

# **Ministerio de Producción**

## **Secretaría de Emprendedores y de la Pequeña y Mediana Empresa**

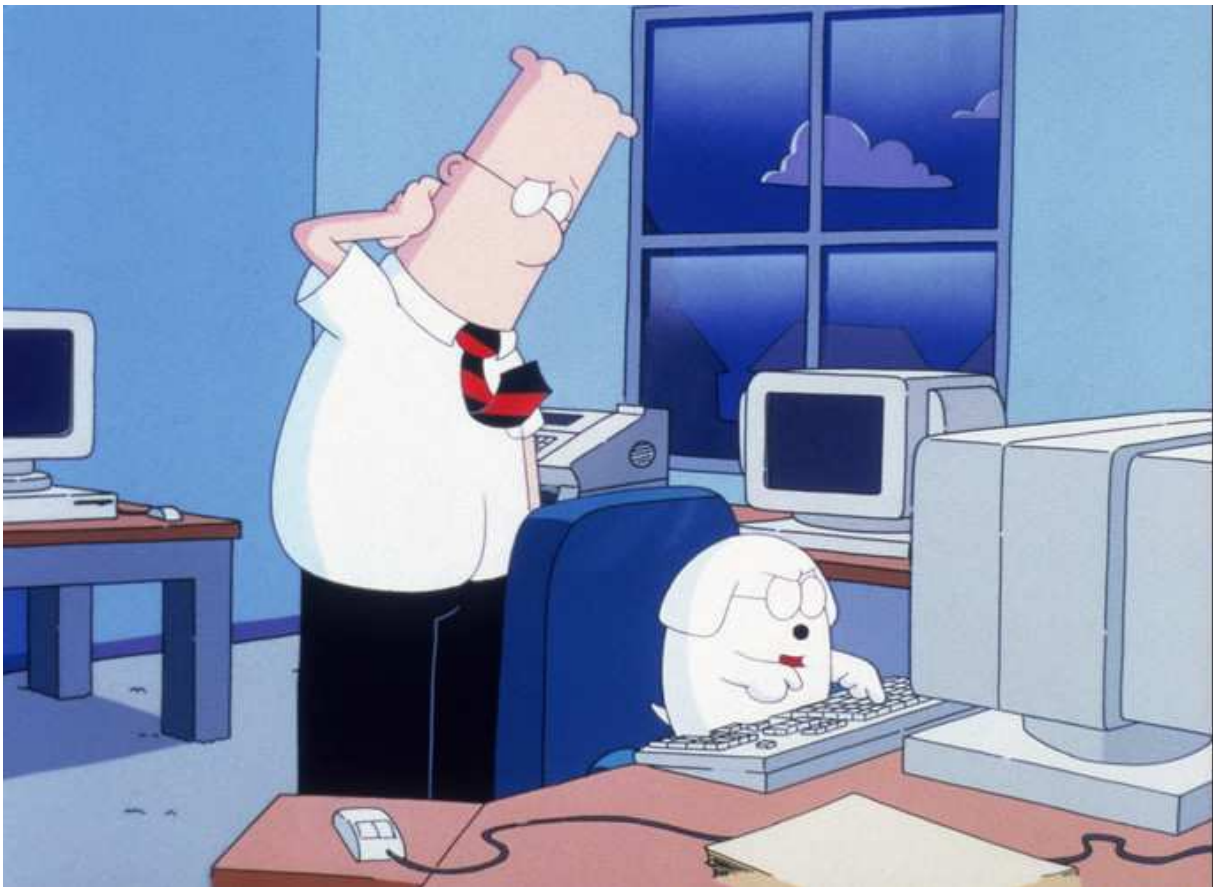
### **Dirección Nacional de Servicios Basados en el Conocimiento**



**Programadores**

# Apunte del Módulo

## Programación Orientada a Objetos



## Tabla de Contenido

<b>DEFINICIÓN DEL MÓDULO .....</b>	<b>5</b>
<b>PRESENTACIÓN .....</b>	<b>5</b>
<b>INTRODUCCIÓN .....</b>	<b>7</b>
<b>LENGUAJES DE PROGRAMACIÓN .....</b>	<b>9</b>
<b>TIPOS DE LENGUAJES DE PROGRAMACIÓN .....</b>	<b>14</b>
NIVEL DE ABSTRACCIÓN DEL PROCESADOR .....	14
PARADIGMA DE PROGRAMACIÓN.....	16
FORMA DE EJECUCIÓN .....	16
<b>¿QUÉ SON LOS PARADIGMAS? .....</b>	<b>18</b>
PARADIGMAS DE PROGRAMACIÓN .....	18
CLASIFICACIÓN DE PARADIGMAS DE PROGRAMACIÓN .....	19
PARADIGMA IMPERATIVO .....	19
PARADIGMA DECLARATIVO .....	19
PARADIGMA ESTRUCTURADO .....	19
PARADIGMA FUNCIONAL .....	19
PARADIGMA LÓGICO .....	20
PARADIGMA ORIENTADO A OBJETOS.....	20
<b>DESARROLLO DE SOFTWARE CON EL PARADIGMA ORIENTADO A OBJETOS.....</b>	<b>21</b>
<b>¿QUÉ ES UNA CLASE? .....</b>	<b>21</b>
INTERFAZ E IMPLEMENTACIÓN .....	21
<b>¿QUÉ ES UN OBJETO? .....</b>	<b>22</b>
ESTADO .....	23
COMPORTAMIENTO .....	24
IDENTIDAD .....	25
<b>RELACIONES ENTRE OBJETOS .....</b>	<b>25</b>
<b>ENLACES.....</b>	<b>25</b>
VISIBILIDAD .....	26
<b>AGREGACIÓN .....</b>	<b>26</b>
<b>RELACIONES ENTRE CLASES .....</b>	<b>26</b>
ASOCIACIÓN.....	27
HERENCIA .....	27
EL ORNITORRINCO, QUE DERIVA DE MAMÍFERO Y OVÍPARO. LAS DOS CLASES DERIVAN DE ANIMAL.....	29
AGREGACIÓN.....	29
<b>EL MODELO DE OBJETOS.....</b>	<b>30</b>
ABSTRACCIÓN.....	30
ENCAPSULAMIENTO.....	32
MODULARIDAD.....	33
JERARQUÍA.....	35
TIPOS (TIPIFICACIÓN).....	37

<b>CONCURRENCIA .....</b>	<b>38</b>
PERSISTENCIA .....	39
CLASIFICACIÓN.....	40
<b><u>EL MODELADO EN EL DESARROLLO DE SOFTWARE .....</u></b>	<b><u>44</u></b>
¿QUÉ ES UN MODELO? .....	44
LA IMPORTANCIA DE MODELAR .....	44
PRINCIPIOS DE MODELADO .....	45
ADVERTENCIAS EN EL MODELADO .....	45
<b><u>LENGUAJE DE MODELADO UNIFICADO (UML) .....</u></b>	<b><u>46</u></b>
CONCEPTOS BÁSICOS SOBRE UML .....	46
BREVE RESEÑA HISTÓRICA.....	46
¿QUÉ ES LA OMG? .....	46
PRESENTACIÓN DE LOS DIAGRAMAS DE UML.....	47
<b>DIAGRAMA DE CLASES .....</b>	<b>49</b>
COMPONENTES DEL DIAGRAMA DE CLASES.....	49
<b>DIAGRAMA DE MÁQUINA DE ESTADOS .....</b>	<b>57</b>
COMPONENTES DEL DIAGRAMA DE MÁQUINA DE ESTADO .....	57
<b>DIAGRAMA DE SECUENCIA.....</b>	<b>62</b>
COMPONENTES DEL DIAGRAMA DE SECUENCIA .....	63
<b><u>REFERENCIA DE FIGURAS Y TABLAS.....</u></b>	<b><u>69</u></b>
FIGURAS .....	69
TABLAS .....	70
<b><u>FUENTES DE INFORMACIÓN .....</u></b>	<b><u>71</u></b>

## 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 con 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.

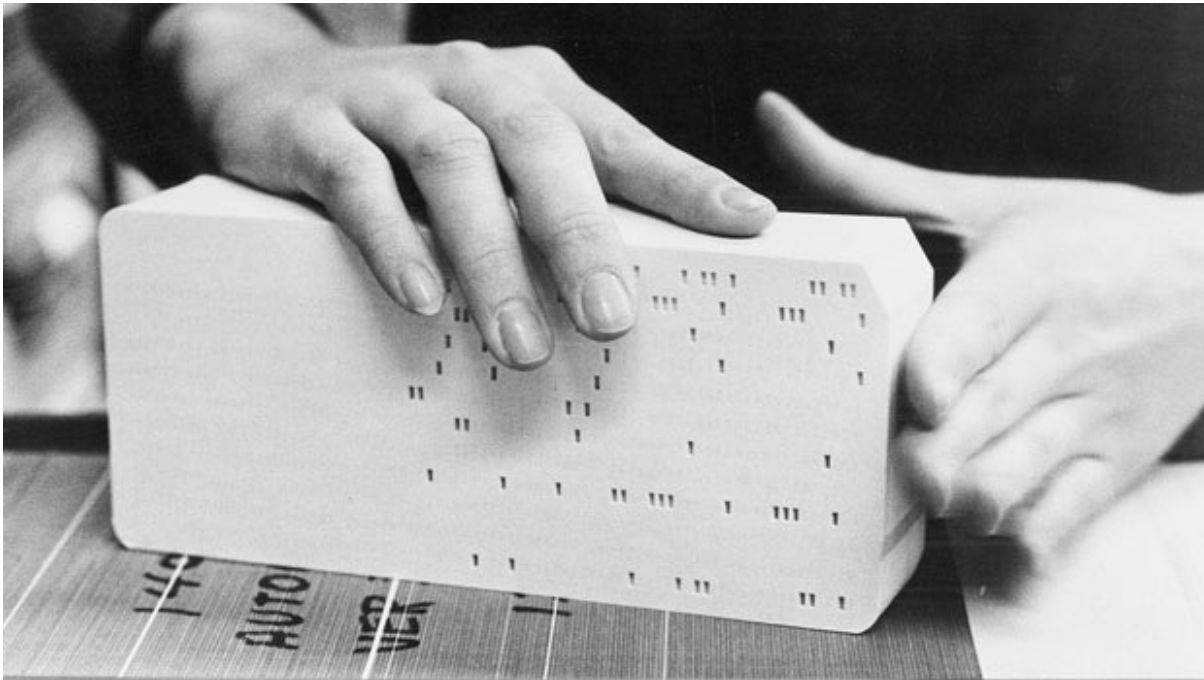
Para el caso del presente módulo las prácticas formativas profesionalizantes y los objetivos de aprendizajes se organizan para el desarrollo de:

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 binario <sup>1</sup> 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

---

<sup>1</sup> 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*.



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 ordenador <sup>2</sup> 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 diferentes <sup>3</sup>. 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.

<sup>2</sup> En el contexto de este apunte, computadora y ordenador son sinónimos.

<sup>3</sup> [https://es.wikipedia.org/wiki/Anexo%3ALenguajes\\_de\\_programaci%C3%B3n](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”)



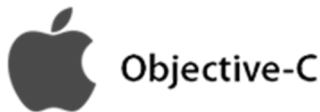
Lenguaje multiparadigma. Una extensión de C con mejoras como clases, funciones virtuales y plantillas.

**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.

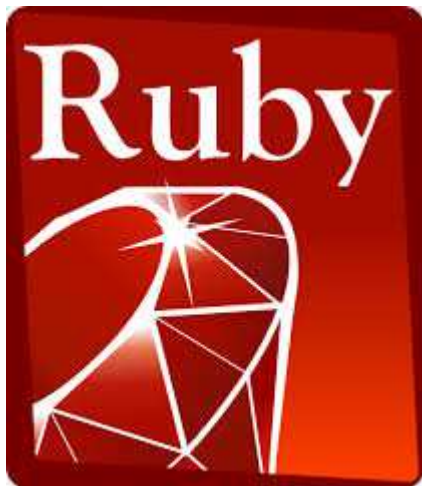
**Creador: Guido Van Rossum (CWI)**

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.

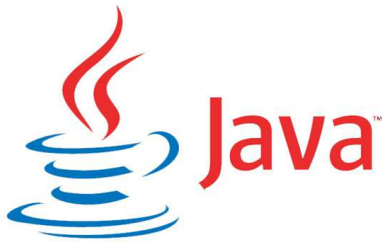
**Creador: Yukihiro Matsumoto**

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

Usado por: Twitter, Hulu, Groupon.

## 1995

- Java (inspirado en las tazas de café consumidas mientras se desarrollaba el lenguaje).



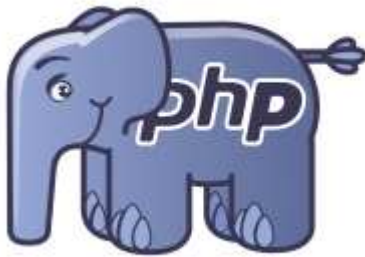
Lenguaje de propósito general, de alto nivel. Hecho para un proyecto de televisión interactiva. Funcionalidad de programación multiplataforma. Es actualmente el lenguaje de programación más popular en el mundo<sup>4</sup>.

**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



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.

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

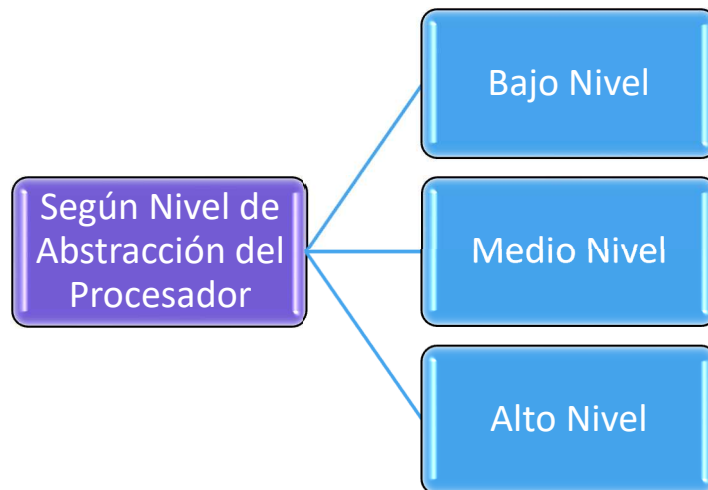
## Tipos de Lenguajes de Programación

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.

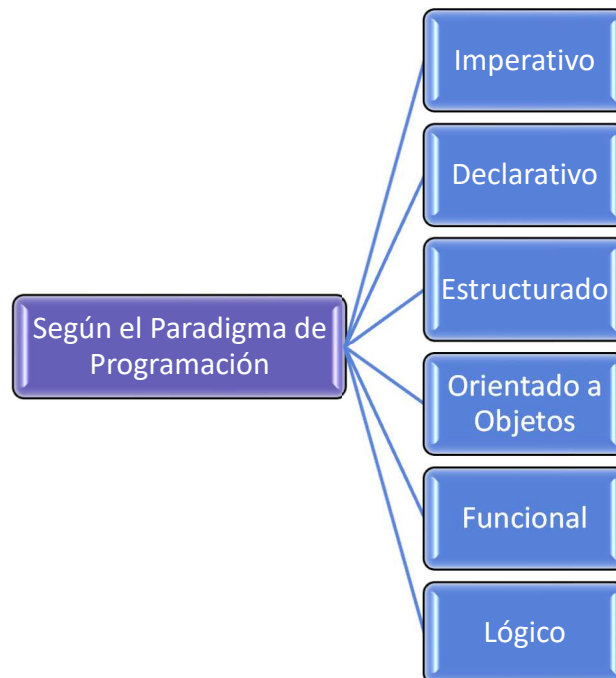
Una característica distintiva, por ejemplo, que convierte a C en un lenguaje de medio nivel es que es posible manejar las letras como si fueran números.

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

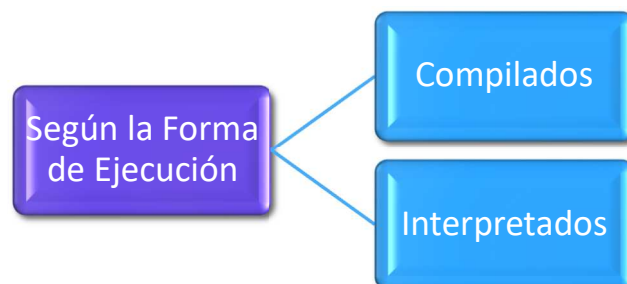
Según el Paradigma de Programación, los lenguajes se clasifican en:



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 comprensión 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 abstracción<sup>5</sup>, 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.

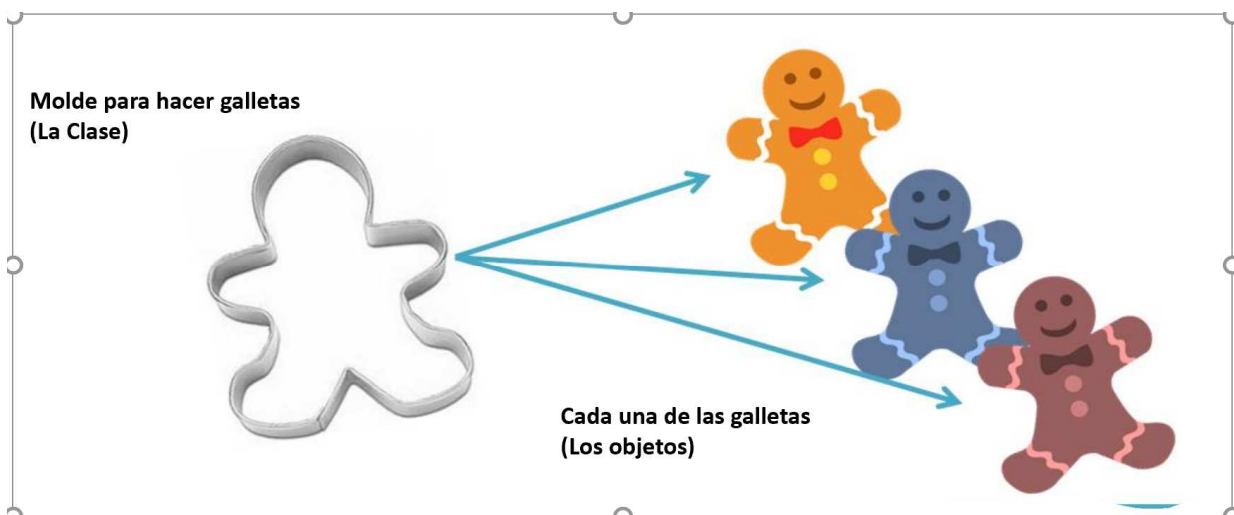


Fig. 3 - Clases y Objetos

### 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.

<sup>5</sup> **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 subclasses 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.

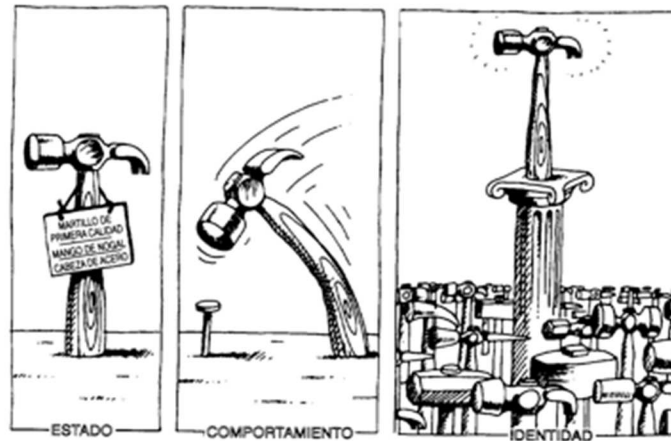


Fig. 4. Estado, comportamiento e identidad de un objeto

“ 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 cualidad 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 diseños 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 es 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:

<b>Modificador:</b>	Una operación que altera el estado de un objeto
<b>Selector:</b>	Una operación que accede al estado del objeto, pero no lo altera.
<b>Iterador:</b>	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:	
<b>Constructor:</b>	Una operación crea un objeto y/o inicializa su estado.
<b>Destructor:</b>	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 RAM<sup>6</sup>, 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.

<sup>6</sup> **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 refinarán 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 reutilización<sup>7</sup>.

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

<sup>7</sup> **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.

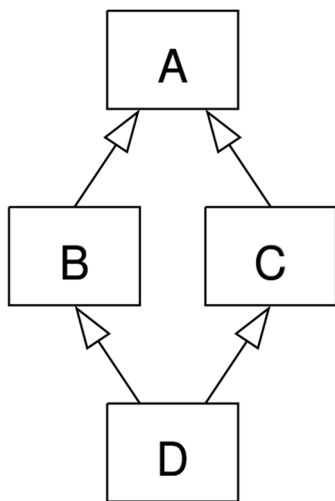


Fig. 6 Representación de la herencia de diamante

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: sí y solo sí 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.”

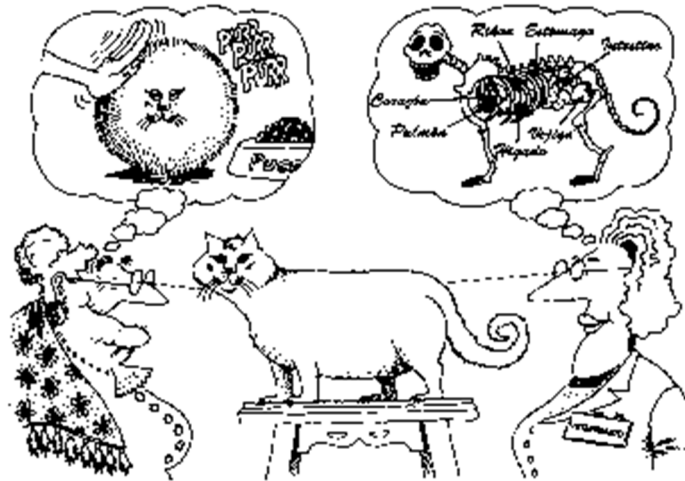


Fig. 7 - Abstracción en el Modelo de Objetos

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.



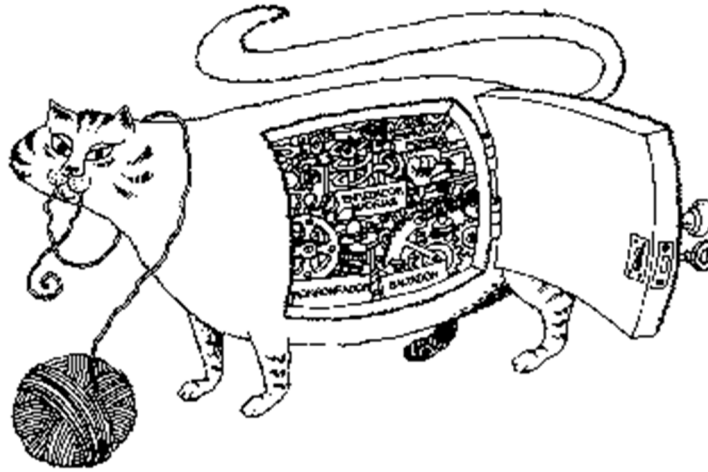


Fig. 9 - El encapsulamiento en el Modelo de Objetos

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 comprensión 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.

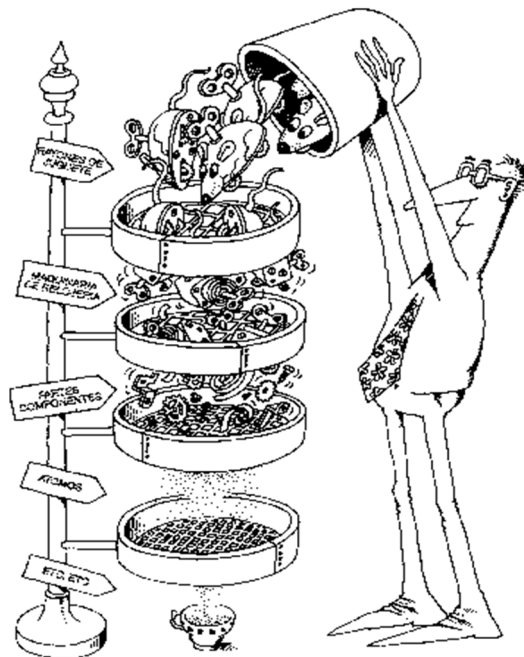


Fig. 11 - Las Abstracciones forman una Jerarquía

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 recompilaciones 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.

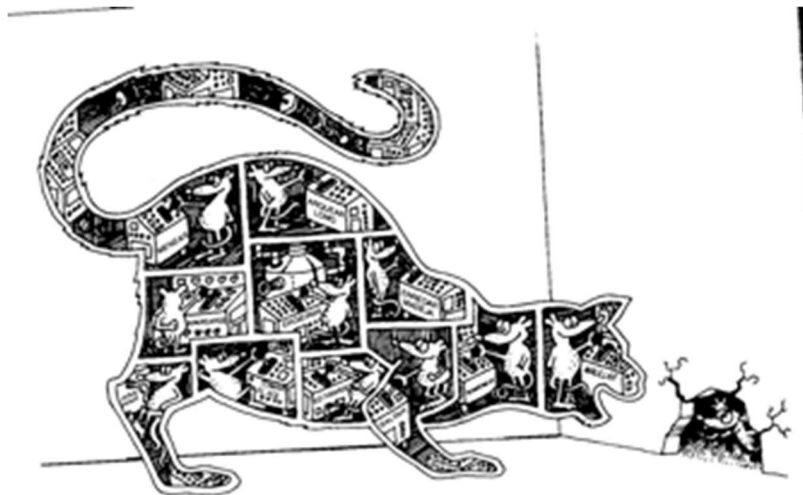


Fig. 13 - La concurrencia permite a dos objetos actuar al mismo tiempo

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.

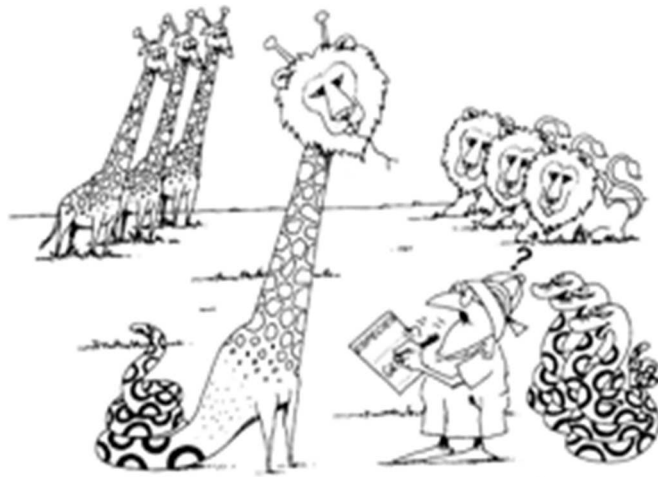


Fig.15- La clasificación es el medio por el cual ordenamos el conocimiento

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.

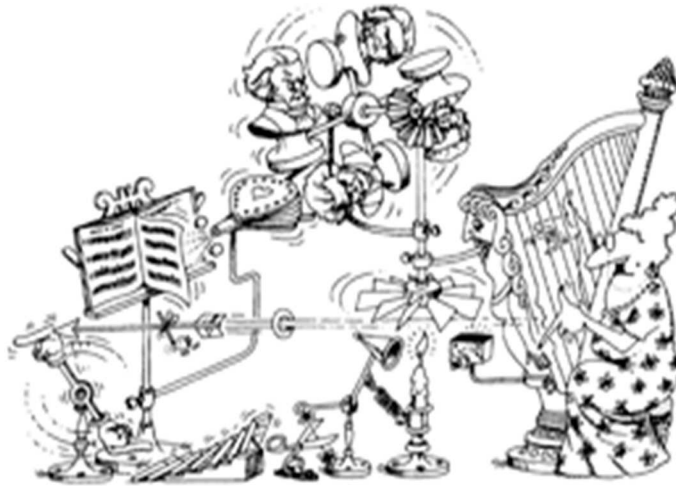
*Tabla 1 – Ejemplo de Clasificación de clases del dominio del problema*

### 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 satisfaga 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 descrito 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
<b>Diagrama de Estructura</b>  <i>Estos diagramas modelan estructura del sistema del sistema, por consiguiente, son diagramas estáticos</i>	Diagrama de Clases	Muestra una colección de elementos de modelado, tales como clases, tipos y sus contenidos y relaciones.
	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.

<b>Diagramas de Comportamiento</b>  <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 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.
	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.
<b>Diagramas de Interacción.</b>  <i>Tipo especial de diagramas de comportamiento que modelan interacción entre objetos, por consecuencia modelan el comportamiento dinámico del sistema.</i>	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.
	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 Objetos con Diagrama de Secuencia
- Modelado de estados de una clase con Diagrama de Máquina de Estados



## 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.

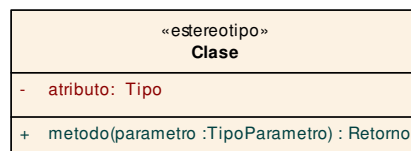


Fig.18- Notación para una clase en UML

- **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

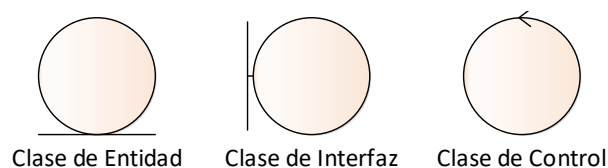


Fig. 19- Representación icónica de los tres tipos de clases de UML

### 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: **PantallaAdmPelícula**.

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 **GestorPelícula**, 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 descrito 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.

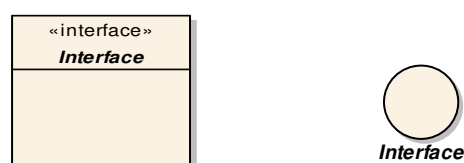


Fig. 20- Notaciones para una Interface en UML, representación etiquetada e icónica

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.

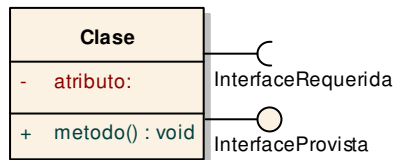


Fig. 21- Visualización de una clase con interfaces asociadas

## 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.

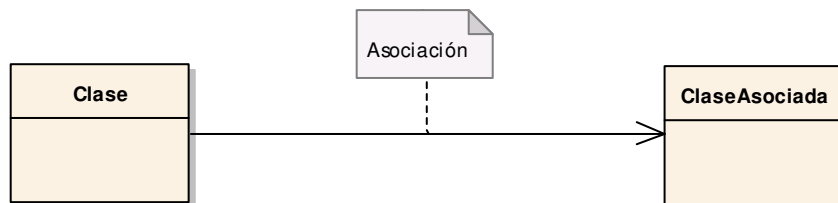


Fig. 22- Notación para representar la relación de asociación entre clases en UML

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



Fig. 23- 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.

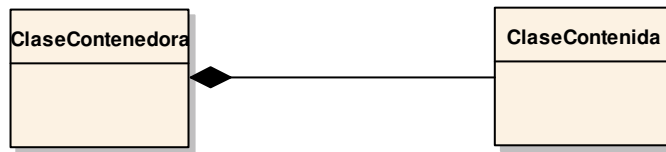


Fig. 24- Notación para representar la relación de composición entre clases en UML

➤ **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.

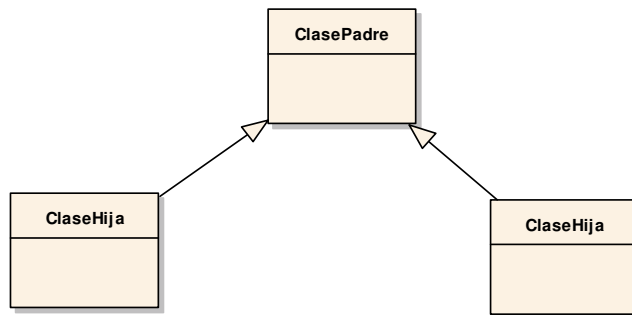


Fig. 25. Notación para representar la relación de generalización entre clases en UML, herencia simple

- **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.

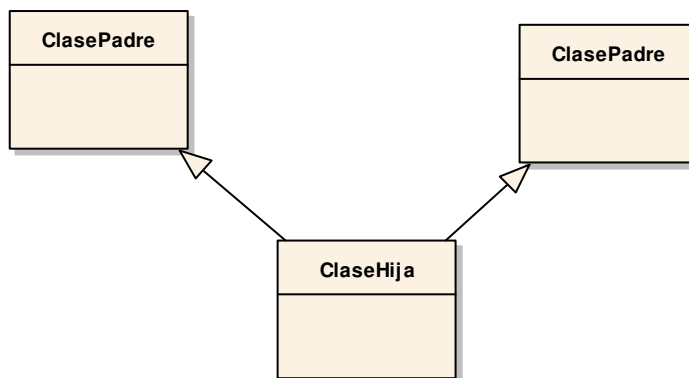


Fig. 26- Notación para representar la relación de generalización entre clases en UML, herencia múltiple

- 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.

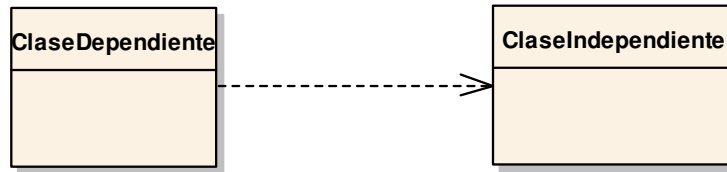


Fig. 27- Notación para representar la relación de dependencia entre clases en UML

- 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.

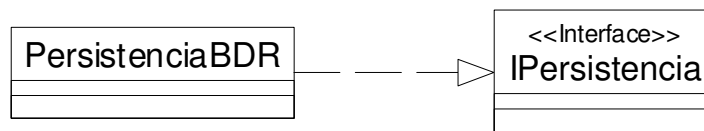


Fig. 28. Notación para representar la relación de realización entre clases en UML

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



Fig. 29- 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, refinar 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.

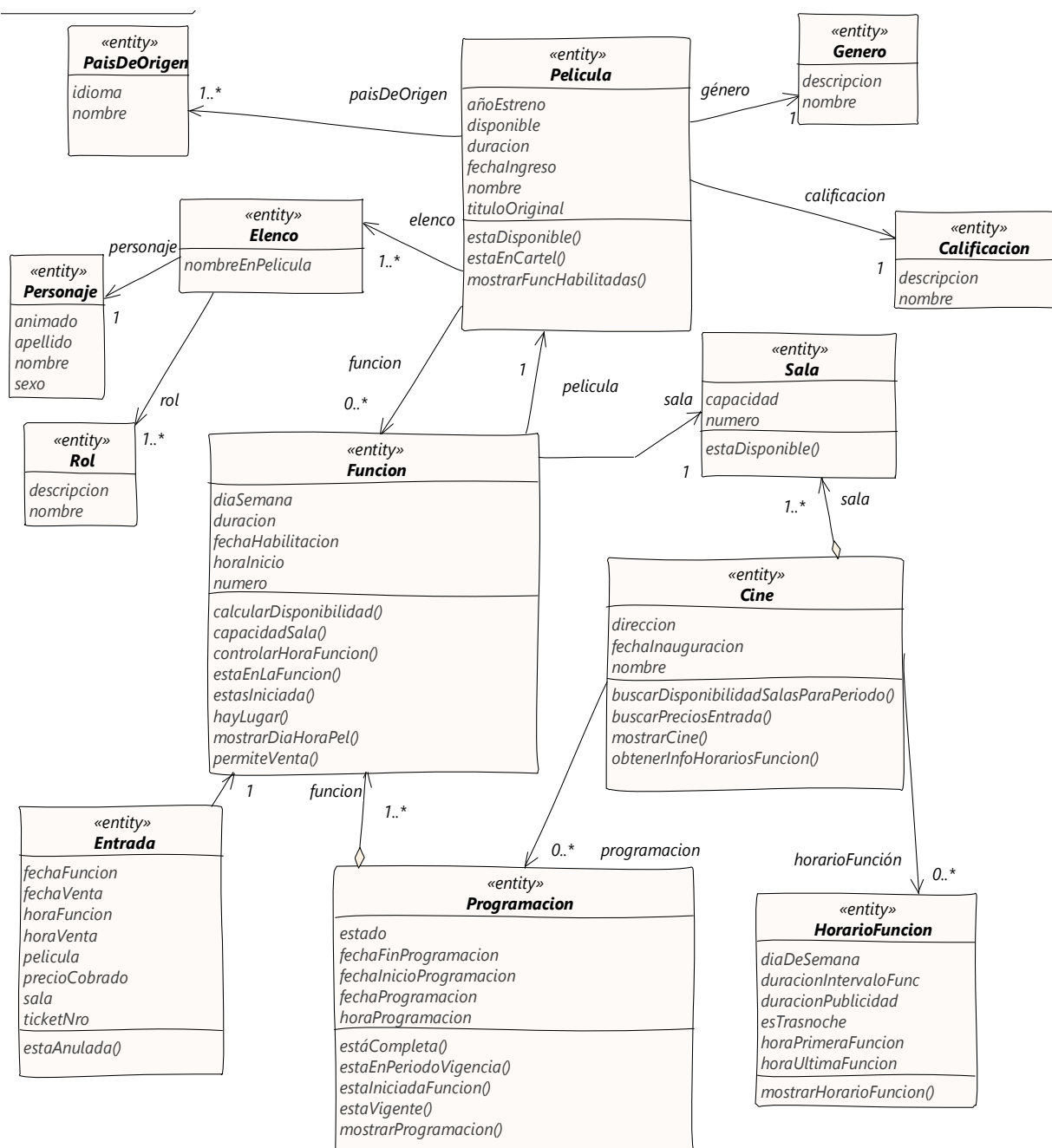


Fig. 30- Vista parcial del Modelo de Dominio para el caso de estudio del Complejo de Cines.

## 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. 31-. 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 círculo negro relleno. El **estado final**, es representado como un círculo negro relleno con un círculo de línea 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.

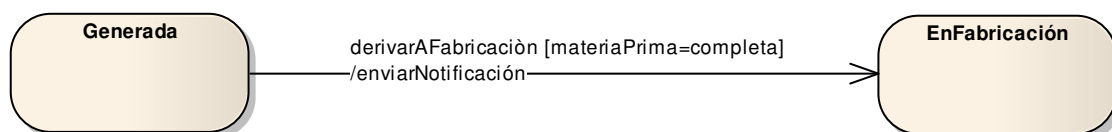


Fig. 32- Transición entre estados en un Diagrama de Máquina de Estados.

¿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:

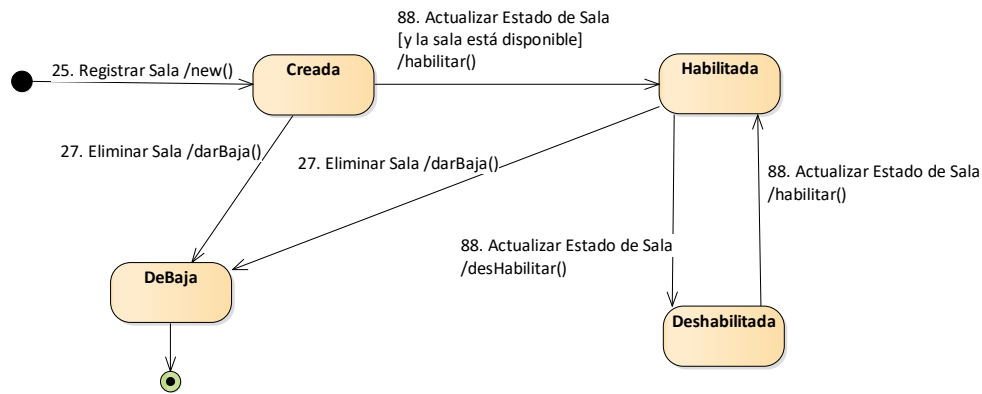


Fig. 33-. Diagrama de Máquina de estados para la clase Sala, utilizada en el caso del Complejo de Cines

Para ejemplificar 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”.

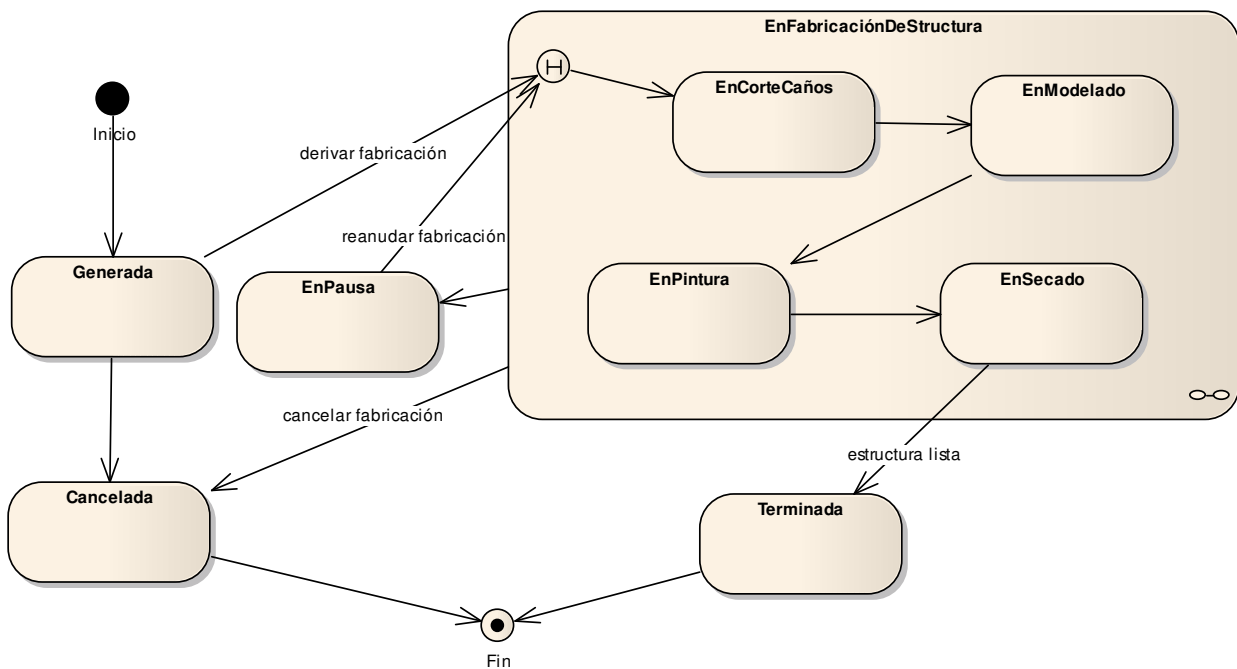


Fig. 34- Diagrama de Máquina de estados que muestra el uso de una sub-máquina y historia para los estados

## 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.

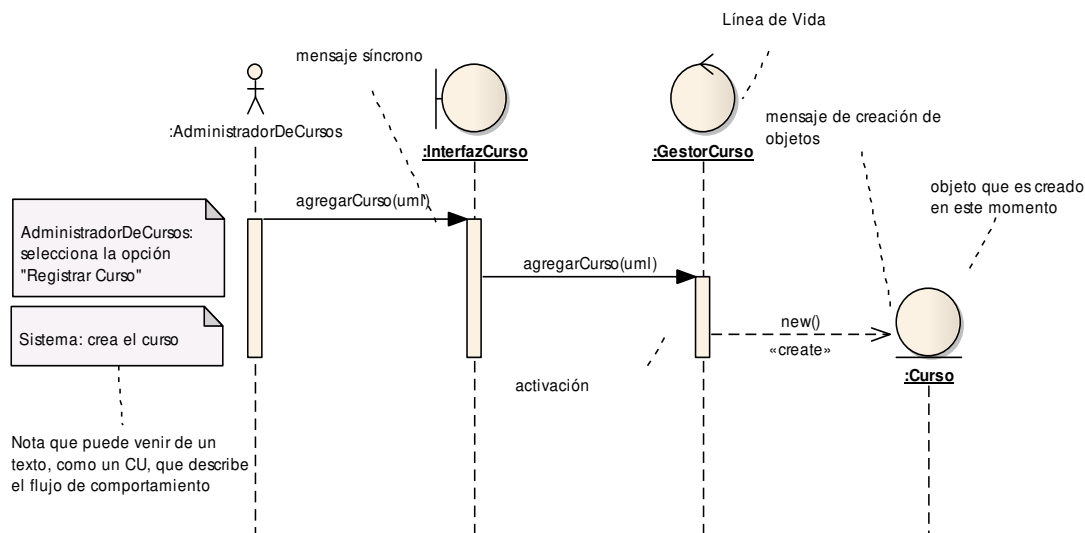


Fig. 35-. Elementos de modelado utilizados en un Diagrama de Secuencia

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 diagrama 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 icono representativo del tipo y además poseen una línea vertical punteada cuando se modelan en un Diagrama de Secuencia.

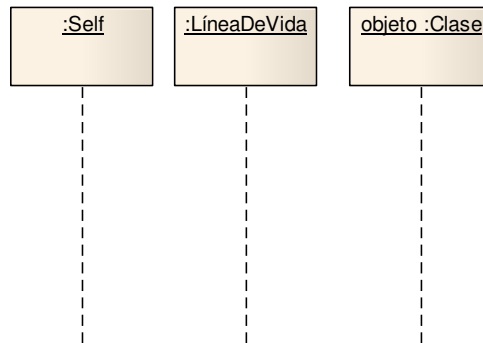


Fig. 36- Líneas de vida de los objetos en un Diagrama de Secuencia

**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.

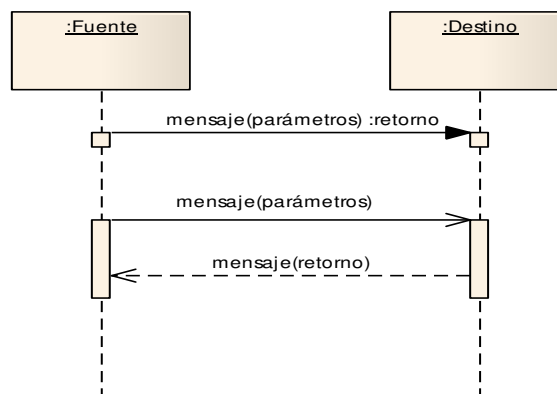


Fig. 37- Tipos de mensajes que pueden utilizarse en un Diagrama de Secuencia

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

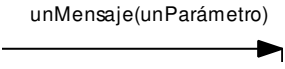
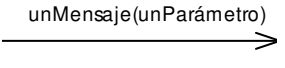
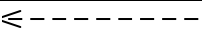
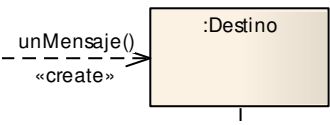
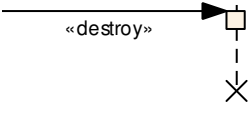
Sintaxis	Nombre	Descripción
	<b>Mensaje Síncrono</b>	El emisor espera que el receptor finalice la ejecución del mensaje.
	<b>Mensaje Asíncrono</b>	El emisor envía el mensaje y continúa ejecutando –éste no espera que el retorno del receptor.
	<b>Mensaje de Retorno</b>	El receptor de un mensaje enviado retorna el foco de control al emisor.
	<b>Creación de Objeto</b>	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».
	<b>Destrucción de Objeto</b>	El emisor destruye la instancia del receptor. Este mensaje no retorna valor.

Tabla 3 – 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 colocarían 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.

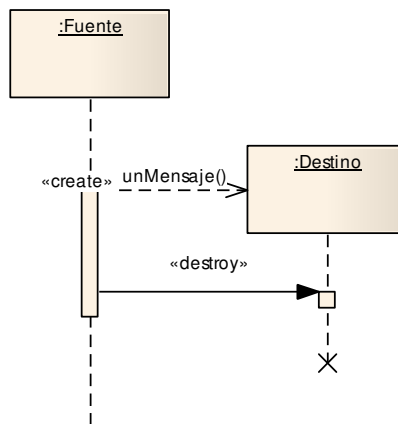


Fig. 38- Creación y destrucción de objetos en un Diagrama de Secuencia

### **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.

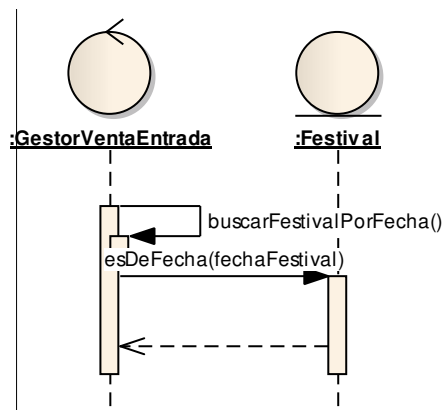


Fig. 39- Parámetros en un Diagrama de Secuencia

## 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 referencien 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, descrito anteriormente, para el caso de uso Registrar Película del Complejo de Cines.

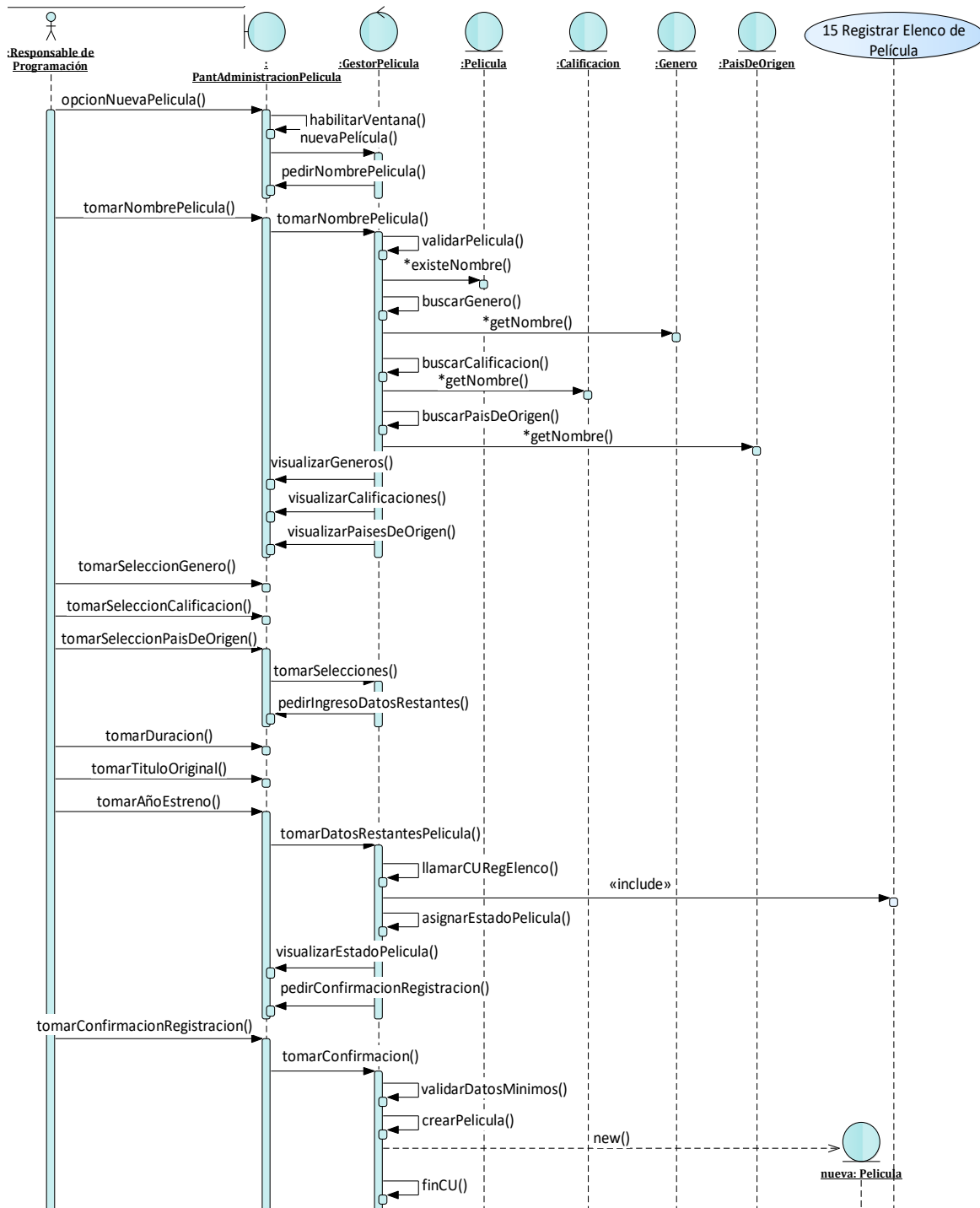


Fig. 40-. Diagrama de secuencia para un escenario de Registrar Película

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

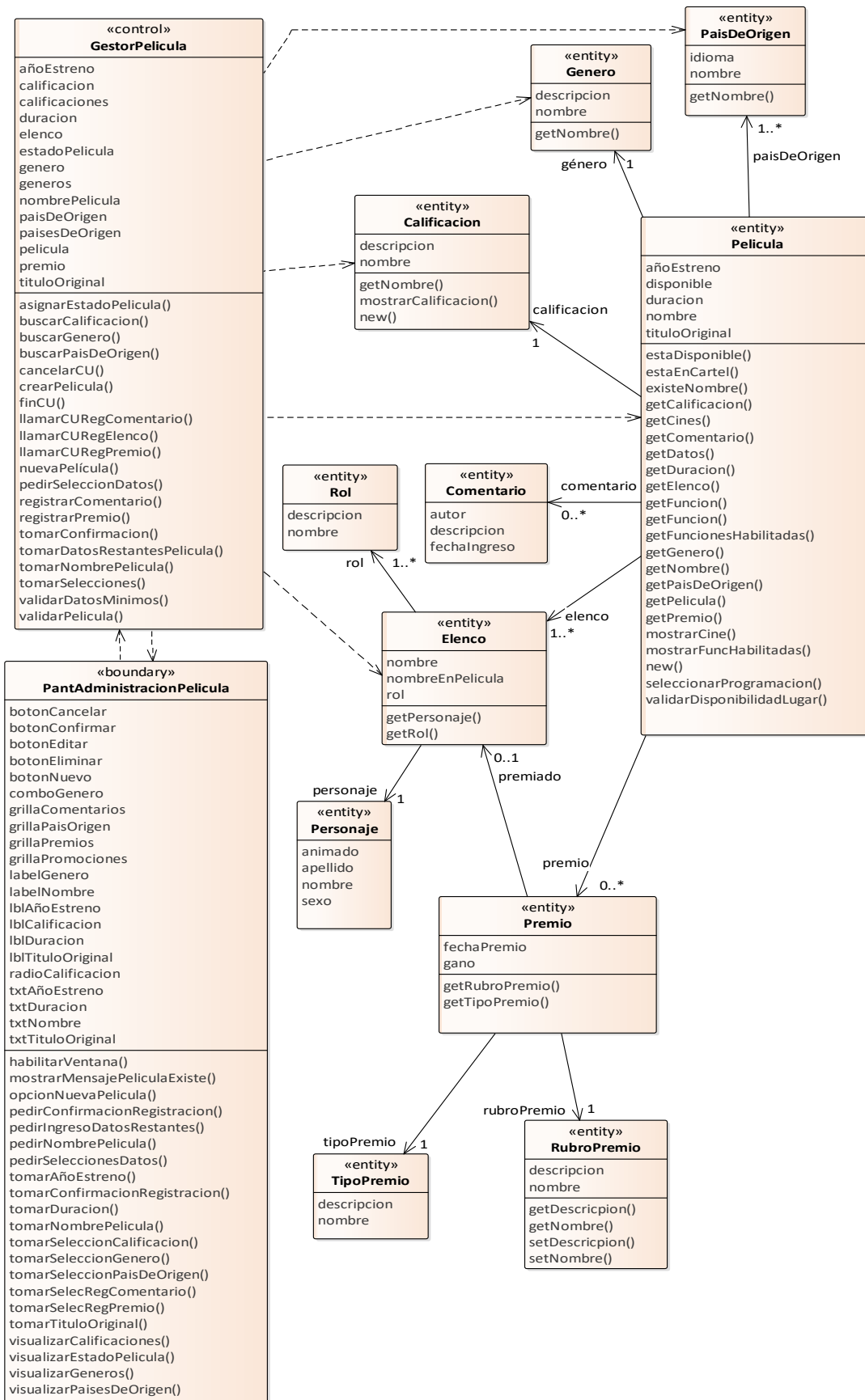


Fig. 41: Vista del Diagrama de clases que muestra las clases necesarias para modelar el escenario del curso normal del caso de uso Registrar Película

## Referencia de Figuras y Tablas

### Figuras

Fig. 1 – Tarjetas perforadas para programación de computadoras .....	7
Fig. 2 – Instrucciones en Assembly en la vista de Terminator (1984) .....	8
Fig. 3 - Clases y Objetos .....	21
Fig. 4. Estado, comportamiento e identidad de un objeto .....	23
Fig. 5 - Herencia entre Clases .....	28
Fig. 6 Representación de la herencia de diamante .....	29
Fig. 7 - Abstracción en el Modelo de Objetos.....	31
Fig. 8- Las clases y objetos deberían estar al nivel de abstracción adecuado: ni demasiado alto ni demasiado bajo .....	32
Fig. 9 - El encapsulamiento en el Modelo de Objetos .....	33
Fig. 10 - Modularidad, cohesión y acoplamiento en el Modelo de Objetos .....	34
Fig. 11 - Las Abstracciones forman una Jerarquía .....	36
Fig. 12- La comprobación estricta de tipos impide que se mezclen abstracciones .....	37
Fig. 13 - La concurrencia permite a dos objetos actuar al mismo tiempo .....	38
Fig. 14 - Conserva el estado de un objeto en el tiempo y el espacio .....	39
Fig.15- La clasificación es el medio por el cual ordenamos el conocimiento.....	40
Fig. 16 - Diferentes observadores pueden clasificar el mismo objeto de distintas formas .....	41
Fig. 17 - Los mecanismos son los medios por los cuales los objetos colaboran para proporcionar algún comportamiento de nivel superior.....	43
Fig.18- Notación para una clase en UML.....	49
Fig. 19- Representación icónica de los tres tipos de clases de UML .....	49
Fig. 20- Notaciones para una Interface en UML, representación etiquetada e icónica.....	51
Fig. 21- Visualización de una clase con interfaces asociadas .....	52
Fig. 22- Notación para representar la relación de asociación entre clases en UML .....	52
Fig. 23- Notación para representar la relación de agregación entre clases en UML .....	52
Fig. 24- Notación para representar la relación de composición entre clases en UML.....	53
Fig. 25. Notación para representar la relación de generalización entre clases en UML, herencia simple .....	53
Fig. 26- Notación para representar la relación de generalización entre clases en UML, herencia múltiple .....	53
Fig. 27- Notación para representar la relación de dependencia entre clases en UML .....	54
Fig. 28. Notación para representar la relación de realización entre clases en UML.....	54



Fig. 29- Notación para representar la relación de contención entre clases en UML .....	54
Fig. 30- Vista parcial del Modelo de Dominio para el caso de estudio del Complejo de Cines.....	56
Fig. 31-. Notaciones para representar estados en UML .....	57
Fig. 32- Transición entre estados en un Diagrama de Máquina de Estados. ....	58
Fig. 33-. Diagrama de Máquina de estados para la clase Sala, utilizada en el caso del Complejo de Cines .....	61
Fig. 34- Diagrama de Máquina de estados que muestra el uso de una sub-máquina y historia para los estados.....	61
Fig. 35-. Elementos de modelado utilizados en un Diagrama de Secuencia .....	62
Fig. 36- Líneas de vida de los objetos en un Diagrama de Secuencia.....	63
Fig. 37- Tipos de mensajes que pueden utilizarse en un Diagrama de Secuencia .....	63
Fig. 38- Creación y destrucción de objetos en un Diagrama de Secuencia .....	65
Fig. 39- Parámetros en un Diagrama de Secuencia .....	65
Fig. 40-. Diagrama de secuencia para un escenario de Registrar Película .....	67
Fig. 41: Vista del Diagrama de clases que muestra las clases necesarias para modelar el escenario del curso normal del caso de uso Registrar Película .....	68

## Tablas

Tabla 1 – Ejemplo de Clasificación de clases del dominio del problema .....	42
Tabla 2 – Diagramas de UML 2.0 .....	48
Tabla 3 – Tipos de Mensajes que pueden utilizarse para modelar diagramas de secuencia.....	64

## Fuentes de Información

---

- **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 1ª 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).