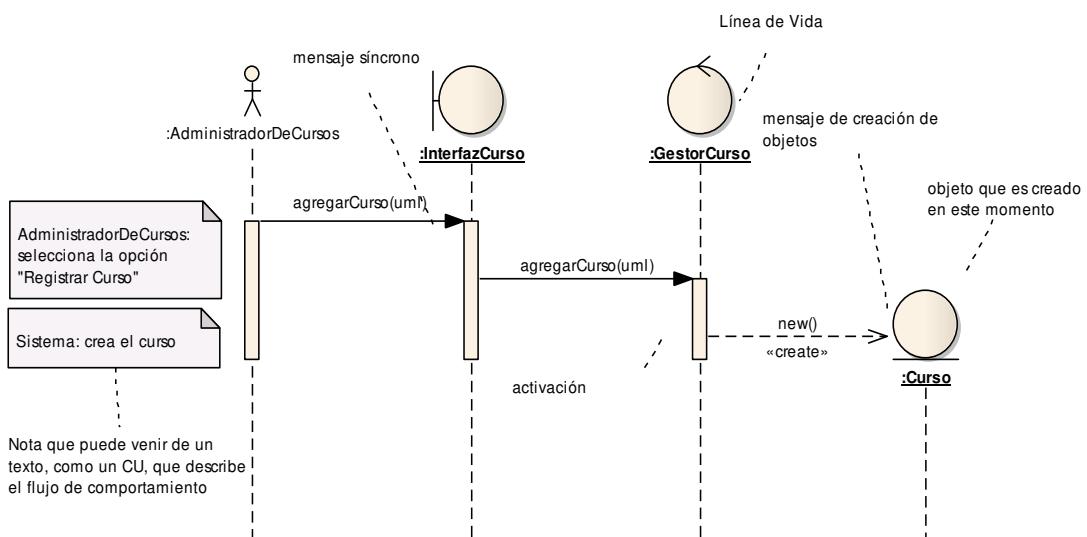


Fig. 38-. Elementos de modelado utilizados en un Diagrama de Secuencia

Por otra parte, los diagramas de secuencia y de comunicación describen información similar, y con ciertas transformaciones, pueden ser transformados unos en otros sin dificultad. Algunas características particulares de éste diagraman que lo diferencia del diagrama de comunicación son:

- Estos diagramas enfatizan en la línea de vida que representa la existencia de un objeto a lo largo del tiempo. Entonces, es posible indicar cuando los objetos son creados y destruidos a lo largo del tiempo.
- Un diagrama de secuencia no muestra explícitamente los enlaces entre objetos.
- Tampoco se representa explícitamente el número de secuencia de los mensajes –producto de la sucesión temporal de estos.

Componentes del Diagrama de Secuencia

Líneas de Vida: representa un participante en una interacción –es una instancia de un clasificador específico (clases, clases activas, interfaces, componentes, actores y nodos) que participa en una interacción. Un ejemplo de línea de vida es un *objeto*, que constituye una instancia del clasificador *clase*. Las líneas de vida se representan mediante un ícono representativo del tipo y además poseen una línea vertical punteada cuando se modelan en un Diagrama de Secuencia.

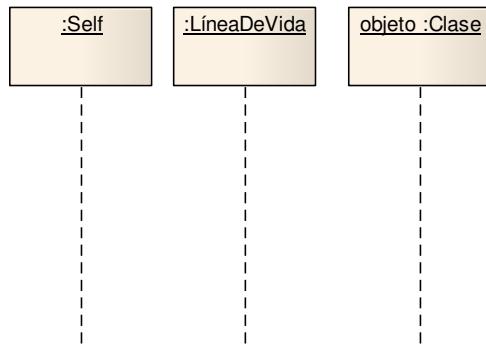


Fig. 39- Líneas de vida de los objetos en un Diagrama de Secuencia

Mensajes: es un tipo específico de comunicación entre dos líneas de vida en una interacción; puede representar:

- la *llamada a una operación*, invocando un mensaje específico.
- la *creación/destrucción de una instancia*, invocando un mensaje de creación o destrucción.
- el envío de una señal.

Cuando una línea de vida recibe un mensaje, éste constituye la invocación de una determinada operación que tiene la misma firma (signatura) que el mensaje. Por lo tanto, deberá existir una correspondencia entre la firma (signatura) del mensaje y de la operación del clasificador. Al recibir el mensaje se inicia una actividad en el objeto receptor que se llama *ocurrencia de ejecución*. Una ocurrencia de ejecución muestra la línea de vida de los objetos que la ejecutan, en un momento de tiempo. Un objeto activado, o bien está ejecutando su propio código o está esperando el retorno de otro objeto al cual se le ha enviado un mensaje. Esta ocurrencia de ejecución es opcional en el diagrama.

Un mensaje se grafica como una flecha horizontal entre dos líneas de vida.

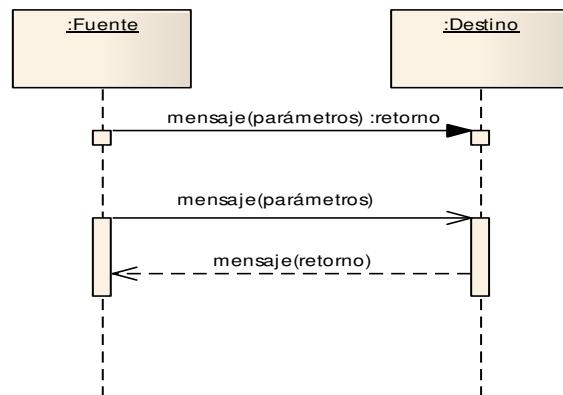


Fig. 40- Tipos de mensajes que pueden utilizarse en un Diagrama de Secuencia

El formato de la flecha variará según el tipo de mensaje:

Sintaxis	Nombre	Descripción
unMensaje(unParámetro)	Mensaje Síncrono	El emisor espera que el receptor finalice la ejecución del mensaje.
unMensaje(unParámetro)	Mensaje Asíncrono	El emisor envía el mensaje y continúa ejecutando –éste no espera que el retorno del receptor.
<-----	Mensaje de Retorno	El receptor de un mensaje enviado retorna el foco de control al emisor.
	Creación de Objeto	El emisor crea una instancia del clasificador especificado por el receptor. Se puede graficar sin necesidad de indicar el mensaje creador – sólo con el estereotipo «create».
	Destrucción de Objeto	El emisor destruye la instancia del receptor. Este mensaje no retorna valor.

Tabla 8 – Tipos de Mensajes que pueden utilizarse para modelar diagramas de secuencia.

¿Cómo se construye?:

Un diagrama de secuencia sencillo se crea a través de una serie de pasos y luego se podrá refinar utilizando adornos de modelado más avanzados que proporcionan mayor detalle y compresión al lector:

- En primer lugar, cada línea de vida que participa en la interacción será colocada a lo largo del eje X a la altura temporal representativa de su creación. Normalmente se ubica a la izquierda el objeto que inicia la interacción y los objetos subordinados a la derecha.

Entonces si este diagrama fuera una realización de caso de uso de análisis, estaríamos ubicando:

- Un objeto Actor que representa el usuario que inicia el flujo de eventos.
- Un objeto de Interfaz que representan la pantalla con la cual interactúa el usuario.
- Un objeto de Control que se ocupa de organizar y controlar la lógica requerida para alcanzar el objetivo del caso de uso.
- Un objeto de Entidad para cada una de las entidades de dominio requeridas para realizar el comportamiento definido en el caso de uso.

Cada línea de vida se graficará como se mostró anteriormente con una cabecera representativa del tipo y una línea discontinua vertical (indicadora del tiempo durante el cual la instancia existe e interviene en las interacciones).

- Luego se colocarán los mensajes que esos objetos envían y reciben a lo largo del eje Y, ordenándolos cronológicamente de arriba hacia abajo.

Cada mensaje se graficará como una flecha cuyos extremos variarán de forma según el tipo de mensaje (ver tabla anterior). La interpretación del transcurso del tiempo debe realizarse desde arriba hacia abajo.

Creación y Destrucción de Líneas de Vida

Una línea de vida es creada y destruida durante el transcurso del tiempo, mediante el envío de mensajes de creación o de destrucción, respectivamente.

La creación de una línea de vida se grafica dibujando la cabecera a la altura del mensaje enviado por el objeto creador.

La destrucción de una línea de vida se indica por medio de un símbolo terminal, representado por una cruz.

En el siguiente diagrama se muestra la creación y destrucción de un objeto.

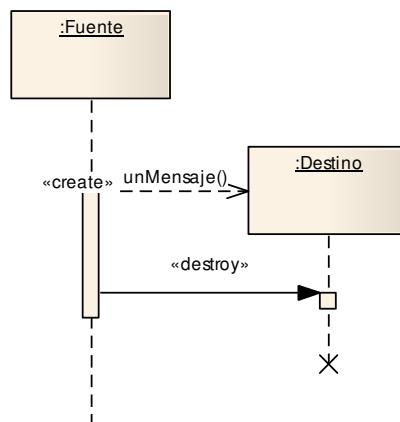


Fig. 41- Creación y destrucción de objetos en un Diagrama de Secuencia

Parámetros

Estos proveen valores específicos para el uso de la interacción. Permiten parametrizar la interacción en cada ocurrencia. Para escribirlos se utiliza la notación de parámetros establecida por UML. En la figura se muestra una interacción que retorna un valor que luego es asignado a una variable.

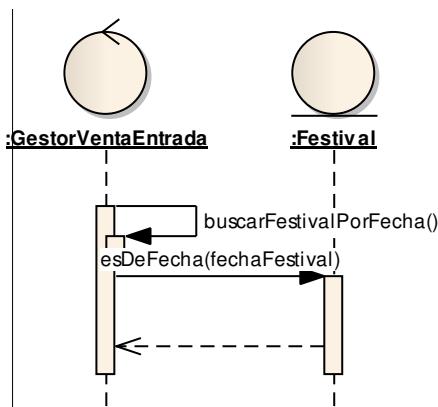


Fig. 42- Parámetros en un Diagrama de Secuencia

Lineamientos Generales:

En el apartado siguiente, se presentan algunos lineamientos generales a considerar en el momento de modelar con diagramas de secuencia:

- **Ubicar los mensajes ordenados de izquierda a derecha:** Se debe comenzar el flujo de mensajes desde la esquina superior izquierda; un mensaje que aparezca más abajo es enviado después de uno que aparezca arriba de éste. Consistentemente con la forma en que leemos hay que tratar de organizar los clasificadores (actores, clases, objetos y casos de uso) a lo largo del borde superior del diagrama de manera tal que representa el flujo de mensajes de izquierda a derecha. A veces no es posible para absolutamente todos los mensajes organizarlos de izquierda a derecha; por ejemplo, es común para pares de objetos que invocan métodos unos del otro.
- **Nombrar los objetos sólo cuando se refieren en mensajes:** Los objetos en los diagramas de secuencia tienen etiquetas en el formato estándar de UML “nombre: NombreDeLaClase”, donde “nombre” es opcional (si no se les especificó un nombre se los denomina objetos anónimos). Un objeto necesita ser nombrado cuando es referenciado como un valor de retorno a un mensaje, en cambio una clase que no necesita ser referenciada en ningún otro lado del diagrama puede ser anónima.
- **Nombrar a los objetos cuando existen varios del mismo tipo:** Cuando el diagrama incluya varios objetos del mismo tipo, se debería especificar el nombre a todos los objetos de ese tipo a fin de crear un diagrama sin ambigüedades. Por ejemplo, en un sistema de transferencia bancaria se deberían identificar al objeto cuenta Origen del objeto cuenta Destino, quedando las respectivas líneas de vida de la siguiente manera:
 - *origen: Cuenta destino: Cuenta*
- **Concentrarse en las interacciones críticas:** Se aconseja concentrarse en los aspectos críticos del sistema -no en los detalles superfluos- cuando se crea un modelo. Entonces logrará un modelo bien hecho, incrementando tanto la productividad como la legibilidad de los diagramas. Por ejemplo, si se trabaja con entidades de negocio lógicas, no es necesario que se incluyan los detalles de interacción para almacenarlas en la base de datos; será suficiente con los métodos guardar() o eliminar().
- **Modelar Escenarios de Uso:** un escenario de uso es una descripción de una forma potencial en la que el sistema es utilizado. La lógica de uso de un escenario puede ser una parte de un caso de uso, tal vez un curso alternativo. También puede ser el modelado de un curso completo de acción o una porción de un curso básico de acciones más uno o más escenarios alternativos. La lógica de uso de un escenario también puede pasar por la lógica contenida en varios casos de uso. Por ejemplo, un estudiante se inscribe en una universidad y luego inmediatamente se inscribe en tres seminarios.
- **Modelar la lógica de métodos:** explorar la lógica de una operación compleja, de una función o un procedimiento. Una forma de pensar los diagramas de secuencia, particularmente los diagramas altamente detallados son como código visual de un objeto.
- **Validar y profundizar la lógica y completitud de un escenario de uso,** que puede ser parte de un caso de uso, por ejemplo, un curso normal o básico de acción, o una pasada por la lógica contenida en varios casos de uso.
- **Mostrar la interacción temporal (paso de mensajes) entre las distintas instancias.**

Ejemplo de Diagrama de Secuencia

Este diagrama modela el escenario del curso normal, descripto anteriormente, para el caso de uso Registrar Película del Complejo de Cines.

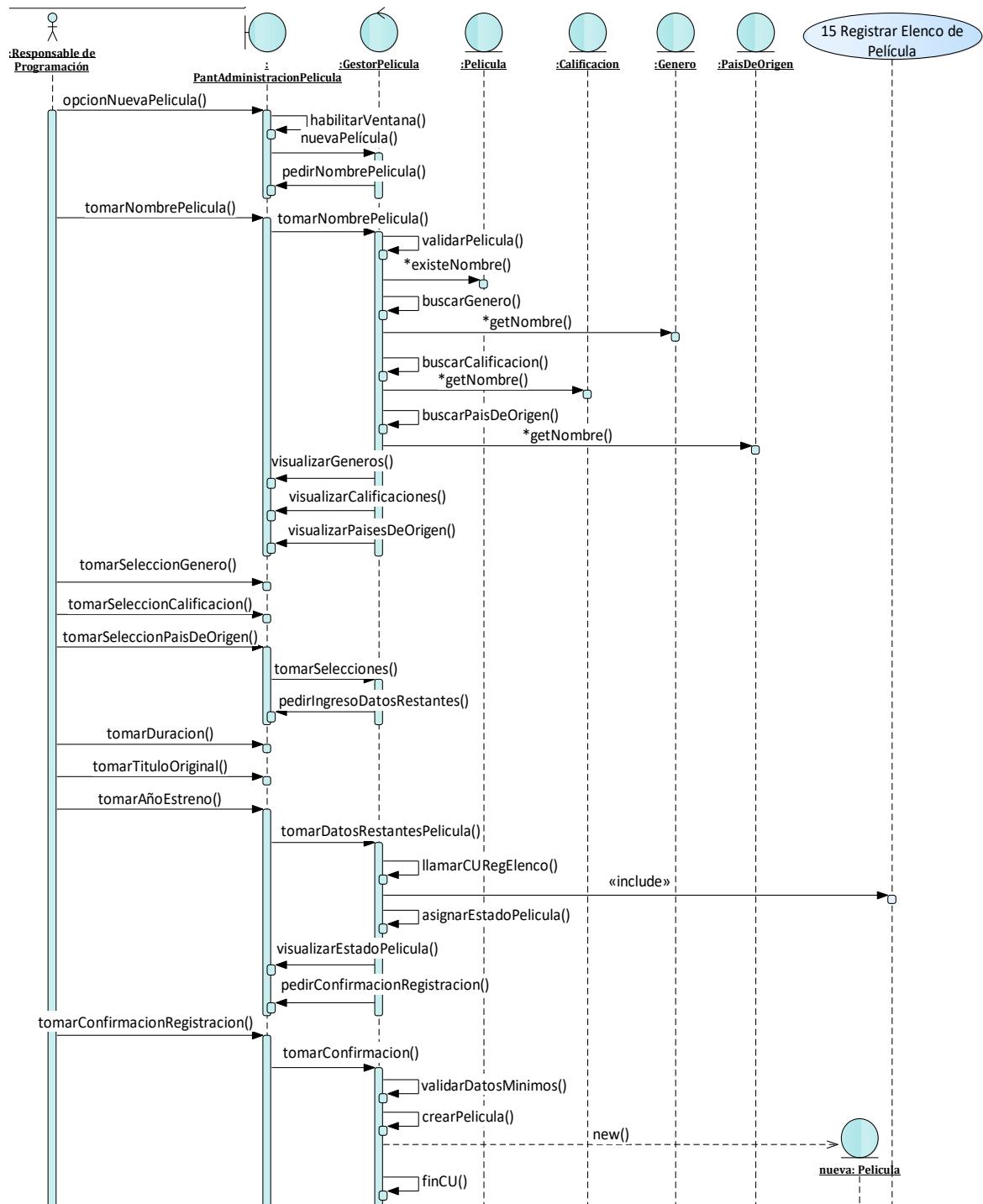


Fig. 43-. Diagrama de secuencia para el escenario del curso normal del caso de uso Registrar Película

A continuación se muestra la vista del diagrama de clases relacionado con el diagrama de secuencia presentado anteriormente:

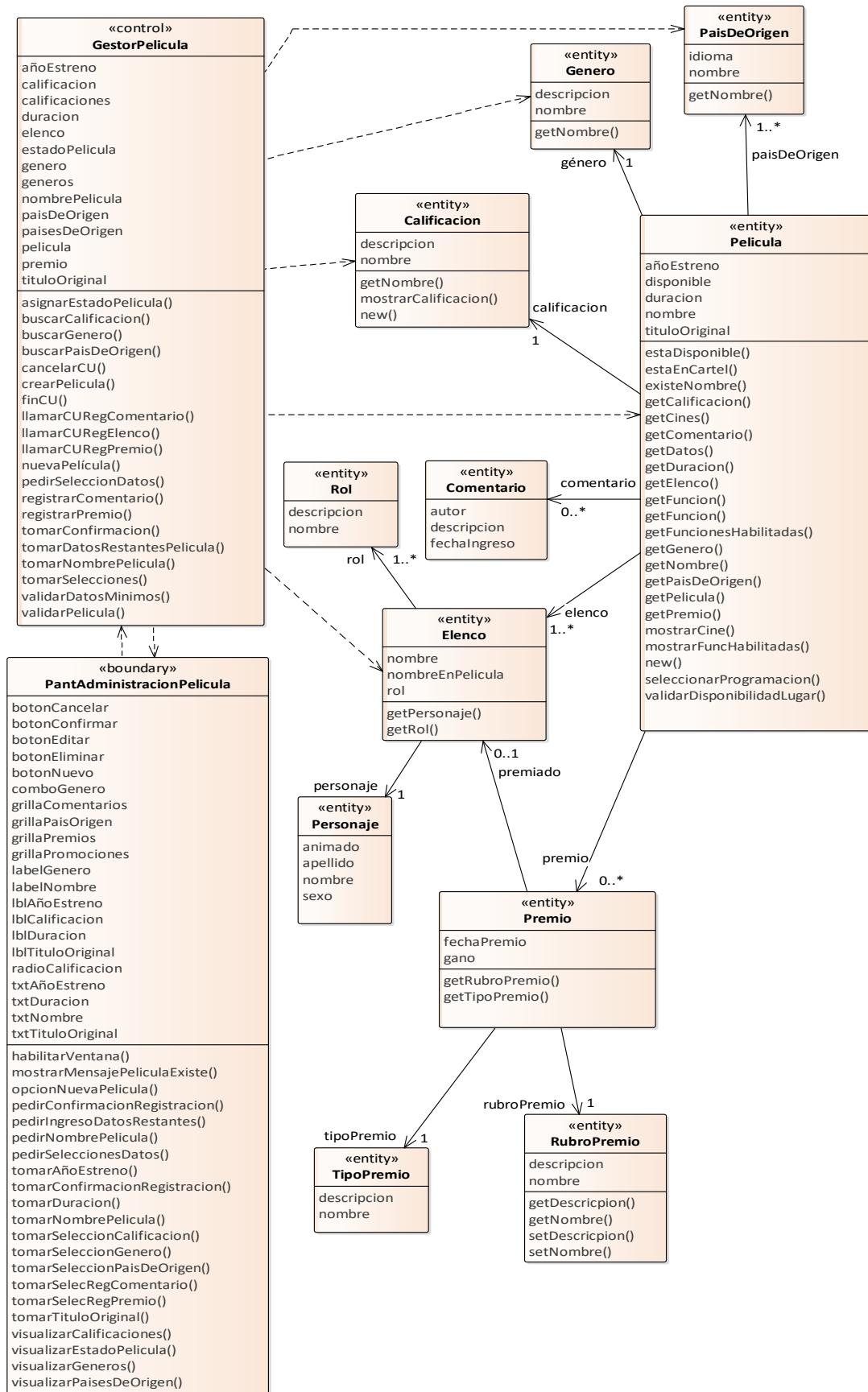


Fig. 44: Vista del Diagrama de clases que muestra las clases necesarias para modelar el escenario del curso normal del caso de uso Registrar Película

Lenguaje de Programación Orientada a Objetos

A partir de este punto, habiendo analizado las características del paradigma orientado a objetos y el lenguaje estándar que se utiliza para modelar (UML), estudiaremos el lenguaje de programación Java, que está basado en el paradigma orientado a objetos que estudiamos anteriormente.

Surgimiento del Lenguaje

Java surgió en 1991 cuando un grupo de ingenieros de la empresa Sun Microsystems trataron de diseñar un nuevo lenguaje de programación destinado a electrodomésticos. La reducida potencia de cálculo y memoria de los electrodomésticos llevó a desarrollar un lenguaje sencillo capaz de generar código de tamaño muy reducido.

Debido a la existencia de distintos tipos de CPUs y a los continuos cambios, era importante conseguir una herramienta independiente del tipo de CPU utilizada. Es por esto que desarrollan un código “neutro” que no depende del tipo de electrodoméstico, el cual se ejecuta sobre una “máquina hipotética o virtual” denominada Java Virtual Machine (JVM). Es la JVM quien interpreta el código neutro convirtiéndolo a código particular de la CPU utilizada. Esto permitía lo que luego se ha convertido en el principal lema del lenguaje: “Write Once, Run Everywhere”. A pesar de los esfuerzos realizados por sus creadores, ninguna empresa de electrodomésticos se interesó por el nuevo lenguaje. Java, como lenguaje de programación para computadoras, se introdujo a finales de 1995. Si bien su uso se destaca en la Web, sirve para crear todo tipo de aplicaciones (locales, intranet o internet).

Entendiendo Java

Java es un lenguaje compilado e interpretado; esto significa que el programador escribirá líneas de código utilizando un determinado programa que se denomina IDE (Ambiente de Desarrollo Integrado; Integrated Development Environment por sus siglas en inglés). Este programa le provee al programador un espacio de trabajo para crear código que se almacena en archivos de formato .java.

Luego, para poder correr el código y ejecutar el programa creado, se debe realizar primero lo que se denomina Compilación; para eso un programa denominado compilador será el encargado de revisar la sintaxis del código y si está correcto transformará el archivo .java en otro archivo de formato .class que se denominan: “bytecodes”. Una vez realizada la compilación, será la Máquina Virtual (Java Virtual Machine - JVM), la encargada de leer los archivos .class y transformarlos en el código entendido por la CPU de la computadora.

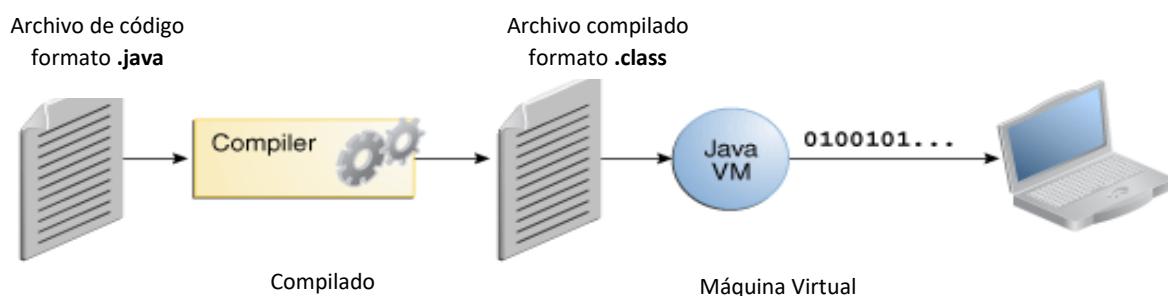


Fig. 45- Funcionamiento general del Lenguaje Java

El entorno de desarrollo de Java

Existen distintos programas comerciales que permiten desarrollar código Java. Oracle, quien compró a Sun (la creadora de Java), distribuye gratuitamente el Java Development Kit (JDK). Se trata de un conjunto de programas y librerías que permiten desarrollar, compilar y ejecutar programas en Java. Incorpora además la posibilidad de ejecutar parcialmente el programa, deteniendo la ejecución en el punto deseado y estudiando en cada momento el valor de cada una de las variables (es el denominado Debugger). Es tarea muy común de un programador realizar validaciones y pruebas del código que construye, el debugger permite realizar una prueba de escritorio automatizada del código, lo cual ayuda a la detección y corrección de errores. En el momento de escribir este trabajo las herramientas de desarrollo: JDK, van por la versión 1.8. Estas herramientas se pueden descargar gratuitamente de <http://www.oracle.com/technetwork/java>.

Los IDEs (Integrated Development Environment), tal y como su nombre indica, son entornos de desarrollo integrados. En un mismo programa es posible escribir el código Java, compilarlo y ejecutarlo sin tener que cambiar de aplicación. Algunos incluyen una herramienta para realizar Debug gráficamente, frente a la versión que incorpora el JDK basada en la utilización de una Consola bastante difícil y pesada de utilizar. Estos entornos integrados permiten desarrollar las aplicaciones de forma mucho más rápida, incorporando en muchos casos librerías con componentes ya desarrollados, los cuales se incorporan al proyecto o programa. Estas herramientas brindan una interfaz gráfica para facilitar y agilizar el proceso de escritura de los programas. El entorno elegido para este módulo, por su relevancia en el mercado y su licencia gratuita se denomina Netbeans. El mismo se encuentra disponible para descargar de forma gratuita en www.netbeans.org. En la Guía Práctica abordaremos la forma de instalación y configuración de estas herramientas.

El compilador de Java

Se trata de una de las herramientas de desarrollo incluidas en el JDK. Realiza un análisis de sintaxis del código escrito en los ficheros fuente de Java (con extensión *.java). Si no encuentra errores en el código genera los ficheros compilados (con extensión *.class). En otro caso muestra la línea o líneas erróneas. En el JDK de Sun dicho compilador se llama javac.exe si es para el sistema operativo Windows.

La Java Virtual Machine

Tal y como se ha comentado al comienzo, la existencia de distintos tipos de procesadores y ordenadores llevó a los ingenieros de Sun a la conclusión de que era muy importante conseguir un software que no dependiera del tipo de procesador utilizado. Se plantea la necesidad de conseguir un código capaz de ejecutarse en cualquier tipo de máquina. Una vez compilado no debería ser necesaria ninguna modificación por el hecho de cambiar de procesador o de ejecutarlo en otra máquina. La clave consistió en desarrollar un código “neutro” el cual estuviera preparado para ser ejecutado sobre una “máquina hipotética o virtual”, denominada Java Virtual Machine (JVM). Es esta JVM quien interpreta este código neutro convirtiéndolo a código particular de la CPU o chip utilizada. Se evita tener que realizar un programa diferente para cada CPU o plataforma.

La JVM es el intérprete de Java. Ejecuta los “bytecodes” (ficheros compilados con extensión *.class) creados por el compilador de Java (javac.exe). Tiene numerosas opciones entre las que destaca la posibilidad de utilizar el denominado JIT (Just-In-Time Compiler), que puede mejorar entre 10 y 20 veces la velocidad de ejecución de un programa.

Características de Java

Java es un lenguaje:

- **Orientado a objetos:** Al contrario de otros lenguajes como C++, Java no es un lenguaje modificado para poder trabajar con objetos, sino que es un lenguaje creado originalmente para trabajar con objetos. De hecho, todo lo que hay en Java son objetos.
- **Independiente de la plataforma:** Debido a que existen máquinas virtuales para diversas plataformas de hardware, el mismo código Java puede funcionar prácticamente en cualquier dispositivo para el que exista una JVM.
- **Compilado e Interpretado:** La principal característica de Java es la de ser un lenguaje compilado e interpretado. Todo programa en Java ha de compilarse y el código que se genera **bytecode** es interpretado por una máquina virtual, como se vio anteriormente. De este modo se consigue la independencia de la máquina: el código compilado se ejecuta en máquinas virtuales que sí son dependientes de la plataforma. Para cada sistema operativo distintos, ya sea Microsoft Windows, Linux, OS X, existe una máquina virtual específica que permite que el mismo programa Java pueda funcionar sin necesidad de ser recompilado.
- **Robusto:** Su diseño contempla el manejo de errores a través del mecanismo de Excepciones y fuerza al desarrollador a considerar casos de mal funcionamiento para reaccionar ante las fallas.
- **Gestiona la memoria automáticamente:** La máquina virtual de Java gestiona la memoria dinámicamente como veremos más adelante. Existe un **recolector de basura** que se encarga de liberar la memoria ocupada por los objetos que ya no están siendo utilizados.
- **No permite el uso de técnicas de programación inadecuadas:** Como veremos más adelante, todo en Java se trata de objetos y clases, por lo que para crear un programa es necesario aplicar correctamente el paradigma de objetos.
- **Multihilos (multithreading):** Soporta la creación de partes de código que podrán ser ejecutadas de forma paralela y comunicarse entre sí.
- **Cliente-servidor:** Java permite la creación de aplicaciones que pueden funcionar tanto como clientes como servidores. Además, provee bibliotecas que permiten la comunicación, el consumo y el envío de datos entre los clientes y servidores.
- **Con mecanismos de seguridad incorporados:** Java posee mecanismos para garantizar la seguridad durante la ejecución comprobando, antes de ejecutar código, que este no viola ninguna restricción de seguridad del sistema donde se va a ejecutar. Además, posee un *gestor de seguridad* con el que puede restringir el acceso a los recursos del sistema.
- **Con herramientas de documentación incorporadas:** Como veremos más adelante, Java contempla la creación automática de documentación asociada al código mediante la herramienta **Javadoc**.



Aplicaciones de java

Java es la base para prácticamente todos los tipos de aplicaciones de red, además del estándar global para desarrollar y distribuir aplicaciones móviles y embebidas, juegos, contenido basado en web y software de empresa. Java⁸ se encuentra aplicado en un amplio rango de dispositivos desde portátiles hasta centros de datos, desde consolas para juegos hasta súper computadoras, desde teléfonos móviles hasta Internet.

- El 97% de los escritorios empresariales ejecutan Java.
- Es la primera plataforma de desarrollo.
- 3 mil millones de teléfonos móviles ejecutan Java.
- El 100% de los reproductores de Blu-ray incluyen Java.
- 5 mil millones de Java Cards en uso.
- 125 millones de dispositivos de televisión ejecutan Java.

Antes de comenzar

Como se detalló en el apartado “Entendiendo Java”, los archivos de código escritos por el programador que se denominan: *archivos de código fuente*, en Java son los archivos .java. Éstos se podrían crear utilizando simplemente un editor de texto y guardándolo con la extensión .java. Por lo tanto, esto se puede hacer utilizando cualquier editor de texto como el bloc de notas de Windows. Para facilitar la tarea de desarrollo en la práctica, como vimos, se utilizan IDEs (entornos de desarrollo integrados) que ofrecen herramientas como coloreado de palabras clave, análisis de sintaxis en tiempo real, compilador integrado y muchas otras funciones que usaremos.

Al ser un lenguaje multiplataforma y especialmente pensado para su integración en redes, la codificación de texto utiliza el estándar Unicode, lo que implica que los programadores y programadoras hispanohablantes podemos utilizar sin problemas símbolos de nuestra lengua como la letra “ñ” o las vocales con tildes o diéresis a la hora de poner nombre a nuestras variables.

Algunos detalles importantes son:

- En java (como en C) hay diferencia entre mayúsculas y minúsculas por lo que la variable nombrecompleto es diferente a nombreCompleto.
- Cada línea de código debe terminar con un; (punto y coma)
- Una instrucción puede abarcar más de una línea. Además, se pueden dejar espacios y tabuladores a la izquierda e incluso en el interior de la instrucción para separar elementos de la misma.
- A veces se marcan bloques de código, es decir código agrupado. Cada bloque comienza con llave que abre, "{" y termina con llave que cierra, "}"

⁸ Datos obtenidos en <https://www.java.com/es/about/>

Declaración de Clases

En el caso más general, la declaración de una clase puede contener los siguientes elementos:

```
[public] [final | abstract] class NombreDeLaClase [extends ClaseMadre] [implements Interfase1 [, Interfase2 ]...]
```

Donde las porciones encerradas entre corchetes son opcionales a optar entre las posibilidades separadas por la barra vertical.

O bien, para interfaces:

```
[public] interface NombreDeLaInterface [extends InterfaceMadre1 [, InterfaceMadre2 ]...]
```

Como se ve, lo único obligatorio es la palabra reservada `class` y el nombre que queramos asignar a la clase. Las interfaces son un caso de tipo particular que veremos más adelante.

De esta forma, podríamos declarar la clase `Perro` como sigue:

```
public class Perro
{
    // esto es un comentario
    // aquí completaremos luego el cuerpo de la clase
}
```

De esta forma una instancia (objeto) de la clase `Perro` puede declararse utilizando el operador `new` de la siguiente forma, ocupando un espacio en memoria que luego podrá ser accedido mediante el nombre `miPerro`.

```
Perro miPerro = new Perro();
```

En el ejemplo anterior podemos identificar, además de la declaración de la clase `Perro` como de tipo `public` (veremos luego su significado), el uso de llaves (`{ y }`) para encerrar los componentes de la clase y las barras oblicuas para definir comentarios de una sola línea. En caso de querer escribir comentarios más extensos podemos encerrar el bloque entre `/* y */` de la siguiente forma:

```
/* Este es un comentario más extenso que ocupa más de una línea:
Al utilizar los símbolos barra oblicua y asterisco, indicamos que lo que sigue a continuación
es comentario del código. Los comentarios de código se utilizar frecuentemente en el ámbito de
la programación ya que es necesario aclarar distintos aspectos de la codificación, como, por
ejemplo: cuáles son los objetivos de los métodos, qué parámetros espera un método, explicar
una sentencia en particular, etc. Los comentarios son muy útiles para explicar utilizando el
lenguaje natural qué es lo que se está programando y de esta manera, poder en un futuro leer
rápidamente los comentarios y entender y recordar cuál era el objetivo del código. Además, en
esta disciplina realizar comentarios durante el proceso de programación es una muy buena
práctica que se utiliza frecuentemente para que en el caso de que otro programador tenga que
continuar con el código pueda entender cuál fue el pensamiento del que creó originariamente el
mismo.*/
```

Los comentarios enmarcados entre /* y */ no se pueden anidar. Los comentarios tanto sean de una línea o de bloques serán ignorados por el compilador a la hora de generar el **bytecode** a ser ejecutado en la máquina virtual, sólo sirven a los fines de lectura y comprensión del código por parte del equipo desarrollador. Más adelante veremos una forma especial de escritura de comentarios denominada **JavaDoc** que nos permitirá generar la documentación asociada al programa con las definiciones de las clases, métodos y demás de forma automatizada. Sintácticamente los comentarios JavaDoc comienzan con barra y dos asteriscos y terminan con asterisco barra:

```
/**  
...  
*/
```

Tipos de Datos

Los tipos de variables disponibles son básicamente 3:

- Tipos básicos (no son objetos)
- Arreglos (arrays o vectores)
- Clases e Interfaces

Con lo que vemos que cada vez que creamos una clase o interface estamos definiendo un nuevo tipo. Siempre es aconsejable asignar un valor por defecto en el momento de declaración de una variable. En algunos casos, incluso, se producirá un error durante la compilación si hemos olvidado inicializar el valor de alguna variable y tratamos de manipular su contenido.

Tipos básicos de datos

Nombre	Declaración	Rango	Descripción
Booleano	boolean	true - false	Define una bandera que puede tomar dos posibles valores: true o false.
Byte	byte	[-128 .. 127]	Representación del número de menor rango con signo.
Entero pequeño	short	[-32,768 .. 32,767]	Representación de un entero cuyo rango es pequeño.
Entero	int	[-2 ³¹ .. 2 ³¹ -1]	Representación de un entero estándar. Este tipo puede representarse sin signo usando su clase Integer a partir de la Java SE 8.
Entero largo	long	[-2 ⁶³ .. 2 ⁶³ -1]	Representación de un entero de rango ampliado. Este tipo puede representarse sin signo usando su clase Long a partir de la Java SE 8.
Real	float	[±3,4·10 ⁻³⁸ .. ±3,4·10 ³⁸]	Representación de un real estándar. Recordar que al ser real, la precisión del dato contenido varía en función del tamaño del número: la precisión se amplía con números más próximos a 0 y disminuye cuanto más se aleja del mismo.

Real largo	double	$[\pm 1,7 \cdot 10^{-308} .. \pm 1,7 \cdot 10^{308}]$	Representación de un real de mayor precisión. Double tiene el mismo efecto con la precisión que float.
Carácter	char	<code>['\u0000' .. '\uffff'] o [0 .. 65.535]</code>	Carácter o símbolo. Para componer una cadena es preciso usar la clase String, no se puede hacer como tipo primitivo.

Tabla 9 – Tipos de básicos de datos en JAVA

Tipos de datos referencia

En Java los objetos, instancias de clases, se manejan a través de referencias. Cuando se crea una nueva instancia de una clase con el operador *new*, este devuelve una referencia al tipo de la clase. Para aclararlo veamos un ejemplo:

Si tenemos la clase Punto definida de la siguiente manera:

```
public class Punto {
    private float float x;
    private float y;
    ...
    public void setX(float x) {
        this.x = x;
    }
    ...
}
Punto unPunto = new Punto();
```

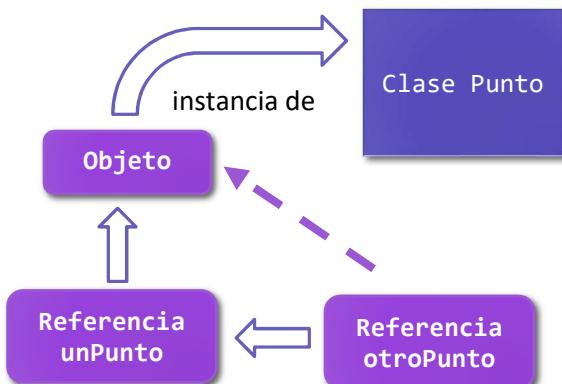
El operador *new* reserva espacio en memoria para contener un objeto del tipo Punto y devuelve una referencia que se asigna a *unPunto*, con los valores de atributos que se definieron en el constructor. A partir de aquí, accedemos al objeto a través de su referencia. Es posible, por tanto, tener varias referencias al mismo objeto. Presta atención al siguiente fragmento de código.

```
Punto unPunto = new Punto();
unPunto.print();

Punto otroPunto = unPunto;
otroPunto.setX(1.0f);
otroPunto.setY(2.0f);
otroPunto.print();
```

La salida por pantalla es:

Coordenadas del punto (0.0f,0.0f)
Coordenadas del punto (1.0f,2.0f)



Como las dos referencias: *unPunto* y *otroPunto*, hacen referencia a la misma instancia, los cambios sobre el objeto se pueden realizar a través de cualquiera de ellas.

Conversión de tipos de datos o Casting

En muchas ocasiones hay que transformar una variable de un tipo a otro, por ejemplo, de *int* a *double*, o de *float* a *long*. En otras ocasiones la conversión debe hacerse entre objetos de clases diferentes, aunque relacionadas mediante la herencia de una clase común o la implementación de una interfaz compatible. Veremos más sobre este último caso cuando hablaremos de Polimorfismo.

La conversión entre tipos primitivos es más sencilla. En Java, se realizan de modo automático conversiones implícitas *de un tipo a otro de más precisión*, por ejemplo, de *int* a *long*, de *float* a *double*, etc. Estas conversiones se hacen al mezclar variables de distintos tipos en expresiones matemáticas o al ejecutar sentencias de asignación en las que el miembro izquierdo tiene un tipo distinto que el resultado de evaluar el miembro derecho.

Por ejemplo:

```
int a = 5;
float b = 1.5;
float c = a * b; // producto de distintos tipos de datos primitivos convertidos
automáticamente
```

Las conversiones de un tipo de mayor a otro de menor precisión requieren una orden explícita del programador, pues son *conversiones inseguras* que pueden dar lugar a errores (por ejemplo, para pasar a *short* un número almacenado como *int*, hay que estar seguro de que puede ser representado con el número de cifras binarias de *short*). A estas conversiones **explícitas** de tipo se les llama *cast*. El *cast* se hace poniendo el tipo al que se desea transformar entre paréntesis, como, por ejemplo,

```
long result;
result = (long) (a/(b+c));
```

El Recolector de basura

Los objetos que dejan de estar referenciados a través de alguna variable no se pueden volver a recuperar. Para que estos objetos “desreferenciados” no ocupen memoria, un recolector de basura se encarga de «destruirlos» y liberar la memoria que estaban ocupando. Por lo tanto, para «destruir» un objeto basta con asignar a su variable de referencia el valor null como se puede ver en el siguiente ejemplo.

```
Punto unPunto = new Punto(1.0f, 2.0f);
Punto otroPunto = new Punto(1.0f, -1.0f);
unPunto = new Punto(2.0, 2.0f); // El punto (1.0f, 2.0f) se pierde
otroPunto = null; // El punto (1.0f, -1.0f) se pierde
```

Ámbito de las variables

Toda variable tiene un ámbito. Esto es la parte del código en la que una variable se puede utilizar. De hecho, las variables tienen un ciclo de vida:

1. En la declaración se reserva el espacio necesario para que se puedan comenzar a utilizar (digamos que se avisa de su futura existencia).
2. Se la asigna su primer valor (la variable nace).

3. Se la utiliza en diversas sentencias.
4. Cuando finaliza el bloque en el que fue declarada, la variable muere. Es decir, se libera el espacio que ocupa esa variable en memoria. No se la podrá volver a utilizar.

Una vez que la variable ha sido eliminada, no se puede utilizar. Dicho de otro modo, no se puede utilizar una variable más allá del bloque en el que ha sido definida.

El ámbito de las variables está determinado por el bloque de código donde se declaran y todos los bloques que estén anidados por debajo de este. Presta atención al siguiente fragmento de código:

```
{
    // Aquí tengo el bloque externo
    int entero = 1;
    Punto unPunto = new Punto();
    {
        // Y aquí tengo el bloque interno
        int entero = 2; // Error ya está declarada
        unPunto = new Punto(1.0f, 1.0f); // Correcto
    }
}
```

Operadores

Las variables se manipulan muchas veces utilizando operaciones con ellos. Los datos se suman, se restan, multiplican, etc. y a veces se realizan operaciones más complejas.

Operadores aritméticos

En Java disponemos de los operadores aritméticos habituales en lenguajes de programación como son suma, resta, multiplicación, división y operador que devuelve el resto de una división entre enteros, también llamado módulo:

OPERADOR	DESCRIPCIÓN
+	Suma
-	Resta
*	Multiplicación
/	División
%	Resto de una división entre enteros (en otros lenguajes denominado mod)

Tabla 10 – Tipos de operadores aritméticos en JAVA

Cabe destacar que el operador % es de uso exclusivo entre enteros. $7 \% 3$ devuelve 1 ya que el resto de dividir 7 entre 3 es 1. Al valor obtenido lo denominamos módulo (en otros lenguajes en vez del símbolo % se usa la palabra clave mod) y a este operador a veces se le denomina “operador módulo”.

Las operaciones con operadores siguen un **orden de prelación o de precedencia** que determinan el orden con el que se ejecutan. Si existen expresiones con varios operadores del mismo nivel, la operación se ejecuta de izquierda a derecha. Para evitar resultados no deseados, en casos donde pueda existir duda se recomienda el uso de paréntesis para dejar claro con qué orden deben ejecutarse las operaciones. Por ejemplo, si dudas si la expresión:

$3 * 2 / 7 + 2$

se ejecutará en el orden que esperas, es posible agrupar términos utilizando paréntesis. Por ejemplo:

$3 * ((a / 7) + 2)$

Se podrían mostrar también los resultados de las dos expresiones, para ver las diferencias.

Operadores lógicos

En Java disponemos de los operadores lógicos habituales en lenguajes de programación como son “es igual”, “es distinto”, menor, menor o igual, mayor, mayor o igual, and (y), or (o) y not (no). La sintaxis se basa en símbolos como veremos a continuación y cabe destacar que hay que prestar atención a no confundir == con = porque implican distintas cosas.

OPERADOR	DESCRIPCIÓN
<code>==</code>	Es igual
<code>!=</code>	Es distinto
<code><, <=, >, >=</code>	Menor, menor o igual, mayor, mayor o igual
<code>&&</code>	Operador and (y)
<code> </code>	Operador or (o)
<code>!</code>	Operador not (no)

Tabla 11 – Tipos de operadores lógicos en JAVA

Los conectivos lógicos (AND, OR y NOT), sirven para evaluar condiciones complejas. NOT sirve para negar una condición. Ejemplo:

```
boolean mayorDeEdad, menorDeEdad;
int edad = 21;
mayorDeEdad = edad >= 18; //mayorDeEdad será true menorDeEdad = !mayorDeEdad;
//menorDeEdad será false
```

El operador **&&** (AND) sirve para evaluar dos expresiones de modo que, si ambas son ciertas, el resultado será true sino el resultado será false. Ejemplo:

```
boolean carnetConducir=true;
int edad=20;
boolean puedeConducir= (edad>=18) && carnetConducir; //Si la edad es de al
menos 18 años y carnetConducir es //true, puedeConducir es true
```

El operador **||** (OR) sirve también para evaluar dos expresiones. El resultado será true si al menos una de las expresiones es true. Ejemplo:

```
boolean nieva =true, llueve=false, graniza=false;
malTiempo= nieva || llueve || graniza;
```

Los operadores **&&** y **||** se llaman operadores en cortocircuito porque si no se cumple la condición de un término no se evalúa el resto de la operación. Por ejemplo:

$$(a == b \&\& c != d \&\& h >= k)$$

tiene tres evaluaciones: la primera comprueba si la variable a es igual a b. Si no se cumple esta condición, el resultado de la expresión es falso y no se evalúan las otras dos condiciones posteriores.

En un caso como:

$$(a < b \mid\mid c != d \mid\mid h <= k)$$

se evalúa si a es menor que b. Si se cumple esta condición el resultado de la expresión es verdadero y no se evalúan las otras dos condiciones posteriores.

Operadores de asignación

Permiten asignar valores a una variable. El fundamental es “**=**”. Pero sin embargo se pueden usar expresiones más complejas como:

```
x = 5;
x += 3; // x vale 8 ahora
```

En el ejemplo anterior lo que se hace es sumar 3 a la x (es lo mismo **x+=3**, que **x=x+3**).

Eso se puede hacer también con todos estos operadores:

Nombre	Operador	Ejemplo	Equivalencia
Operadores aritméticos			
Suma y asignación	$+=$	$a += b$	$a = a+b$
Resta y asignación	$-=$	$a -= b$	$a = a-b$
Multiplicación y asignación	$*=$	$a *= b$	$a = a*b$
División y asignación	$/=$	$a /= b$	$a = a/b$
Resto de la división y asignación	$%=$	$a %= b$	$a = a \% b$
Operadores a nivel de bits			
AND binario y asignación	$\&=$	$a \&= b$	$a = a \& b$
OR binario y asignación	$ =$	$a = b$	$a = a b$
XOR binario y asignación	$^=$	$a ^= b$	$a = a ^ b$
Desplazamiento de bits hacia la izquierda en b posiciones y asignación	$<<=$	$a <<= b$	$a = a << b$
Desplazamiento de bits hacia la derecha en b posiciones y asignación	$>>=$	$a >>= b$	$a = a >> b$

Tabla 12 – Operadores de Asignación en JAVA

También se pueden concatenar asignaciones:

```
x1 = x2 = x3 = 5; // todas valen 5
```

Otros operadores de asignación son “**++**” (incremento) y “**--**”(decremento), que incrementan o decrementan en una unidad el valor de la variable. Son muy utilizados en algoritmos de recorrido o dentro de estructuras de control como el ciclo for que vimos en el módulo de Técnicas de Programación.

Pero hay dos formas de utilizar el incremento y el decremento. Se puede usar por ejemplo **x++** o **++x**, la diferencia estriba en el modo en el que se comporta la asignación en cuanto al orden en que es evaluada.

Ejemplo:

```
int x=5, y=5, z;
z=x++; // z vale 5, x vale 6
z=++y; // z vale 6, y vale 6
```

Operador ternario

Este operador (conocido como if de una línea) permite devolver un valor u otro según el valor de la expresión analizada. Su sintaxis es la siguiente:

Expresionlogica ? valorSiVerdadero: valorSiFalso;

Como, por ejemplo:

```
float precioDeLista = aplicaDescuento == true ? 123 : 234;
```

En el caso de que la variable `aplicaDescuento` tenga valor verdadero, el precio de lista del producto que estamos analizando será menor; es importante destacar que la asignación anterior es equivalente a utilizar condicionales como veremos en las siguientes secciones.

Arrays

En Java los arrays son objetos, instancias de la clase `Array`, la cual dispone de ciertos métodos útiles. La declaración sigue la siguiente sintaxis: se debe declarar el tipo base de los elementos del array. El tipo base puede ser un tipo primitivo o un tipo de referencia:

```
int arrayDeEnteros[] = null; // Declara un array de enteros
Punto arrayDePuntos[] = null; /* Declara un array de referencias a Puntos */
```

La creación del array se hace, como con cualquier otro objeto, mediante el uso del operador `new()`:

```
arrayDeEnteros = new int[100]; /* Crea el array con espacio para 100 enteros */
arrayDePuntos = new Punto[100]; /* Crea el array con espacio para 100 referencias
a Punto */
```

En el primer caso se reserva espacio para contener 100 enteros. En el segundo caso se crea espacio para contener 100 referencias a objetos de la clase `Punto`, pero no se crea cada uno de esos 100 objetos. En el siguiente ejemplo se muestra como se crea cada uno de esos 100 objetos de la clase `Punto` y se asignan a las referencias del array.

```
for(int i = 0; i < 100; i++)
    arrayDePuntos[i] = new Punto();
```

Los arrays se pueden iniciar en el momento de la creación, como en el siguiente ejemplo:

```
int arrayDeEnteros[] = {1, 2, 3, 4, 5};
Punto arrayDePuntos[] = {new Punto(), new Punto(1.0f, 1.0f)};
```

Los arrays disponen de un atributo llamado `length`, que significa longitud, indica el número de elementos que contiene, al que se puede acceder como sigue:

```
int arrayDeEnteros[] = {1, 2, 3, 4, 5};
int tamaño = arrayDeEnteros.length; // el valor es 5
```

Cadenas de caracteres

En Java existe una clase para representar y manipular cadenas, la clase `String`. Una vez creado un `String` no se puede modificar. Se pueden crear instancias de una manera abreviada y sobre ellas se puede utilizar el operador de concatenación `+`:

```
String frase = "Esta cadena es una frase "
String larga = frase + "que se puede convertir en una frase larga."
```

Constantes

Una constante es una “variable” de solo lectura. Dicho de otro modo, más correcto, es un valor que no puede variar (por lo tanto, no es una variable en sí).

La forma de declarar constantes es la misma que la de crear variables, sólo que hay que anteponer la palabra final que es la que indica que estamos declarando una constante y por tanto no podremos variar su valor inicial durante la ejecución del programa:

```
final double PI = 3.141591;
```

Es una buena práctica comúnmente aceptada el asignar nombres en mayúscula sostenida para las constantes a fin de diferenciarlas luego en el código del programa.

Estructuras de control

Estructuras condicionales

Dos son las estructuras de control condicionales en Java: bifurcación y selección múltiple.

Bifurcación: if-else, if-else-if

Su sintaxis es:

```
if (condicion)
{
    instruccion1();
    instruccion2();
    // etc
} else
{
    instruccion1();
    instruccion2();
    // etc
}
```

Es necesario que la condición sea una variable o expresión booleana. Si sólo existe una instrucción en el bloque, las llaves no son necesarias. No es necesario que exista un bloque else, como en el siguiente ejemplo:

```
if (condicion)
{
    bloqueDeInstrucciones();
}

// continua la ejecución del programa
```

```

    }
else
{
    if(condicion2)
    {
        bloqueDeInstrucciones();
    }
    else
    {
        bloqueDeInstrucciones();
    }
}

```

Cada cláusula `else` corresponde al último `if` inmediato anterior que se haya ejecutado, es por eso que debemos tener especial consideración de encerrar correctamente entre llaves los bloques para determinar exactamente a qué cláusula corresponde. En este caso, es de especial utilidad indentar nuestro código utilizando espacios o tabulaciones como se muestra en el ejemplo anterior."

Un ejemplo del uso para un caso en particular es el siguiente:

```

final int totalMaterias = 4;
int materiasAprobadas = 2;

if (materiasAprobadas == totalMaterias)
{
    otorgarCertificado();
}
else
{
    continuarCapacitacion();
}

```

Selección múltiple: switch

Su sintaxis es la siguiente:

```

switch (expresion)
{
    case valor1:
        instrucciones();
        break;
    case valor2:
        instrucciones();
        break;
    default:
        instrucciones();
}

```

La expresión ha de ser una variable de tipo `int` o una expresión que devuelva un valor entero. Cuando se encuentra coincidencia con un `case` se ejecutan las instrucciones a él asociadas hasta encontrar el primer `break`. Si no se encuentra ninguna coincidencia se ejecutan las instrucciones del bloque `default`, la cual es opcional. El `break` no es exigido en la sintaxis y entonces si no se pone, el código se ejecuta atravesando todos los cases hasta que encuentra uno.

Bucles

Un bucle se utiliza para realizar un proceso repetidas veces. Se denomina también lazo o loop. El código incluido entre las llaves {} (opcionales si el proceso repetitivo consta de una sola línea), se ejecutará mientras se cumpla unas determinadas condiciones. Hay que prestar especial atención a los bucles infinitos, hecho que ocurre cuando la condición de finalizar el bucle (expresión booleana) no se llega a cumplir nunca.

Bucle While

Las sentencias encerradas por llaves se ejecutan mientras **condición** tenga un valor verdadero:

```
while (condicion)
{
    accion();
}
```

Bucle For

La forma general del bucle for es la siguiente:

```
for (inicio; condicion; incremento)
{
    accion();
}
```

que es equivalente a utilizar while en la siguiente forma,

```
inicio();
while (condicion)
{
    accion();
    incremento();
}
```

Bucle Do While

Es similar al bucle while pero con la particularidad de que el control está al final del bucle lo que hace que el cuerpo del bucle se ejecute al menos una vez, independientemente de que la condición se cumpla o no. Una vez ejecutados el cuerpo, se evalúa la condición: si resulta true se vuelven a ejecutar las sentencias incluidas en el bucle, mientras que si la condición se evalúa a false finaliza el bucle.

```
do
{
    acciones();
}
while (condicion);
```

Sentencias break y continue

La sentencia **break** es válida tanto para las bifurcaciones como para los bucles. Hace que se salga inmediatamente del bucle o bloque que se está ejecutando sin finalizar el resto de las sentencias.

La sentencia **continue** se utiliza en los bucles (no en bifurcaciones). Finaliza la iteración que en ese momento se está ejecutando (no ejecuta el resto de sentencias que hubiera hasta el final del cuerpo del bucle). Vuelve al comienzo del bucle y comienza la siguiente iteración si existiera.

Bloques para manejo de excepciones

Java incorpora en el propio lenguaje la gestión de errores. El mejor momento para detectar los errores es durante la compilación. Sin embargo, prácticamente solo los errores de sintaxis son detectados en esta operación. El resto de problemas surgen durante la ejecución de los programas.

En el lenguaje *Java*, una *Exception* es una clase que representa un tipo de error o una condición anormal que se ha producido durante la ejecución de un programa. Algunas *excepciones* son *fatales* y provocan que se deba finalizar la ejecución del programa. En este caso conviene terminar ordenadamente y dar un mensaje explicando el tipo de error que se ha producido. Otras *excepciones*, como por ejemplo no encontrar un archivo en el que hay que leer o escribir algo, pueden ser *recuperables*. En este caso el programa debe dar al usuario la oportunidad de corregir el error (dando por ejemplo la opción de seleccionar un nuevo archivo).

Existen algunos tipos de excepciones que *Java* obliga a tener en cuenta. Esto se hace mediante el uso de bloques *try*, *catch* y *finally*.

```
try {
    // código que puede arrojar una excepción
}
catch (Exception ex) { // atrapamos el error producido
    // lo procesamos
}
finally {
    // y finalmente lo post procesamos
}
```

El código dentro del bloque *try* está *vigilado*: si se produce una situación anormal y se lanza como consecuencia una excepción, el control pasa al bloque *catch* que se hace cargo de la situación y decide lo que hay que hacer. Se pueden incluir tantos bloques *catch* como se desee, cada uno de los cuales tratará un tipo de excepción. Finalmente, si está presente, se ejecuta el bloque *finally*, que es opcional, pero que en caso de existir se ejecuta siempre, sea cual sea el tipo de error. Es importante destacar que el parámetro del bloque *catch* es un objeto de una clase que hereda de *Exception* (veremos esto más adelante), pero por ahora debemos saber que este objeto puede contener información importante acerca de la causa del error.

En el caso en que el código de un método pueda generar una *Exception* y no se desee incluir en dicho método la gestión del error (es decir los bloques *try/catch* correspondientes), es necesario que el método pase la *Exception* al método desde el que ha sido llamado. Esto se consigue mediante la adición de la palabra *throws* seguida del nombre de la *Exception* concreta, después de la lista de argumentos del método. A su vez el método superior deberá incluir los bloques *try/catch* o volver a pasar la *Exception*. De esta forma se puede ir pasando la *Exception* de un método a otro hasta llegar al último método del programa, el método *main()*.

En el peor de los casos que nadie se haga cargo de la excepción ésta llegará al usuario en forma de un mensaje de “Excepción no capturada” lo cual no es una buena práctica.

Como veremos más adelante para algunos tipos de excepciones es obligatorio el uso de los bloques try-catch para manejar los posibles resultados, o en el último de los casos, el método de ejecución main(String[] args) deberá arrojarla explícitamente usando la palabra reservada throws y llegará al usuario final, lo cual no es en absoluto recomendable. En cambio, algunas excepciones propias del lenguaje Java heredan de la clase RuntimeException y se utilizan para tratar errores que pueden ocurrir al momento de ejecutar el programa posiblemente derivados del manejo de datos que no pudieron ser previstos en el momento de la programación. Estas excepciones no necesitan ser capturadas en un bloque try-catch de forma explícita.

Clases y Objetos

Como vimos anteriormente, las *clases* son el centro del paradigma de *Programación Orientada a Objetos(POO)*. Algunos conceptos importantes de la POO son los siguientes:

1. **Encapsulamiento:** Las clases pueden ser declaradas como públicas (*public*) y como paquete (*package*) (accesibles sólo para otras clases del mismo paquete). Las variables miembros y los métodos pueden ser *public*, *private*, *protected* y *package*. De esta forma se puede controlar el acceso entre objetos y evitar un uso inadecuado.
2. **Herencia:** Una clase puede衍生 de otra (*extends*), y en ese caso hereda todas sus variables y métodos. Una clase derivada puede *añadir* nuevas variables y métodos y/o *redefinir* las variables y métodos heredados.
3. **Polimorfismo:** Los objetos de distintas clases pertenecientes a una misma jerarquía o que implementan una misma interface, pueden responder de forma indistinta a un mismo método. Esto, como se ha visto anteriormente, facilita la programación y el mantenimiento del código.

A continuación, veremos cómo se declaran tanto clases como interfaces, y cuál es el proceso para crear sus instancias.

Intuitivamente, una clase es una agrupación de *datos* (variables o campos) y de *funciones* (métodos) que operan sobre esos datos. Todos los métodos y variables deben ser definidos dentro del *bloque* {...} de la clase.

Un *objeto* (en inglés *instance*) es un ejemplar concreto de una clase. Las *clases* son como tipos de variables, mientras que los *objetos* son como variables concretas de un tipo determinado.

```
NombreDeLaClase unObjeto;  
NombreDeLaClase otroObjeto;
```

A continuación, se enumeran algunas características importantes de las clases:

1. Todas las variables y métodos de *Java* deben pertenecer a una clase. No hay variables y funciones globales.
2. Si una clase deriva de otra (*extends*), hereda todas sus variables y métodos.
3. *Java* tiene una jerarquía de clases estándar de la que pueden derivar las clases que crean los usuarios. Es decir que toda clase definida por el programador es heredada de la clase Object definida por el lenguaje de programación.

4. Una clase sólo puede heredar de una única clase (en *Java* no hay herencia múltiple). Si al definir una clase no se especifica de qué clase deriva, por defecto la clase deriva de *Object*. La clase *Object* es la base de toda la jerarquía de clases de *Java*.
5. En un archivo de código fuente se pueden definir varias clases, pero en un mismo archivo no puede haber más que una clase definida como *public*. Este archivo se debe llamar como la clase *public* que debe tener extensión *.java*. Con algunas excepciones, lo habitual es escribir una sola clase por archivo.
6. Si una clase contenida en un fichero no es *public*, no es necesario que el fichero se llame como la clase.
7. Los métodos y variables de una clase pueden referirse de modo global a un *objeto* de esa clase a la que se aplican por medio de la referencia *this*. Al utilizar la palabra reservada 'this' para referirse tanto a métodos como atributos se restringe el ámbito al objeto que hace la declaración.
8. Las clases se pueden agrupar en *packages que significa paquetes*, introduciendo una línea al comienzo del fichero (*package packageName;*). Esta agrupación en *packages* está relacionada con la jerarquía de carpetas y archivos en la que se guardan las clases.
En la práctica usamos paquetes para agrupar clases con un mismo propósito usando jerarquía de paquetes; esta decisión es muy importante a la hora de diseñar la estructura de nuestro programa.

Variables miembros de objeto

La programación orientada a objetos está *centrada en los datos*. Una clase son *datos y métodos* que operan sobre esos datos.

Cada objeto, es decir cada instancia concreta de una clase, tiene su propia copia de las variables miembro. Las variables miembros de una clase (también llamadas *atributos*) pueden ser de *tipos primitivos* (*boolean*, *int*, *long*, *double*, ...) o referencias a *objetos* de otra clase (*agregación* y *composición*).

Un aspecto muy importante para el correcto funcionamiento de los programas es que no haya datos sin inicializar. Por eso las variables miembros de *tipos primitivos* se inicializan siempre de modo automático, incluso antes de llamar al *constructor* (*false* para *boolean*, el carácter nulo para *char* (código Unicode '\u0000') y cero para los tipos numéricos). De todas formas, lo más adecuado es inicializarlas en el constructor.

También pueden inicializarse explícitamente en la *declaración*, como las variables locales, por medio de constantes o llamadas a métodos. Por ejemplo,

```
public class Alumno
{
    int edad = 18;
}
```

Las variables miembros se inicializan en el mismo orden en que aparecen en el código de la clase. Esto es importante porque unas variables pueden apoyarse en otras previamente definidas.

Cada *objeto* que se crea de una clase tiene *su propia copia* de las variables miembro. Por ejemplo, cada objeto de la clase *Circulo* tiene sus propias coordenadas del centro *x* e *y*, y su propio valor del radio *r*. Se puede aplicar un método a un objeto concreto poniendo el nombre del objeto y luego el nombre del

método separados por un punto. Por ejemplo, para calcular el área de un objeto de la clase *Circulo* llamado *c1* se escribe: *c1.area()*;

La definición de cada atributo debe empezar con un modificador de acceso. Los modificadores de acceso indican la visibilidad, es decir, si se puede tener acceso sólo desde la clase (*private*), desde la clase y las clases que heredan de ella (*protected*) y desde cualquier clase definida en el mismo paquete o desde cualquier clase (*public*).

Tras el modificador de acceso se escribe el tipo del argumento, este puede ser un tipo primitivo a de tipo referencia. Tras el modificador de acceso, un atributo se puede declarar como *static*. Esto implica que no existe una copia de este atributo en cada instancia de la clase, si no que existe uno único común a todas las instancias. A los atributos *static* también se les llama atributos de clase.

Otro modificador que puede afectar al comportamiento de los atributos de una clase es *final*. Si un atributo se declara como *final*, implica que no se puede cambiar su valor una vez definido. Un ejemplo de uso de este modificador son las constantes de clase:

```
public static final float E = 2.8182f;
```

Una clase puede tener variables propias de la clase y no de cada objeto. A estas variables se les llama *variables de clase* o *variables static*. Las variables *static* se suelen utilizar para definir constantes comunes para todos los objetos de la clase (por ejemplo, *PI* en la clase *Circulo*) o variables que sólo tienen sentido para toda la clase (por ejemplo, un contador de objetos creados como *numCirculos* en la clase *Circulo*).

Las variables de clase se crean anteponiendo la palabra *static* a su declaración como en el ejemplo anterior. Para llamarlas se suele utilizar el nombre de la clase (no es imprescindible, pues se puede utilizar también el nombre de cualquier objeto), porque de esta forma su sentido queda más claro. Por ejemplo, *Circulo.numCirculos* es una variable de clase que cuenta el número de círculos creados.

Si no se les da valor en la declaración, las variables miembros *static* se inicializan con los valores por defecto (*false* para *boolean*, el carácter nulo para *char* (código Unicode '\u0000') y cero para los tipos numéricos) para los tipos primitivos, y con *null* si es una referencia.

Las variables miembros *static* se crean en el momento en que pueden ser necesarias: cuando se va a crear el primer objeto de la clase, en cuanto se llama a un método *static* o en cuanto se utiliza una variable *static* de dicha clase. Lo importante es que las variables miembros *static* se inicializan siempre antes que cualquier objeto de la clase.

Métodos

Los métodos especifican el comportamiento de la clase y sus instancias. Los modificadores de acceso y su significado son los mismos que al operar sobre atributos. En particular, al declarar un método estático implica que es un método de la clase, y por lo tanto no es necesario crear ninguna instancia de la clase para poder llamarlo. El conjunto de los métodos públicos de una clase forma su interface.

Un método declarado *final* implica que no se puede redefinir en ninguna clase que herede de esta, veremos el tema de Herencia más adelante, pero es importante destacar que el cuerpo de un método definido como *final* no podrá ser modificado por ninguna clase hijo.

En el momento de la declaración hay que indicar cuál es el tipo del parámetro que devolverá el método o *void* en caso de que no devuelva nada. En otros lenguajes, estos tipos de métodos o funciones se denominan procedimientos.

Los métodos tienen *visibilidad directa* de las variables miembro del objeto que es su *argumento implícito*; es decir, pueden acceder a ellas sin cualificarlas con un nombre de objeto y el operador punto (.). De todas formas, también se puede acceder a ellas mediante la referencia *this*, de modo discrecional (como en el ejemplo anterior con *this.r*) o si alguna variable local o argumento las oculta.

Los métodos pueden definir *variables locales*. Su visibilidad llega desde la definición al final del bloque en el que han sido definidas. No hace falta inicializar las variables locales en el punto en que se definen, pero el compilador no permite utilizarlas sin haberles dado un valor. A diferencia de las variables miembro, las variables locales no se inicializan por defecto.

También se ha de especificar el tipo y nombre de cada uno de los argumentos del método entre paréntesis. Si un método no tiene argumentos el paréntesis queda vacío, no es necesario escribir void. El tipo y número de estos argumentos identifican al método, ya que varios métodos pueden tener el mismo nombre, con independencia del tipo devuelto, y se distinguirán entre sí por el número y tipo de sus argumentos, como veremos a continuación.

Ejemplo:

```
public class Alumno
{
    private String nombre;
    private String apellido;

    public String getNombreCompleto ()
    {
        return nombre + " " + this.apellido;
    }
}
```

En el ejemplo anterior hemos definido la clase Alumno con dos atributos de tipo cadena de texto: nombre y apellido. Es importante destacar que estos atributos son definidos como privados para poder respetar el principio de encapsulamiento del paradigma orientado a objetos y lo tomaremos como una buena práctica: los atributos de un objeto deberían ser accesibles sólo a través de métodos públicos. Además hemos definido el método público *getNombreCompleto()* que no recibe parámetros y devuelve una cadena de texto formada por el nombre y el apellido del alumno separados por un espacio. Es importante destacar que para acceder a los atributos de la instancia de esta clase puede usarse o no la palabra reservada *this* ya que en el cuerpo del método no existe otra variable local con el mismo nombre que pueda resultar en ambigüedad.

Sobre el paso de parámetros a un método

En Java los argumentos de los *tipos primitivos* se pasan siempre *por valor*. El método recibe una copia del argumento actual; si se modifica esta copia, el argumento original que se incluyó en la llamada no queda modificado. Para modificar un argumento de un tipo primitivo dentro del cuerpo del método puede incluirse como variable miembro o ser retornado para luego realizar la asignación en el momento de la llamada. Las *referencias* se pasan también *por valor*, pero a través de ellas se pueden modificar los objetos referenciados.

Sobre los métodos de seteo (get y set)

En el ejemplo anterior hemos visto que el prefijo del nombre es get, y en la práctica veremos que es de uso común nombrar métodos de obtención y modificación con los prefijos get y set respectivamente. Para poder respetar la buena práctica antes mencionada relacionada con el encapsulamiento de la orientación a objetos cada atributo de la clase debería ser definido como privado y existir métodos get y set para poder obtener y modificar sus valores.

Así, la definición de un método get para obtener el valor del atributo de un objeto tendría la siguiente forma:

```
public TipoDeDatos getAtributo ()
{
    return this.atributo;
}
```

Donde TipoDeDatos puede ser tanto un tipo primitivo como una clase o interfaz si estamos referenciando a otros objetos. Otro punto importante es que por convención el nombre del método luego del prefijo get o set normalmente se escribe en CamelCase como mencionamos en el módulo anterior a la hora de nombrar variables.

De esta forma, los métodos set reciben como parámetro el nuevo valor y no retornan nada, por lo que el tipo especificado es void.

```
public void setAtributo (TipoDeDatos nuevoValor)
{
    this.atributo = nuevoValor;
}
```

Ejemplo:

```
public class Alumno
{
    private String nombre;
    private String apellido;

    public String getNombre ()
    {
        return this.nombre;
    }

    public void setNombre (String nombre)
    {
        this.nombre = nombre;
    }

    public String getApellido ()
    {
        return this.apellido;
    }

    public void setApellido (String apellido)
    {
        this.apellido = apellido;
    }
}
```

```

public String getNombreCompleto ()
{
    return this.nombre + " " + this.apellido;
}
}

```

Entonces ahora podríamos crear un par de instancias de alumnos y mostrar en pantalla sus nombres completos de la siguiente forma:

```

Alumno alumno1 = new Alumno();
alumno1.setNombre("Pablo");
alumno1.setApellido("Filippo");

Alumno alumno2 = new Alumno();
alumno2.setNombre("Florencia");
alumno2.setApellido("Venne");

System.out.println("Nombre del alumno 1: " + alumno1.getNombreCompleto());
System.out.println("Nombre del alumno 2: " + alumno2.getNombreCompleto());

```

En programación orientada a objetos las llamadas a los métodos se les llama paso de mensajes, llamar a un método es análogo a pasarle un mensaje.

El método main()

Existe un nombre de método que está reservado en Java y otros lenguajes: el método main. Este método es especial ya que es el que da lugar al inicio del programa y será llamado por la máquina virtual al momento de la ejecución.

Es importante tener claro que el método main() no es el elemento principal en el desarrollo del programa. El programa, de acuerdo con el paradigma de programación orientada a objetos, se desarrolla mediante la interacción entre objetos por lo que el objetivo de este método normalmente es iniciar el programa delegar el comportamiento a los distintos los objetos correspondientes.

Este método debe pertenecer a una clase pública y su definición es la siguiente:

```

public static void main(String[] args)
{
    // cuerpo de nuestro programa
}

```

Es estático ya que no depende de una instancia en particular de la clase en la que se declara y no tiene ningún valor de retorno. Podemos ver que recibe un array de parámetros de tipo String que representan los argumentos pasados a la hora de ejecutar el programa. En el caso de realizar la ejecución mediante la línea de comando, cada elemento del array será una cada cadena de texto luego de la llamada a nuestro programa, separadas por espacios, ejemplo:

>> java ejemplo1 paso parámetros a mi programa

En este caso, al ejecutar el programa llamada ejemplo1, el método main() recibirá el array args[] compuesto de la siguiente forma:

```
args[0] => "paso"
args[1] => "parámetros"
args[2] => "a"
args[3] => "mi"
args[4] => "programa"
```

Métodos sobrecargados y redefinición (overload y override)

Java permite métodos *sobre cargados (overloaded)*, es decir métodos distintos con *el mismo nombre* que se diferencian por el número y/o tipo de datos de los argumentos.

A la hora de llamar a un método sobre cargado, Java sigue unas reglas para determinar el método concreto que debe llamar:

1. Si existe el método cuyos argumentos se ajustan exactamente al tipo de los argumentos de la llamada (argumentos actuales), se llama ese método.
2. Si no existe un método que se ajuste exactamente, se intenta promover los argumentos actuales al tipo inmediatamente superior (por ejemplo *char* a *int*, *int* a *long*, *float* a *double*, etc.) y se llama el método correspondiente.
3. Si sólo existen métodos con argumentos de un tipo más amplio (por ejemplo, *long* en vez de *int*), el programador debe hacer un *cast* explícito en la llamada, responsabilizándose de esta manera de lo que pueda ocurrir.
4. El valor de retorno no influye en la elección del método sobre cargado. En realidad, es imposible saber desde el propio método lo que se va a hacer con él. No es posible crear dos métodos sobre cargados, es decir con el mismo nombre, que sólo difieran en el valor de retorno.

Diferente de la *sobre carga* de métodos es la *redefinición (override)*. Una clase puede *redefinir* un método heredado de una superclase. *Redefinir* un método es dar una nueva definición. En este caso el método debe tener exactamente los mismos argumentos en tipo y número que el método redefinido. Este tema se verá de nuevo al hablar de la *Herencia*.

Ejemplo:

```
public class Alumno
{
    private String nombre;
    private String apellido;

    // se omiten los métodos get y set por simplicidad

    public String getNombreCompleto (String titulo)
    {
        return titulo + " " + this.nombre + " " + this.apellido;
    }
    public String getNombreCompleto ()
    {
```

```

        return this.getNombreCompleto("Sr/a.");
    // usamos un título por defecto para cuando no se especifica por parámetros
}
}

```

Entonces para la ejecución del ejemplo anterior que coloca como prefijo del nombre del alumno el título en caso de tenerlo o "Sr/a." en su defecto tenemos lo siguiente:

```
System.out.println("Nombre del alumno 1: " + alumno1.getNombreCompleto());
// muestra en pantalla "Sr/a. Pablo Filippo"
```

```
System.out.println("Nombre del alumno 2: " +
alumno2.getNombreCompleto("Contadora."));
// muestra en pantalla "Contadora. Paula Filippi"
```

Constructores de Objetos

Los métodos que tienen el mismo nombre que la clase tienen un comportamiento especial, sirven para crear las instancias de la clase y se les denomina constructores. Un *constructor* es un operador que se llama automáticamente cada vez que se crea un objeto de una clase. La principal misión del *constructor* es reservar memoria e inicializar las variables miembros de la clase. A continuación, se listan los principales aspectos de los constructores en Java que es necesario entender:

- Cuando se llama al constructor de una clase para instanciarla y crear el objeto, se invoca al constructor.
- Los constructores no tienen valor de retorno (ni siquiera void) y su nombre es el mismo que el de la clase.
- Su argumento implícito es el objeto que se está creando.
- Si no se define explícitamente un constructor, Java lo hará por nosotros ya que siempre es necesario que exista. Se creará un constructor sin argumentos.
- Una clase puede tener *varios constructores*, que se diferencian por el tipo y número de sus argumentos (los constructores justamente son un ejemplo típico de métodos *sobrecargados* que vimos anteriormente).

Si es necesario que un constructor llame a otro constructor lo debe hacer antes que cualquier otra cosa. Se llama *constructor por defecto* al constructor que no tiene argumentos. El programador debe proporcionar en el código, valores iniciales adecuados para todas las variables miembro. En caso de que sólo definamos un constructor con parámetros el constructor por defecto no será creado por Java, por lo que deberemos definirlo explícitamente en caso de ser necesario.

Por ejemplo:

```
public class Alumno
{
    private String nombre;
    private String apellido;

    public Alumno ()
    {

    }

    public Alumno (String nombre, String apellido)
```

```

{
    this.nombre = nombre;
    this.apellido = apellido;
}

// se omiten los métodos get y set por simplicidad
}

```

De esta forma podemos crear los mismos alumnos anteriores de la forma:

```

Alumno alumno1 = new Alumno("Pablo", "Filippo");
Alumno alumno2 = new Alumno("Florencia", "Venne");

```

Al igual que los demás métodos de una clase, los *constructores* pueden tener también los modificadores de acceso *public*, *private*, *protected* y *package*. Si un *constructor* es *private*, ninguna otra clase puede crear un objeto de esa clase. En este caso, puede haber métodos *public* y *static* (*factory methods*) que llamen al *constructor* y devuelvan un objeto de esa clase.

Dentro de una clase, los *constructores* sólo pueden ser llamados por otros *constructores* o por métodos *static*. No pueden ser llamados por los *métodos de objeto* de la clase.

Finalización y Destrucción de Objetos

Como mencionamos anteriormente, en *Java* el sistema se ocupa automáticamente de liberar la memoria de los objetos que ya han *perdido la referencia*, esto es, objetos que ya no tienen ningún nombre que permita acceder a ellos, por ejemplo, por haber llegado al final del bloque en el que habían sido definidos (se acabó su *scope*), porque a la *referencia* se le ha asignado el valor *null* o porque a la *referencia* se le ha asignado la dirección de otro objeto. A esta característica de *Java* se le llama *garbage collection* (recogida de basura).

Antes de que un objeto sea completamente eliminado de la memoria por el *recolector de basura*, se llama a su método *finalize()*. Este método está definido en la clase *Object* de la que hereda implícitamente cualquier nueva clase.

Un *finalizador* es un método de objeto (no *static*), sin valor de retorno (*void*), sin argumentos y que siempre se llama *finalize()*. Los *finalizadores* se llaman de modo automático siempre que hayan sido definidos por el programador de la clase. Para realizar su tarea correctamente, un *finalizador* debería terminar siempre llamando al *finalizador* de su *super-clase*.

Agrupando Clases en Paquetes

Un *package* es una agrupación de clases que sirve para establecer una jerarquía lógica en la organización de las clases. Además, tiene una relación directa con la organización física de nuestro código ya que también se representa en la estructura de archivos y carpetas que conforman nuestro programa. Al estructurar de este modo las clases, estamos estableciendo un dominio de nombres que la máquina virtual de *Java* utiliza, por ejemplo, cuando nuestra aplicación utiliza clases distribuidas en una red de computadores.

Todas las clases dentro de un mismo paquete tienen acceso al resto de clases declaradas como públicas. Para poder acceder a una clase de otro paquete se ha de importar anteriormente mediante la sentencia *import*.

Para que una clase pase a formar parte de un *package* llamado *nombreDelPaquete*, hay que introducir en ella la sentencia:

```
package nombreDePaquete;
```

Debe ser la primera sentencia del archivo sin contar comentarios y líneas en blanco.

Los nombres de los *packages* se suelen escribir con minúsculas, para distinguirlos de las clases, que empiezan por mayúsculo. El nombre de un *package* puede constar de varios nombres unidos por puntos (los propios *packages* de Java siguen esta norma, como por ejemplo *java.awt.event*).

Todas las clases que forman parte de un *package* deben estar en la misma carpeta. Los nombres compuestos de los *packages* están relacionados con la jerarquía de carpetas en que se guardan las clases. Es recomendable que los *nombres de las clases* de Java sean únicos, es el nombre del *package* lo que permite obtener esta característica.

En un programa de Java, una clase puede ser referida con su nombre completo (el nombre del *package* más el de la clase, separados por un punto). También se pueden referir con el nombre completo las variables y los métodos de las clases. Esto se puede hacer siempre de modo opcional, pero es incómodo y hace más difícil el reutilizar el código y portarlo a otras máquinas.

La sentencia *import* permite abreviar los nombres de las clases, variables y métodos, evitando el tener que escribir continuamente el nombre del *package* importado. Se importan por defecto el *package* *java.lang* y el *package* actual o por defecto (las clases del directorio actual).

Existen dos formas de utilizar *import*: para *una clase* y para *todo un package*:

```
import poo.cine.Actor;
import poo.cine.*;
```

El importar un *package* no hace que se carguen todas las clases del *package*: sólo se cargarán las clases *public* que se vayan a utilizar. Al importar un *package* no se importan los *sub-packages*. Éstos deben ser importados explícitamente, pues en realidad son *packages* distintos. Por ejemplo, al importar *java.awt* no se importa *java.awt.event*.

Imaginemos la siguiente estructura de paquetes y clases:

```
futsal
- torneos
    -- Torneo.java
    -- Fecha.java
    -- Encuentro.java
- equipos
    -- Equipo.java
    -- Jugador.java
- predio
    -- Cancha.java
```

Podríamos definir la clase Equipo de la siguiente forma:

```

package futsal.equipos;

public class Equipo
{
    private String nombre;
    private Jugador[] jugadores;
    // Podríamos seguir listando atributos, pero por simplificación dejamos los establecidos anteriormente.
}

```

Es importante destacar la primera línea "package" en el ejemplo anterior, que es la que indica cuál es el paquete en el que está ubicada la clase Equipo. Además, vemos que nuestra clase tiene relación con la clase Jugador, que al estar en el mismo paquete no debe ser importada ya que comparte la visibilidad. Otro dato importante es que la definición de visibilidad de la clase Equipo es public, lo que nos permitirá utilizarla en otros paquetes.

Por otro lado, podríamos escribir la clase Encuentro entre 2 equipos como sigue:

```

package futsal.torneos;

import futsal.equipos.Equipo;
import futsal.predio.Cancha;

public class Encuentro
{
    private Date fechaHora;
    private Cancha cancha;
    private Equipo equipo1;
    private Equipo equipo2;
    private int golesEquipo1 = 0;
    private int golesEquipo2 = 0;
    private boolean jugado = false;
    ...
}

```

En este caso es necesaria la sentencia import para poder utilizar la clase Equipo ya que se encuentra en otro paquete sobre el que no tenemos visibilidad. Como práctica podrías definir el método obtenerEquipoGanador() que nos retorne el Equipo ganador en caso de que el partido ya se haya jugado?

Herencia de Clases

Como mencionamos en la sección anterior Modelo de Objetos anterior, el concepto de Herencia es de suma importancia en el Paradigma Orientado a Objetos y es una herramienta muy útil a la hora de diseñar nuestros programas. La Herencia define relaciones del tipo "es un" como, por ejemplo "un Alumno es una Persona" por lo que comparte tanto sus atributos como nombre, apellido, dni y su comportamiento, así como tiene sus propios atributos como las materias a las que está inscripto, cuáles ha rendido, entre otras.

Es posible construir una nueva clase a partir de otra mediante el mecanismo de la *herencia*. Mediante el uso de la herencia las clases pueden extender el comportamiento de otra clase permitiendo con ello un gran aprovechamiento del código. Aunque su principal virtud es el Polimorfismo, que veremos luego. Cuando una clase deriva de otra, hereda todas sus variables y métodos. Estas funciones y variables miembro pueden ser *redefinidas (overridden)* en la clase derivada, que puede también definir o añadir nuevas variables y métodos. En cierta forma es como si la *sub-clase* (la clase derivada) “contuviera” un objeto de la *super-clase*; en realidad lo “amplía” con nuevas variables y métodos.

Java permite múltiples niveles de herencia, pero no permite que una clase derive de varias (no es posible la herencia múltiple). Se pueden crear tantas clases derivadas de una misma clase como se quiera.

La herencia induce una jerarquía en forma de árbol sobre las clases con raíz en la clase *Object*. Una clase se dice que hereda o extiende a otra clase antecesora. La palabra reservada *extends* sirve para indicar que una clase extiende a otra. La clase que extiende a otra, hereda todos los atributos y métodos de la clase antecesora, los atributos y métodos privados no tienen visibilidad. La clase antecesora puede extender a su vez otra clase.

Todas las clases de *Java* creadas por el programador tienen una *superclase*. Cuando no se indica explícitamente una *superclase* con la palabra *extends*, la clase deriva de *java.lang.Object*, que es la clase raíz de toda la jerarquía de clases de *Java*. Como consecuencia, todas las clases tienen algunos métodos que han heredado de *Object*.

Un ejemplo de herencia puede ser el siguiente:

```
Persona.java
public class Persona {
    private String nombre;
    private String apellido;
    private String dni;

    public String getNombreCompleto () {
        return this.nombre + " " + this.apellido;
    }

    // ...
}
```

```
Alumno.java
public class Alumno extends Persona {
    private int legajo;
    private Materia[] materiasInscriptas;
    private Materia[] materiasRendidas;

    // ...
}
```

De esta forma cualquier objeto de la clase *Alumno* también tendrá los atributos *nombre*, *apellido* y *dni* y podrá hacer uso del método público *getNombreCompleto()*.

Sobrescritura de variables y métodos

Una clase puede *redefinir* (volver a definir) cualquiera de los métodos heredados de su *superclase* que no sean *final*. El nuevo método sustituye al heredado para todos los efectos en la clase que lo ha redefinido. Además, una clase que extiende a otra puede declarar atributos con el mismo nombre que algún atributo de la clase a la que extiende; se dice que el atributo se ha sobreescrito u ocultado. Un uso de sobre escritura de atributos es la extensión del tipo.

Los métodos de la *super-clase* que han sido redefinidos pueden ser todavía accedidos por medio de la palabra *super* desde los métodos de la clase derivada, aunque con este sistema sólo se puede subir un nivel en la jerarquía de clases.

Finalmente, un método declarado *static* en una clase antecesora puede sobreescibirse en una clase que la extienda, pero también se debe declarar *static*; de lo contrario se producirá un error en tiempo de compilación.

En el ejemplo anterior podríamos redefinir el método *getNombreCompleto()* para que incluya el número de legajo del Alumno a la hora de mostrar su nombre en pantalla.

```
public class Alumno extends Persona {
    private int legajo;
    private Materia[] materiasInscriptas;
    private Materia[] materiasRendidas;

    public String getNombreCompleto () {
        return this.legajo + ":" + super.getNombreCompleto();
    }

    // ...
}
```

Otro ejemplo puede ser el de las Cuentas Bancarias, donde nuestra clase *CuentaBancaria* es heredada por *CuentaCorriente* y *CajaDeAhorro*. De esta forma la clase heredad contiene el atributo *saldo* que es común a ambos tipos de cuenta y define ciertos métodos también comunes como *depositar()* y *extraer()* que reciben un monto como parámetros. Al ser conocida por nosotros la interfaz de la clase *CuentaBancaria* podríamos acceder a estos métodos comunes sin conocer explícitamente la clase que la hereda, esto se conoce como Polimorfismo.

```
CuentaBancaria miCuenta = new CajaDeAhorros();
// realizamos una extracción usando el objeto miCuenta
// cuyo tipo de datos fue definido como CuentaBancaria
miCuenta.extraer(100);
CuentaBancaria miOtraCuenta = new CuentaCorriente();
// realizamos una extracción usando el objeto miOtraCuenta
// cuyo tipo de datos fue definido como CuentaBancaria
// aunque es una instancia de CuentaCorriente
miOtraCuenta.extraer(100);
```

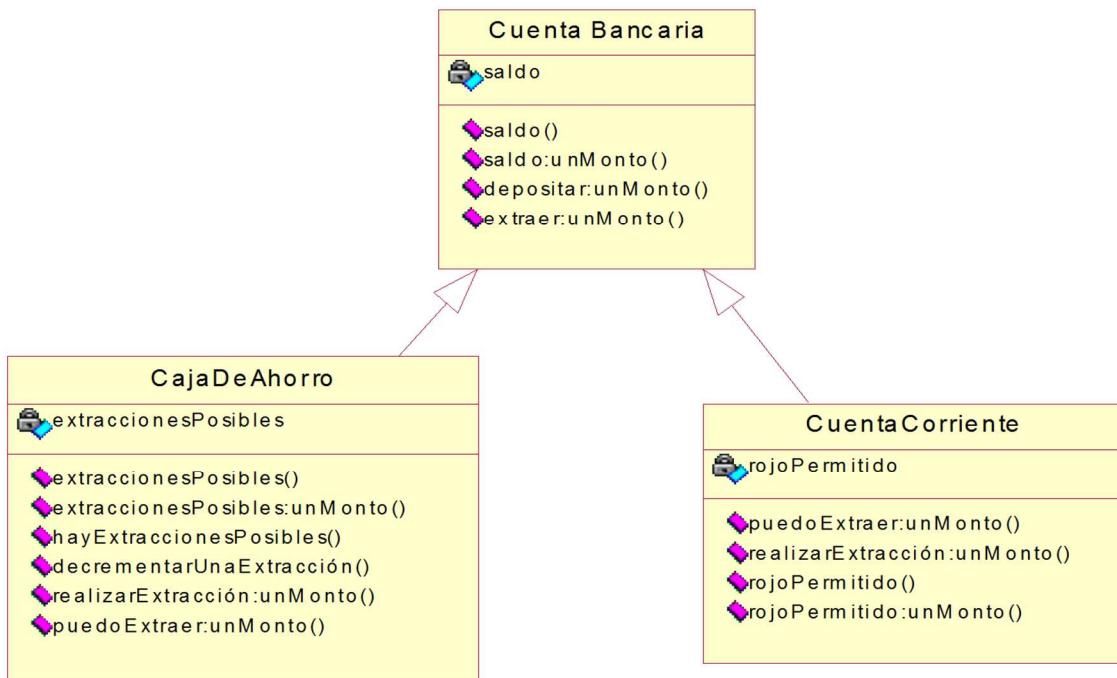


Fig. 46- Ejemplo de herencia y polimorfismo con la Jerarquía de clases de CuentaBancaria

Clases y métodos abstractos

Las clases abstract funcionan como plantillas para la creación de otras clases, son clases de las que no se pueden crear objetos. Su utilidad es permitir que otras clases deriven de ella, proporcionándoles un marco o modelo que deben seguir y algunos métodos de utilidad general. Las clases abstractas se declaran anteponiéndoles la palabra *abstract*, como, por ejemplo

```
public abstract class Geometria { ... }
```

Un método se puede declarar como *abstract*. Un método declarado de esta manera no implementa nada. Si una clase contiene uno o más métodos declarados como *abstract*, ella a su vez debe ser declarada como *abstract*. Las clases que la extiendan deben *obligatoriamente* sobrescribir los métodos declarados como *abstract* o la clase también debe declararse *abstract*.

Interfaces

Así como la Herencia representa relaciones del tipo “es un”, las interfaces nos permiten representar relaciones “se comporta como un”. El concepto de la implementación de interfaces garantiza que objetos de una Clase con esa interfaz tendrán disponibles ciertos métodos definidos. Es en este punto que entra el concepto de Polimorfismo, ya que cualquier objeto independientemente de su clase que implemente una interfaz determinada podrá responder ante la llamada a esos métodos definidos. Las *interfaces* permiten “publicar” el comportamiento de una clase develando un mínimo de información. El nombre de una *interface* se puede utilizar como un *nuevo tipo de referencia*. En este sentido, el nombre de una interface puede ser utilizado en lugar del nombre de cualquier clase que la implemente, aunque su uso estará restringido a los métodos de la *interface*. Un objeto de ese tipo puede también ser utilizado como valor de retorno o como argumento de un método.

Si queremos que una determinada clase vaya a tener cierto comportamiento, hacemos que implemente una determinada interface. En la interface no se implementa el comportamiento, únicamente se especifica cuál va a ser, es decir, se definirán los métodos, pero no se implementan. No se pueden crear instancias de una interface. Todos los métodos declarados en una interface son públicos, así como sus atributos y la misma interface.

Una *clase* puede *implementar* una o varias *interfaces*. Para indicar que una clase implementa una o más interfaces se ponen los nombres de las *interfaces*, separados por comas, detrás de la palabra *implements*, que a su vez va siempre a la derecha del nombre de la clase o del nombre de la superclase en el caso de herencia.

Veamos un ejemplo:

```
public interface DiagramadorDeTorneo {
    public List<Partidos> diagramar (List<Equipos> equipos, Date fechaInicio);
}
```

La interfaz anterior define un método para poder diagramar un Torneo para el cual tenemos diferentes formas de hacerlo, puede ser por eliminatorias, todos contra todos, entre otras formas. Este método debe recibir una lista con los Equipos que se enfrentarán y una fecha que representa el inicio del Torneo. Por otro lado, retorna una lista con los partidos a jugar para la forma de diagramación correspondiente. Así podríamos tener dos clases que implementen esta interfaz:

```
public class TodosContraTodos implements DiagramadorDeTorneo {
    public List<Partidos> diagramar (List<Equipos> equipos, Date fechaInicio) {
        // acá irá el código que genera los partidos con la combinación de todos los
        equipos
    }
}

public class PorEliminatorias implements DiagramadorDeTorneo {
    public List<Partidos> diagramar (List<Equipos> equipos, Date fechaInicio) {
        // acá irá el código que genera los partidos con llaves de eliminatorias
    }
}
```

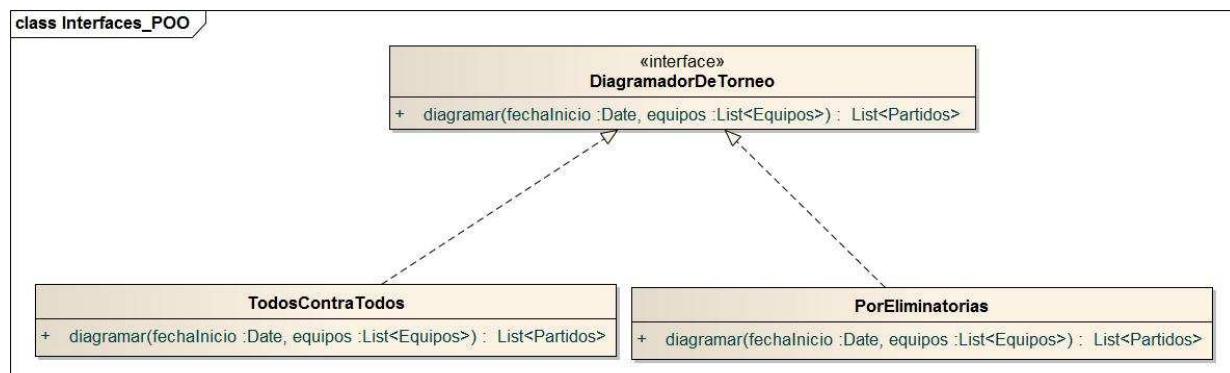


Fig. 47- Ejemplo de interfaces – Diagramación de Torneos

Documentación de Clases y Métodos

Documentar un proyecto es algo fundamental de cara a su futuro mantenimiento. Cuando programamos una clase, debemos generar documentación lo suficientemente detallada sobre ella como para que otros programadores sean capaces de usarla sólo con su interfaz. No debe existir necesidad de leer o estudiar su implementación, lo mismo que nosotros para usar una clase la biblioteca de Java no leemos ni estudiamos su código fuente.

Javadoc es una utilidad de Oracle para la generación de documentación en formato de página web a partir de código fuente Java. Javadoc es el estándar para documentar clases de Java. La mayoría de los IDEs utilizan javadoc para generar de forma automática documentación de clases.

La documentación a ser utilizada por javadoc se escribe en comentarios que comienzan con `/**` (notar el doble `*`) y que terminan con `*/`. A la vez, dentro de estos comentarios se puede escribir código HTML y operadores para que interprete javadoc (generalmente precedidos por `@`).

Tag	Descripción	Uso
<code>@author</code>	Nombre del desarrollador creador de la clase o interfaz	nombre
<code>@deprecated</code>	Indica que el método o clase es antigua y que no se recomienda su uso porque posiblemente desaparecerá en versiones posteriores	descripción
<code>@param</code>	Definición de un parámetro de un método, es requerido para todos los parámetros del método	nombre_parametro descripción
<code>@return</code>	Informa lo que devuelve el método, no se puede usar en constructores o métodos "void"	descripción
<code>@see</code>	Asocia con otro método o clase, brinda más información o elementos relacionados	referencia (<code>#método()</code> ; <code>clase#método();</code> <code>paquete.clase;</code> <code>paquete.clase#método()</code>).
<code>@throws</code>	Excepción lanzada por el método (lo veremos más adelante)	nombre_clase descripción
<code>@version</code>	Versión del método o clase	versión

Tabla 13 – Tags utilizados en Javadoc

Las etiquetas `@author` y `@version` se usan para documentar clases e interfaces. Por tanto no son válidas en cabecera de constructores ni métodos. La etiqueta `@param` se usa para documentar constructores y métodos. La etiqueta `@return` se usa solo en métodos de tipo función.

Dentro de los comentarios se admiten etiquetas HTML, por ejemplo con @see se puede referenciar una página web como link para recomendar su visita de cara a ampliar información.

Un ejemplo de su uso en un archivo de código puede ser:

```
package futsal.equipos;

import java.util.Date;

/**
 * Clase que representa a cualquier Jugador registrado en
 * el Torneo. Es importante considerar que sólo puede pertenecer
 * a un Equipo inscripto.
 *
 * @author joaquinleonelrobles
 * @see futsal.equipos.Equipo
 */
public class Jugador {

    private String nombre;
    private String apellido;
    private Date fechaNacimiento;

    /**
     * Calculamos la edad del jugador en base a su fecha de nacimiento
     *
     * @return Edad
     */
    public int calcularEdad () {
        // TODO recordar implementar este método
        return 0;
    }

    /**
     * Comprobamos si el Jugador puede participar en un torneo en
     * base a la edad mínima pasada por parámetros.
     *
     * @param edadMinima Edad mínima (inclusive) en años
     * @return Verdadero si puede participar
     */
    public boolean puedeParticiparPorSuEdad (int edadMinima) {
        return edadMinima >= this.calcularEdad();
    }
}
```

En el ejemplo anterior puede notarse un comentario que inicia con las letras “TODO”, se trata de una convención utilizada por los desarrolladores para denotar código que aún debe implementarse, del inglés “To Do” o “Pendiente”. Algunos IDE agregan automáticamente estos comentarios y además permiten resaltarlos para recordarnos de las porciones de código que nos faltan escribir.

```

24 public int calcularEdad () {
25     // TODO recordar implementar este metodo
26     return 0;
27 }
```

Fig. 48- Resaltado de comentarios TODO en la IDE Eclipse

La documentación generada por JavaDoc tendrá una apariencia similar a la siguiente captura de pantalla:

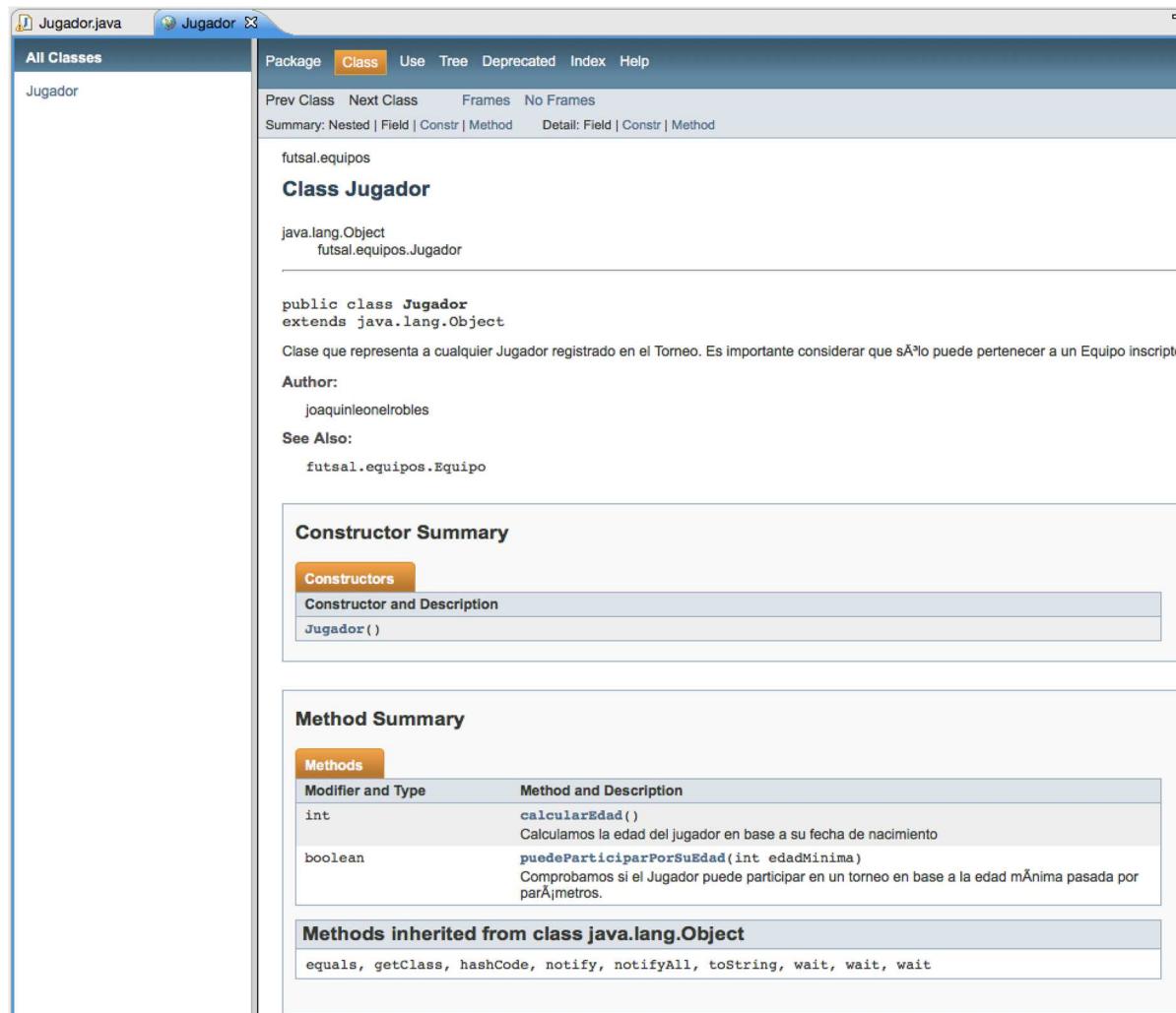


Fig. 49-. Documentación JavaDoc generada automáticamente en base a comentarios

Colecciones de tamaño variable

Anteriormente hemos mencionado los arrays de datos, los cuales son inicializados con un tamaño fijo que no podremos modificar a lo largo del desarrollo del programa. En algunos casos es útil poder tener colecciones de tamaño variable, a las cuales podamos aplicar operaciones de agregado o remoción de elementos sin tener que preocuparnos si el espacio en memoria está o no reservado. Java incorpora en su biblioteca de clases algunas alternativas muy útiles para trabajar con este tipo de colecciones, a continuación, veremos algunas de ellas.

Listas

Las listas son estructuras de datos que permiten tener cierta flexibilidad en su manejo, pueden crecer o acortarse según se lo requiera. Existen varias formas de implementar una lista en Java: en este caso se presenta un ejemplo en código utilizando punteros mediante la referencia a objetos. Una lista es una secuencia de elementos dispuestos en un cierto orden, en la que cada elemento tiene como mucho un predecesor y un sucesor. El número de elementos de la lista no suele estar fijado, ni suele estar limitado por anticipado.

Representaremos la estructura de datos de forma gráfica con cajas y flechas. Las cajas son los elementos y las flechas simbolizan el orden de los elementos.



Fig. 50- Representación de nodos en una lista ordenada

La estructura de datos deberá permitirnos determinar cuál es el primer elemento y el último de la estructura, cuál es su predecesor y su sucesor (si existen de cualquier elemento dado). Cada uno de los elementos de información suele denominarse **nodo**.

La interfaz java.util.List

Esta interfaz normalmente acepta elementos repetidos o duplicados y al igual que los arrays es lo que se llama “basada en 0”, esto quiere decir que el primer elemento no es el que está en la posición “1”, sino en la posición “0”.

Esta interfaz proporciona iterador especial que nos permite recorrer los distintos elementos de la lista. Este iterador permite además de los métodos definidos por cualquier iterador (estos métodos son hasNext, next y remove) métodos para inserción de elementos y reemplazo, acceso bidireccional para recorrer la lista y un método proporcionado para obtener un iterador empezando en una posición específica de la lista.

Debido a la gran variedad y tipo de listas que puede haber con distintas características como permitir que contengan o no elementos nulos, o que tengan restricciones en los tipos de sus elementos, hay una gran cantidad de clases que implementan esta interfaz.

Es posible restringir de qué clase serán instancia los objetos que pertenezcan a la lista utilizando los símbolos < y > en la definición de la variable, como, por ejemplo:

```
List<Jugador> jugadores;
```

De esta forma la lista jugadores sólo podrá estar compuesta por elementos que sean instancias de la clase Jugador.

A continuación, veremos las siguientes clases que implementan esta interfaz:

- ArrayList
- Stack

La Clase java.util.ArrayList

ArrayList, como su nombre lo indica, basa la implementación de la lista en un array. Eso sí, un array dinámico en tamaño (es decir, de tamaño variable), pudiendo agrandarse o disminuirse el número de elementos. Implementa todos los métodos de la interfaz List y permite incluir elementos null.

Un beneficio de usar esta implementación de List es que las operaciones de acceso a elementos, capacidad y saber si es vacía o no se realizan de forma eficiente y rápida. Todo arraylist tiene una propiedad de capacidad, aunque cuando se añade un elemento esta capacidad puede incrementarse. Java amplía automáticamente la capacidad de un arraylist cuando sea necesario. A través del código podemos incrementar la capacidad del arraylist antes de que este llegue a llenarse, usando el método ensureCapacity.

A continuación, veremos un ejemplo en el cual calculamos la edad promedio de un grupo de Jugadores.

```
package futsal.equipos;

public class Jugador {

    private String nombre;
    private String apellido;
    private int edad;

    public Jugador (nombre, apellido, edad) {
        this.nombre = nombre;
        this.apellido = apellido;
        this.edad = edad;
    }
}
```

En el cuerpo de nuestro programa creamos algunos jugadores con sus edades, los incluimos en un ArrayList y luego lo recorremos para obtener el promedio de edades.

```
// creamos los jugadores con sus edades
Jugador j1 = new Jugador("Miguel", "Houlebecq", 21);
Jugador j2 = new Jugador("Pablo", "Auster", 19);
Jugador j3 = new Jugador("Aldo", "Huxley", 23);
Jugador j4 = new Jugador("Alejandro", "Baricco", 21);

// inicializamos la variable acumuladora de promedios
float promedio = 0;

// inicializamos la lista de jugadores
// se especifica el tipo de datos de los elementos de la lista
// entre cocodrilos (< y >) para evitar insertar otro tipo de objetos
List<Jugador> jugadores = new ArrayList<Jugador>();

// agregamos un elemento a la lista con el método add()
jugadores.add (j1);
jugadores.add (j2);
```

```

jugadores.add (j3);
jugadores.add (j4);

// obtenemos un objeto Iterador que nos permita recorrer la lista
Iterator<Jugador> iter = jugadores.iterator();

// mientras exista un elemento siguiente por recorrer
while (iter.hasNext()) {
    // obtenemos el Jugador siendo recorrido
    Jugador j = iter.next();

    // acumulamos su edad promedio sobre el total de jugadores
    // que obtenemos con el método size() de la lista
    promedio += j.getEdad() / jugadores.size();
}

// mostramos el resultado en pantalla
System.out.println("Promedio: " + promedio);

```

A continuación, se listan algunos de los métodos más relevantes de la clase ArrayList:

Retorno	Método y descripción
boolean	add(E e) Agrega el elemento al final de la lista
void	add(int index, E element) Agrega el elemento en la posición especificada
boolean	addAll(Collection<? extends E> c) Agrega todos los elementos de otra colección al final de esta lista
boolean	addAll(int index, Collection<? extends E> c) Agrega todos los elementos de otra colección en la posición especificada
void	clear() Elimina todos los elementos de la lista
boolean	contains(Object o) Devuelve true si el objeto o existe en la lista
void	ensureCapacity(int minCapacity) Incrementa la capacidad del array subyacente para asegurar la capacidad especificada
E	get(int index) Devuelve el elemento de la posición solicitada
int	indexOf(Object o) Devuelve la posición del objeto solicitado
boolean	isEmpty() Devuelve true si la lista no contiene elementos
Iterator<E>	iterator() Devuelve un objeto iterador para recorrer la lista
E	remove(int index)

	Elimina el elemento de la posición especificada
boolean	remove(Object o)
	Elimina la primera ocurrencia del objeto en la lista
E	set(int index, E element)
	Reemplaza el elemento en la posición especificada
int	size()
	Devuelve el tamaño de la lista

Tabla 14. Extracto del JavaDoc para la clase ArrayList

La clase java.util.Stack

Esta clase es una implementación de la estructura de datos que vimos en el módulo de Técnicas de Programación denominada Pila, una estructura de tipo LIFO (Last In First Out, último en entrar primero en salir). Provee las operaciones de push (colocar) y pop (extraer) así como otros métodos accesorios como el size(), peek (consulta el primer elemento de la cima de la pila), empty (que comprueba si la pila está vacía) y search (que busca un determinado elemento dentro de la pila y devuelve su posición dentro de ella).

Veamos un ejemplo apilando Jugadores:

```
// creamos los jugadores con sus edades
Jugador j1 = new Jugador("Miguel", "Houellebecq", 21);
Jugador j2 = new Jugador("Pablo", "Auster", 19);
Jugador j3 = new Jugador("Aldo", "Huxley", 23);
Jugador j4 = new Jugador("Alejandro", "Baricco", 21);

// inicializamos la pila de jugadores
Stack<Jugador> pila = new Stack<Jugador>();

// agregamos un elemento a la lista con el método add()
pila.push (j1);
pila.push (j2);
pila.push (j3);
pila.push (j4);

// mientras existan elementos por extraer
while (!pila.empty()) {
    // mostramos el nombre del jugador en pantalla
    System.out.println (pila.pop().getNombreCompleto());
}
```

La salida de nuestro programa será la siguiente, respetando el orden de extracción Último en Entrar Primero en Salir:

Alejandro Baricco

Aldo Huxley

Pablo Auster

Miguel Houellebecq

A continuación, presentamos un extracto del JavaDoc de la clase Stack con los métodos más relevantes:

Retorno	Método y Descripción
boolean	empty() Devuelve true si la pila está vacía
E	peek() Consultamos el primer elemento de la pila sin extraerlo
E	pop() Extraemos el primer elemento de la pila
E	push(E item) Colocamos un elemento en el tope de la pila
int	search(Object o) Devuelve la posición del elemento en la pila (basado en 1)

Tabla 15. Extracto del JavaDoc para la clase Stack

Excepciones

Como mencionamos anteriormente, en Java existe un tipo especial de errores denominado Excepción. El mejor momento para detectar los errores es durante la compilación. Sin embargo, prácticamente sólo los errores de sintaxis son detectados durante este periodo. El resto de problemas surgen durante la ejecución de los programas.

Una “Exception” es un cierto tipo de error o una condición anormal que se ha producido durante la ejecución de un programa. Algunas excepciones son fatales y provocan que se deba finalizar la ejecución del programa. En este caso conviene terminar ordenadamente y dar un mensaje explicando el tipo de error que se ha producido. Otras, como por ejemplo no encontrar un archivo en el que hay que leer o escribir algo, pueden ser recuperables. En este caso el programa debe dar al usuario la oportunidad de corregir el error (indicando una nueva localización del fichero no encontrado).

Para poder trabajar con excepciones y capturarlas para poder desarrollar un programa que pueda ser tolerante a errores se utilizan los bloques try-catch-finally que enunciamos anteriormente cuando hablamos de Estructuras de Control. Ahora veremos con mayor profundidad cómo se utilizan.

Los errores se representan mediante dos tipos de clases derivadas de la clase *Throwable*: *Error* y *Exception*, sobre estas últimas es que trabajaremos en nuestros programas.

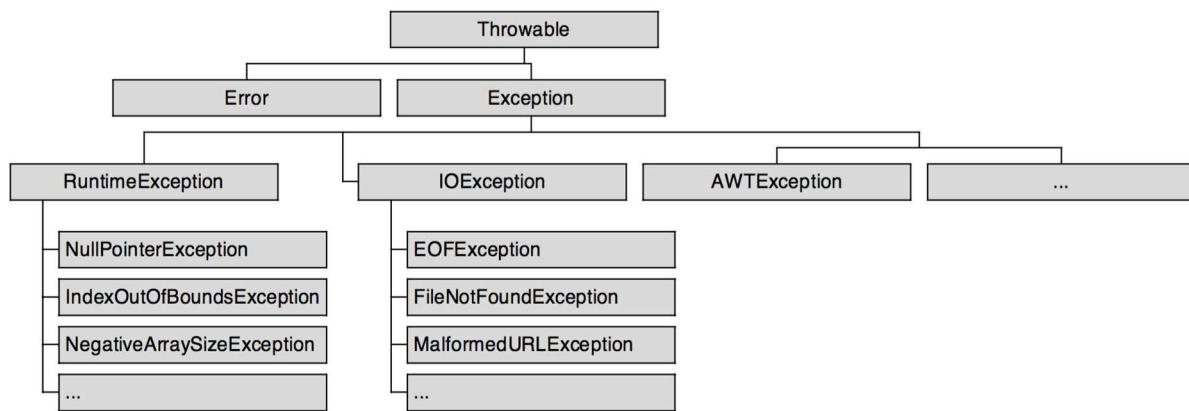


Fig. 51- Jerarquía de Excepciones en el lenguaje Java

Las clases derivadas de *Exception* pueden pertenecer a distintos packages de *Java*. Algunas pertenecen a *java.lang* (*Throwable*, *Exception*, *RuntimeException*, ...); otras a *java.io* (*EOFException*, *FileNotFoundException*, ...) o a otros packages. Por heredar de *Throwable* todos los tipos de excepciones pueden usar los siguientes métodos:

1. `String getMessage()`
Extrae el mensaje asociado con la excepción
2. `String toString()`
Devuelve un String que describe la excepción
3. `void printStackTrace()`
Indica el método donde se lanzó la excepción

Además disponemos de los siguientes constructores propios de la clase *java.lang.Exception* que podemos invocar en nuestra clase:

1. `Exception()`
2. `Exception(String message)`

El primero determina un mensaje nulo para la causa de la excepción mientras que el segundo nos permite especificar el mensaje en la invocación al constructor de la clase padre.

En base a lo anterior, podemos declarar nuestras propias excepciones a la hora de desarrollar nuestros programas para manejar los errores propios de la funcionalidad que implementemos; por ejemplo, para el caso de un Encuentro entre dos Equipos, a la hora de intentar obtener el equipo ganador podríamos lanzar una excepción en el caso de que el partido aún no se haya jugado. Para crear nuestra propia excepción sólo tenemos que hacer lo siguiente:

```

public class PartidoNoJugadoException extends Exception
{
    // acá podríamos declarar y redefinir los métodos enunciados más arriba
    // por ahora sólo nos ocuparemos del mensaje de error

    public String getMessage ()
    {
        return "El partido aún no se ha jugado";
    }
}
  
```

```

    }
}

```

De la misma forma podemos especificar el mensaje de la causa declarando el constructor de nuestra clase invocando al constructor de la clase Exception:

```

public class PartidoNoJugadoException extends Exception
{
    public PartidoNoJugadoException()
    {
        super("El partido aún no se ha jugado");
    }
}

```

Es importante notar que es una buena práctica agregar como sufijo al nombre del método la palabra Exception para saber de antemano la naturaleza de esta clase.

En la implementación del método obtenerEquipoGanador() deberíamos agregar la cláusula throws para indicar que el método puede arrojar una excepción de la que explícitamente deberemos hacernos cargo:

```

public Equipo obtenerEquipoGanador() throws PartidoNoJugadoException
{
    // comprobamos si el partido ya se ha jugado
    if (jugado)
    {
        // comparamos la cantidad de goles
        if (golesEquipo1 > golesEquipo2)
            return equipo1; //Cuando el bloque if tiene una única sentencia se
                           //puede evitar el uso de {}
        else
        {
            if (golesEquipo1 < golesEquipo2)
                return equipo2;
            else
                return null;// hubo un empate por lo que devolvemos nulo
        }
    }
    // el partido aún no se ha jugado por lo que lanzamos una excepcion
    else
    {
        throw new PartidoNoJugadoException();
    }
}

```

Por otro lado, también podríamos redefinir el constructor de la clase de excepción y agregar parámetros propios del dominio de nuestro problema. En este caso podríamos, por ejemplo, agregar cuál fue el partido del cual intentamos obtener el resultado.

Durante la ejecución de nuestro programa podríamos tratar de obtener el equipo ganador de la siguiente forma:

```

Encuentro encuentro = new Encuentro();
encuentro.setEquipo1(losQuirquinchosVerdes);
encuentro.setEquipo2(losPadecientesFC);

// como la llamada a nuestro método puede arrojar una excepción debemos
// encerrarla en un bloque try-catch
try
{
    Equipo ganador = encuentro.obtenerEquipoGanador();
    System.out.println("El equipo ganador fue: " + ganador);
}
catch (PartidoNoJugadoException ex)
{
    System.out.println("No se puede obtener el equipo ganador porque: " +
ex.getMessage());
}

```

En el ejemplo anterior podemos notar que al capturar la excepción con el bloque catch() además obtenemos una instancia, un objeto de la clase PartidoNoJugadoException, lo que nos permite llamar al método getMessage() para conocer la causa del problema.

La salida del ejemplo anterior sería:

No se puede obtener el equipo ganador porque: El Partido aún no se ha jugado

Ahora que tenemos nuestras excepciones definidas y creadas es necesario arrojarlas en el momento en que se producen durante la ejecución de nuestro programa, para ello disponemos de la palabra reservada throw cuya sintaxis es la siguiente:

```

// alguna condición que genera nuestra excepción
throw objetoDeExcepcion;

```

A los fines de abreviar el código es frecuente ver la creación de los objetos instancia de la clase de excepción en la misma línea en que son arrojados; como, por ejemplo:

```

// alguna condición que genera nuestra excepción
throw new PartidoNoJugadoException();

```

Es importante destacar la diferencia con la palabra reservada throws que informa que un método en particular puede arrojar una excepción de un tipo determinado y deberá ser manejada por el método que lo invoque.

Por otro lado, al reformular el ejemplo agregando el siguiente bloque antes del try-catch la ejecución tomaría el camino feliz mostrándonos al flamante ganador del encuentro: Los Padecientes FC.

```

encuentro.setJugado (true);
encuentro.setGolesEquipo1 (1);
encuentro.setGolesEquipo2 (9); // <-- goleada!

```

Jerarquía de Excepciones

Como vimos anteriormente todas las excepciones heredan de la clase `java.lang.Exception`, pero demás podemos heredar de otras excepciones que finalmente hereden ésta clase. De esta forma podemos diseñar una jerarquía de excepciones para ir del caso más general al más particular en la definición de nuestros errores, veamos un ejemplo expandiendo la jerarquía de la excepción `PartidoNoJugadoException`:

```
public class PartidoNoJugadoException extends ResultadoDePartidoException
{
    // el código de nuestra excepción va aquí
}

public class PartidoCanceladoException extends ResultadoDePartidoException
{
    // el código de nuestra excepción va aquí
}

public class ResultadoDePartidoException extends Exception
{}
```

Entonces ahora tenemos dos excepciones para tratar los posibles errores a la hora de intentar obtener el resultado de un partido en base su estado. Además, a la hora de capturarlas podemos hacer uso de múltiples bloques `catch`, de la siguiente forma:

```
try
{
    Equipo ganador = encuentro.obtenerEquipoGanador();
    System.out.println("El equipo ganador fue: " + ganador);
}
catch (PartidoNoJugadoException ex)
{
    System.out.println("No se puede obtener el equipo ganador porque el partido aún no se ha jugado");
}
catch (PartidoCanceladoException ex)
{
    System.out.println("No se puede obtener el equipo ganador porque el partido fue cancelado");
}
```

O utilizando la propiedad del polimorfismo propia del lenguaje de programación orientado a objetos podemos capturar todas las excepciones que heredan de `ResultadoDePartidoException` con sólo un bloque `catch`, usando:

```
try
{
    Equipo ganador = encuentro.obtenerEquipoGanador();
    System.out.println("El equipo ganador fue: " + ganador);
}
```

```
catch (ResultadoDePartidoException ex)
{
    System.out.println("No se puede obtener el equipo ganador porque" + ex.getMessage());
}
```

Y usando una combinación de ambas estrategias podemos tomar acciones para algunas excepciones en particular y, en los demás casos, realizar una operación para todos los demás casos como, por ejemplo:

```
try
{
    Equipo ganador = encuentro.obtenerEquipoGanador();
    System.out.println("El equipo ganador fue: " + ganador);
}
catch (PartidoNoJugadoException ex)
{
    System.out.println("No se puede obtener el equipo ganador porque el partido aún no se ha jugado");
}
catch (ResultadoDePartidoException ex)
{
    System.out.println("No se puede obtener el equipo ganador porque" + ex.getMessage());
}
```

Es importante destacar que cada bloque catch() es evaluado en orden y sólo se ejecutará el primero que corresponda a la excepción arrojada, por lo cual deberemos declararlos en orden de las excepciones más particulares a las más generales. Si en el ejemplo anterior hubiésemos capturado primero ResultadoDePartidoException antes de PartidoNoJugadoException, el último bloque catch nunca sería evaluado ya que todos los casos serían capturados con el primer bloque más general.

La API de Java

Como el lenguaje Java es un lenguaje orientado a objetos, la API de Java provee de un conjunto de clases utilitarias para efectuar toda clase de tareas necesarias dentro de un programa. Se trata de un conjunto de clases de propósito general ya desarrolladas y probadas, que nos permiten ahorrar tiempo a la hora de programar nuestras aplicaciones. Algunas de ellas ya las hemos mencionado y hasta utilizado en ejemplos anterior, como ArrayList, Stack y Exception. La API es parte integral del lenguaje de programación y está implementada para cada máquina virtual independientemente de la plataforma donde ejecutemos nuestro código, por lo cual podemos confiar en su portabilidad.

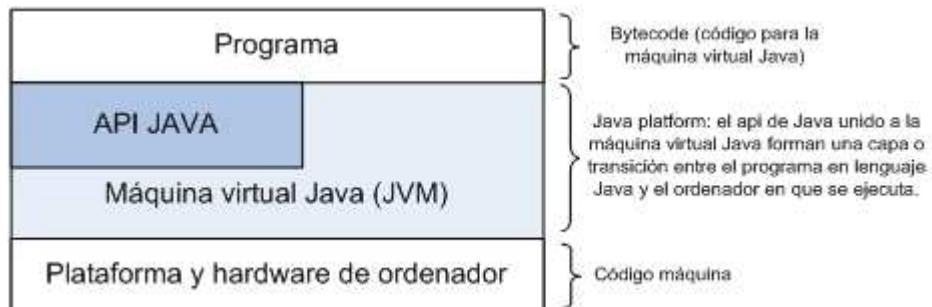


Fig. 52- La API de Java en contexto

La API Java está organizada en paquetes lógicos, donde cada paquete contiene un conjunto de clases relacionadas semánticamente. A continuación, se muestra un esquema que describe en términos generales la estructura de la API.

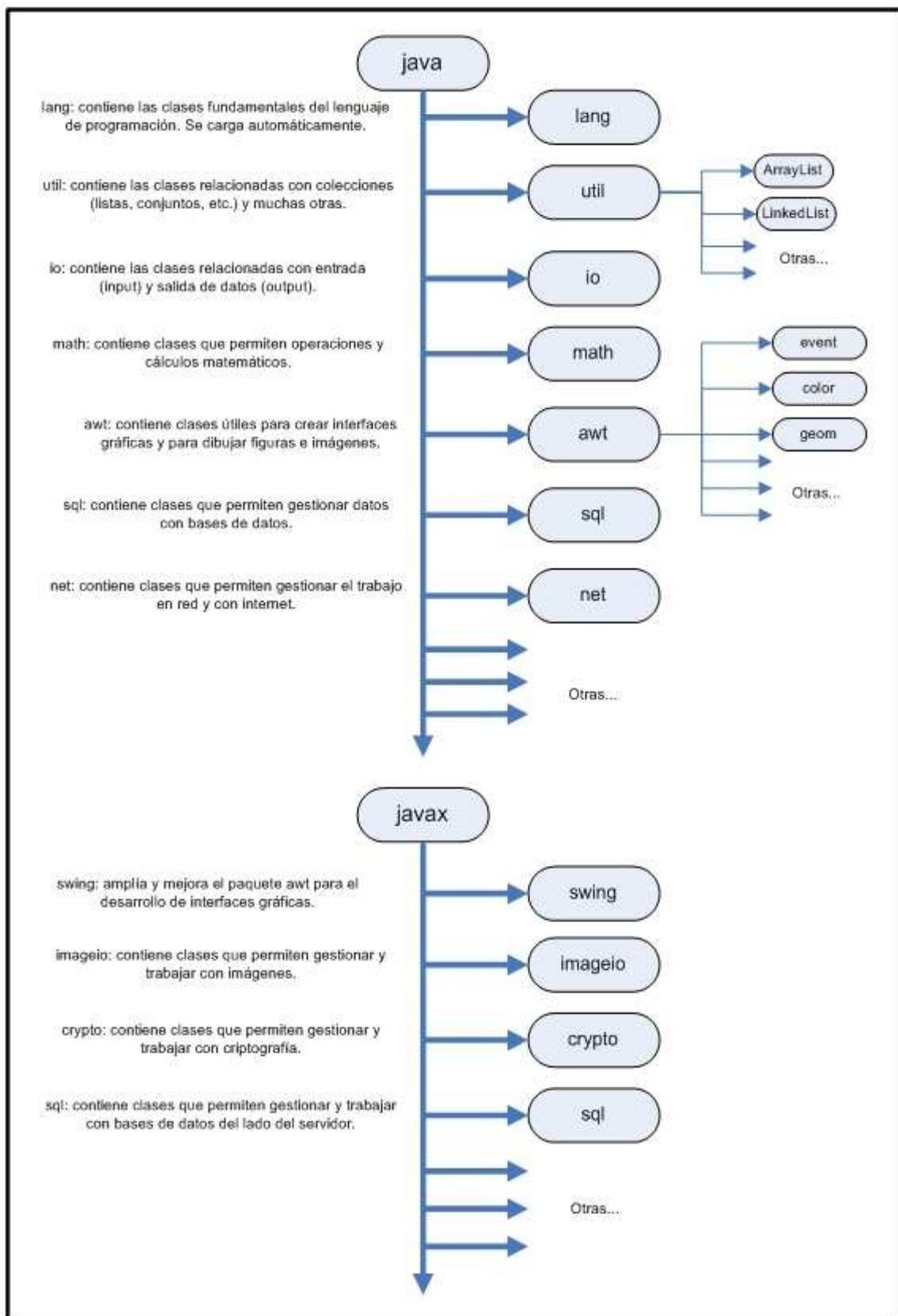


Fig. 53- Estructura de paquetes general para la API de Java 8.

Las librerías podemos decir que se organizan en ramas como si fueran las ramas de un árbol. Vamos a fijarnos en dos grandes ramas: la rama “java” y la rama “javax”. La rama java parte de los orígenes de Java y contiene clases más bien generales, mientras que la rama javax es más moderna y tiene paquetes con fines mucho más específicos.

Encontrar un listado de librerías o clases más usadas es una tarea casi imposible. Cada programador, dependiendo de su actividad, utiliza ciertas librerías que posiblemente no usen otros programadores. Los programadores centrados en programación de escritorio usarán clases diferentes a las que usan programadores web o de gestión de bases de datos.

La especificación de la API de forma completa (en JavaDoc por supuesto) está disponible en <http://www.oracle.com/technetwork/java/api-141528.html> y es propia de cada versión del lenguaje.

Programando interfaces gráficas

Hasta ahora hemos desarrollado aplicaciones que corren en la línea de comandos del sistema operativo y cuya comunicación con el usuario se realiza mediante mensajes en formato de texto que se ingresan o muestran en la consola. Pero la gran mayoría de las aplicaciones modernas requieren interfaces visuales y gráficas para interactuar con los usuarios y cada vez están tomando mayor peso con la predominancia de los dispositivos táctiles. El AWT (Abstract Windows Toolkit) es la parte de Java que se ocupa de construir interfaces gráficas de usuario. Aunque el AWT ha estado presente en Java desde la versión 1.0, la versión 1.2 ha incorporado un modelo distinto de componentes llamado Swing con componentes modernos e independencia del sistema operativo. Esto significa que nuestras aplicaciones con interfaz gráfica podrán utilizarse tanto en sistemas Microsoft Windows, Linux y otros dibujándose con las herramientas propias de cada plataforma.

Java Swing pertenece a las JFC (Java Foundation Classes), está contenido en el paquete javax.swing y fue creada a partir de java.awt. A continuación podemos ver su estructura jerárquica de componentes.

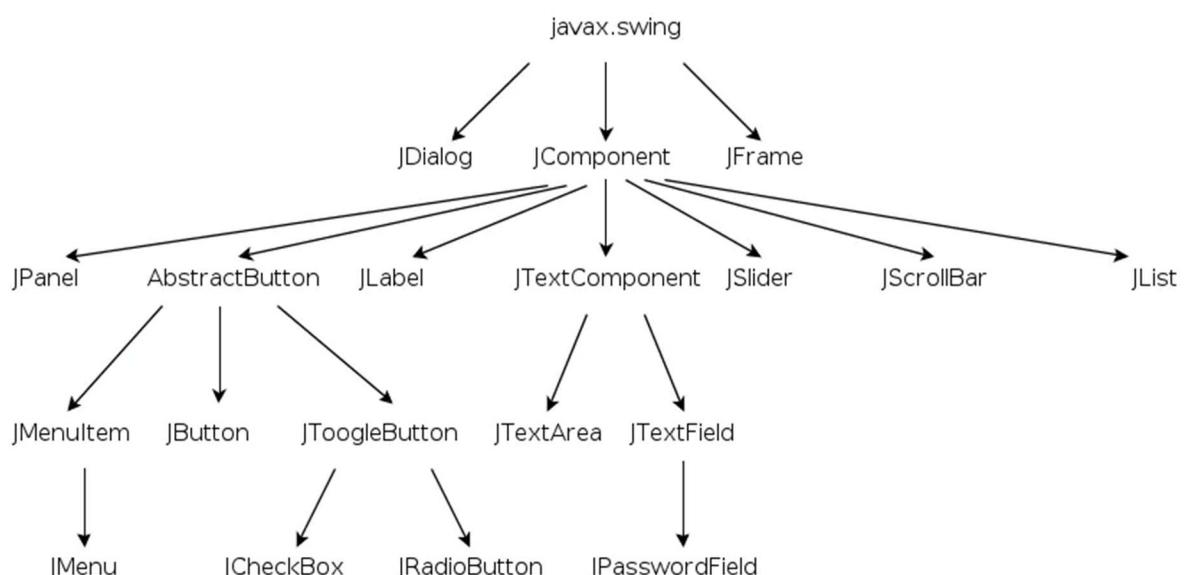


Fig. 54- Jerarquía de componentes de la biblioteca Java Swing para interfaces gráficas

En la imagen anterior es importante destacar que todos los componentes como los campos de ingreso de texto, botones, menues y demás heredan de la clase JComponent. Además, nuestras ventanas serán clases que hereden de JFrame.

El modelo de Eventos para interactuar con la aplicación

Para poder comunicarnos con nuestra aplicación mediante sus componentes dibujados en la ventana, Java utiliza un modelo de eventos donde un evento es alguna acción realizada por el usuario, como un click con el mouse, la presión de una tecla, arrastrar un componente, entre otros, que puede desencadenar una funcionalidad.

El modelo de eventos de Java está basado en que los objetos sobre los que se producen los eventos (event sources) “registran” los objetos que habrán de gestionarlos (event listeners), para lo cual los event listeners habrán de disponer de los métodos adecuados. Estos métodos se llamarán automáticamente cuando se produzca el evento. La forma de garantizar que los event listeners disponen de los métodos apropiados para gestionar los eventos es obligarles a implementar una determinada interface Listener. Las interfaces Listener se corresponden con los tipos de eventos que se pueden producir. De esta forma al producirse alguna acción del usuario que tenga un “oyente” a la espera, se llamará a la funcionalidad definida en la implementación de la interfaz Listener correspondiente. En caso de que no haya ningún oyente registrado para ese evento será descartado sin más.

Veamos un ejemplo a continuación:

HolaMundo.java

```
package ui;

import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;

public class HolaMundo extends JFrame {

    private JLabel texto1;
    private JButton boton;

    public HolaMundo() {
        setLayout (null);

        texto1 = new JLabel("Hola Mundo!");
        texto1.setBounds(100,100,200,40);

        boton = new JButton("Soy un boton");
        boton.setBounds(70,140,200,40);
        boton.addActionListener(new MiClickListener(this));

        add(texto1);
        add(boton);
    }

    public static void main(String[] args) {
        HolaMundo ventana = new HolaMundo();
    }
}
```

```

        ventana.setBounds(500,250,300,250);
        ventana.setVisible(true);
        ventana.setResizable(false);
    }
}

```

MiClickListener.java

```

package ui;

import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.JDialog;
import javax.swing.JFrame;

public class MiClickListener implements ActionListener {

    private JFrame padreFrame;

    public MiClickListener(JFrame padreFrame) {
        this.padreFrame = padreFrame;
    }

    @Override
    public void actionPerformed(ActionEvent e) {
        JDialog d = new JDialog (padreFrame, "Hola", true);
        d.setLocationRelativeTo(padreFrame);
        d.setVisible(true);
    }
}

```

En este ejemplo mostramos una ventana principal con un texto estático y un botón de acción que al ser presionado abre un cuadro de diálogo vacío.

La primera clase extiende a JFrame la cual viene a ser la ventana principal de nuestra aplicación que es inicializada en el método main() al igual que hacímos con las aplicaciones de línea de comandos. En el constructor declaramos dos componentes, uno de tipo JLabel que es un texto estático y un JButton, nuestro botón con llamado a la acción. Además de asignarles las etiquetas de texto al inicializarlos definimos su posición en la pantalla y para el caso del botón, registramos un ActionListener, es decir una clase específica que estará a la escucha de acciones realizadas sobre ese botón.

Por otro lado, hemos creado la clase MiClickListener que, como su nombre lo indica, será la encargada de estar a la escucha de clicks realizados sobre nuestro botón. Es importante destacar que esta clase debe implementar la interfaz ActionListener, para poder reescribir el método actionPerformed() que será invocado al realizar alguna acción sobre el botón. En el cuerpo de este método incializamos y mostramos dentro de la ventana principal un cuadro de diálogo vacío.

Java Swing es una biblioteca con muchos componentes y cada uno puede ser configurado y utilizado para fines específicos dentro de nuestras aplicaciones gráficas. Su documentación está disponible en formato JavaDoc en <https://docs.oracle.com/javase/7/docs/api/javax/swing/package-summary.html>.

Referencia de Figuras y Tablas

Figuras

Fig. 1 – Tarjetas perforadas para programación de computadoras	8
Fig. 2 – Instrucciones en Assembly en la vista de Terminator (1984)	9
Fig. 3 - Clases y Objetos	22
Fig. 4. Estado, comportamiento e identidad de un objeto	24
Fig. 5 - Herencia entre Clases	29
Fig. 6 Representación de la herencia de diamante	30
Fig. 7 - Abstracción en el Modelo de Objetos.....	32
Fig. 8- Las clases y objetos deberían estar al nivel de abstracción adecuado: ni demasiado alto ni demasiado bajo	33
Fig. 9 - El encapsulamiento en el Modelo de Objetos.....	34
Fig. 10 - Modularidad, cohesión y acoplamiento en el Modelo de Objetos	35
Fig. 11 - Las Abstracciones forman una Jerarquía	37
Fig. 12- La comprobación estricta de tipos impide que se mezclen abstracciones	38
Fig. 13 - La concurrencia permite a dos objetos actuar al mismo tiempo	39
Fig. 14 - Conserva el estado de un objeto en el tiempo y el espacio	40
Fig.15- La clasificación es el medio por el cual ordenamos el conocimiento.....	41
Fig. 16 - Diferentes observadores pueden clasificar el mismo objeto de distintas formas	42
Fig. 17 - Los mecanismos son los medios por los cuales los objetos colaboran para proporcionar algún comportamiento de nivel superior.....	44
Fig. 18- Un caso de uso y los escenarios que lo conforman.....	53
Fig. 19- Generalización entre actores.....	56
Fig. 20 – Ejemplo de una vista de un diagrama de casos de uso para un Complejo de Cines	59
Fig.21- Notación para una clase en UML.....	62
Fig. 22- Representación icónica de los tres tipos de clases de UML	62
Fig. 23- Notaciones para una Interface en UML, representación etiquetada e icónica.....	64
Fig. 24- Visualización de una clase con interfaces asociadas	65
Fig. 25- Notación para representar la relación de asociación entre clases en UML	65
Fig. 26- Notación para representar la relación de agregación entre clases en UML	65
Fig. 27- Notación para representar la relación de composición entre clases en UML.....	66
Fig. 28. Notación para representar la relación de generalización entre clases en UML, herencia simple	66

Fig. 29- Notación para representar la relación de generalización entre clases en UML, herencia múltiple	66
Fig. 30- Notación para representar la relación de dependencia entre clases en UML	67
Fig. 31. Notación para representar la relación de realización entre clases en UML.....	67
Fig. 32- Notación para representar la relación de contención entre clases en UML.....	67
Fig. 33- Vista parcial del Modelo de Dominio para el caso de estudio del Complejo de Cines.....	69
Fig. 34-. Notaciones para representar estados en UML.....	70
Fig. 35- Transición entre estados en un Diagrama de Máquina de Estados.	71
Fig. 36-. Diagrama de Máquina de estados para la clase Sala, utilizada en el caso del Complejo de Cines	74
Fig. 37- Diagrama de Máquina de estados que muestra el uso de una sub-máquina y historia para los estados.....	74
Fig. 38-. Elementos de modelado utilizados en un Diagrama de Secuencia	75
Fig. 39- Líneas de vida de los objetos en un Diagrama de Secuencia.....	76
Fig. 40- Tipos de mensajes que pueden utilizarse en un Diagrama de Secuencia	76
Fig. 41- Creación y destrucción de objetos en un Diagrama de Secuencia.....	78
Fig. 42- Parámetros en un Diagrama de Secuencia.....	78
Fig. 43-. Diagrama de secuencia para el escenario del curso normal del caso de uso Registrar Película	80
Fig. 44: Vista del Diagrama de clases que muestra las clases necesarias para modelar el escenario del curso normal del caso de uso Registrar Película	81
Fig. 45- Funcionamiento general del Lenguaje Java.....	82
Fig. 46- Ejemplo de herencia y polimorfismo con la Jerarquía de clases de CuentaBancaria.....	112
Fig. 47- Ejemplo de interfaces – Diagramación de Torneos	113
Fig. 48- Resaltado de comentarios TODO en la IDE Eclipse.....	116
Fig. 49-. Documentación JavaDoc generada automáticamente en base a comentarios	116
Fig. 50- Representación de nodos en una lista ordenada	117
Fig. 51- Jerarquía de Excepciones en el lenguaje Java	122
Fig. 52- La API de Java en contexto	127
Fig. 53- Estructura de paquetes general para la API de Java 8.....	128
Fig. 54- Jerarquía de componentes de la biblioteca Java Swing para interfaces gráficas	129

Tablas

Tabla 1 – Ejemplo de Clasificación de clases del dominio del problema	43
Tabla 2 – Diagramas de UML 2.0.....	49
Tabla 3 – Elementos básicos de modelado para un diagramas de Casos de Uso en UML 2.0.....	50
Tabla 4 – Tipos de Narrativas para la descripción de los Casos de Uso	52
Tabla 5 – Tipos de relaciones utilizadas para estructurar un diagrama de Casos de Uso.....	58
Tabla 6 – Ejemplo de descripción de los actores que participan en la vista del diagrama de casos de uso presentado en la Figura 14.....	60
Tabla 7 – Ejemplo de narrativa de los actores que participan en la vista del diagrama de casos de uso presentado en la Figura 14.....	61
Tabla 8 – Tipos de Mensajes que pueden utilizarse para modelar diagramas de secuencia.....	77
Tabla 9 – Tipos de básicos de datos en JAVA	88
Tabla 10 – Tipos de operadores aritméticos en JAVA	90
Tabla 11 – Tipos de operadores lógicos en JAVA	91
Tabla 12 – Operadores de Asignación en JAVA	93
Tabla 13 – Tags utilizados en Javadoc	114
Tabla 14. Extracto del JavaDoc para la clase ArrayList.....	120
Tabla 15. Extracto del JavaDoc para la clase Stack.....	121

Fuentes de Información

- **Armour Frank & Miller Granville** – ADVANCED USE CASE MODELING- EDITORIAL ADDISON WESLEY, AÑO 2000.
- **Shneider, Geri & Winters, Jason**- APPLYING USE CASES- A PRACTICAL GUIDE- ADDISON WESLEY, AÑO 1998.
- **Sommerville, Ian** - "INGENIERÍA DE SOFTWARE" 9na Edición (Editorial Addison-Wesley Año 2011).
- **Pressman Roger** - "Ingeniería de Software" 7ma. Edición - (Editorial Mc Graw Hill Año 2010).
- **Jacobson, Booch y Rumbaugh** - "EL PROCESO UNIFICADO DE DESARROLLO" (Editorial Addison-Wesley - Año 2000 1^a edición).
- **Booch, Rumbaugh y Jacobson** - "Lenguaje de Modelado Unificado" - (Editorial Addison-Wesley-Pearson Educación – 2da edición - Año 2006).
- **Booch, Grady** - Análisis y Diseño Orientado a Objetos. (Editorial Addison-Wesley/Díaz de Santos Año 1996).
- **Jacobson, Ivar** - Object-Oriented Software Engineering. (Editorial Addison-Wesley Año 1994).
- https://es.wikipedia.org/wiki/Tipo_de_dato#Java
- **Belmonte Fernández Oscar** - Introducción al lenguaje de programación Java. Una guía básica
- **Sánchez Asenjo Jorge** – Programación Básica en Lenguaje Java. (<http://www.jorgesanchez.net> Año 2009)
- **García de Jalón Javier, Rodríguez, José Ignacio, Mingo Iñigo, Imaz Aitor, Brazález Alfonso, Larzabal Alberto, Calleja Jesús, García Jon** – Aprenda Java como si estuviera en primero (Escuela Superior de Ingenieros Industriales, Universidad de Navarra Año 1999)
- <http://java-white-box.blogspot.com.ar/2012/08/javadoc-que-es-el-javadoc-como-utilizar.html>
- <http://www.ecured.cu/Javadoc>
- http://www.ciberaula.com/articulo/listas_en_java
- http://www.aprenderaprogramar.com/index.php?option=com_content&view=article&id=603:interface-list-del-api-java-clases-arraylist-linkedlist-stack-vector-ejemplo-con-arraylist-cu00917c&catid=58:curso-lenguaje-programacion-java-nivel-avanzado-i&Itemid=180
- <https://geekytheory.com/tutorial-14-java-swing-interfaces-graficas/>