La documentación en el desarrollo del software

El propósito de aplicar metodologías en el desarrollo de software es lograr procesos predecibles y repetibles que mejoren la productividad y la calidad.

Para que los procesos puedan ser repetibles, uno de los requisitos es que sean debidamente modelados y documentados. Esto cobra relevancia sobretodo si el equipo de desarrollo cambia, es muy grande, está disperso geográficamente o si los módulos pasan muchas veces de unas personas a otras.

Frecuentemente las compañías de desarrollo se encuentran con problemas por la gran rotación de personal. Esto acarrea serios inconvenientes en la atención de sus clientes en el periodo indicado y cumpliendo con el propósito del producto, porque en el transcurso del proyecto uno de los miembros del equipo deja la compañía, llevando consigo el conocimiento de las reglas comerciales y de la especificación del software. El tiempo y los costos para la recuperación de ese conocimiento pueden ser muy altos dependiendo de la fase en que el proyecto se encuentra.

Por otra parte, es un hecho que la complejidad siempre creciente del software dificulta su desarrollo. Para manejar la complejidad que se encuentra hoy en día en las aplicaciones, los arquitectos de software y desarrolladores dentro de un equipo de desarrollo, son los responsables de especificar, documentar y mantener todos los aspectos de la arquitectura del software de la aplicación. Por lo tanto necesitan instrumentos para llevar a cabo esta tarea.

La documentación no es exactamente una fase del desarrollo del software, sino una actividad que debe practicarse a lo largo de todo el desarrollo.

La documentación que debe haberse generado al terminar un producto software es de dos tipos:

- La **documentación externa:** la conforman todos los documentos ajenos al programa, como ser especificaciones de requisitos, diseños, guías de instalación, guías de usuario, etc.
- La **documentación interna**: es la integrada al programa (básicamente, los comentarios).

Documentación externa

Para producir software que cumpla las necesidades y expectativas del cliente hay que obtener los requerimientos del sistema. Esto se consigue conociendo de una forma disciplinada a los usuarios y haciéndolos participar de manera activa para que no queden cabos sueltos. Para conseguir un software de calidad, que sea duradero y fácil de mantener hay que idear una sólida base arquitectónica que sea flexible al cambio. Para desarrollar software rápida y eficientemente, minimizando el trabajo de recodificación y evitando crear miles de líneas de código inútil hay que disponer, además de la gente y las herramientas necesarias, de un enfoque apropiado.

A la hora de desarrollar software industrial, para conseguir un producto de calidad, es completamente necesario seguir ciertas pautas y no abordar los problemas de manera superficial, con el fin de obtener un modelo que represente lo suficientemente bien el problema que se abordará.

Modelado

El modelado es la espina dorsal del desarrollo de software de calidad. **El modelado es la construcción de un modelo a partir de una especificación.**

Se construyen modelos para:

- explicar el comportamiento del sistema a desarrollar
- la mejor comprensión del sistema por parte de los mismos desarrolladores
- facilitar la comunicación entre los participantes del proyecto
- facilitar la comunicación entre usuarios y desarrolladores
- controlar el riesgo
- atacar problemas que sin el modelado de su resolución seria imposible, tanto desde el punto de vista de los desarrolladores (no se pueden cumplir los plazos estimados, no se consigue ajustar los presupuestos...) como desde el punto de vista del cliente, el cual, cuando finalmente se le entrega el producto del desarrollo, generalmente se encuentra con infinidad de problemas, desde que no se cumplen las especificaciones hasta fallos que dejan inutilizado el sistema.

Al hacer referencia al desarrollo software en el ámbito industrial, se pretende que la capacidad de modelar no se reserve sólo a empresas que disponen de gran número de empleados o empresas que abordan proyectos de gran volumen. Lo que se pretende es la capacidad de obtener un producto comercial, sea cual fuere su costo o tamaño, que cumpla lo que en la industria se denomina **calidad total**[1] y que reporte beneficios a corto plazo, evitando excesivos períodos de desarrollo debido a la falta de previsión o por haber abordado los problemas muy a la ligera, sin planificación.

Modelo: definiciones

- Abstracción de la realidad; representación simplificada de algún objeto o fenómeno del mundo real
- Representación esquemática o conceptual de un fenómeno, que representa una teoría o hipótesis de cómo funciona dicho fenómeno
- Descripción o representación utilizada para ayudar a visualizar algo que no puede ser observado directamente. Una representación abstracta de un objeto o sistema, desde un punto de vista particular.
- Abstracción de algo, que se elabora para comprender ese algo antes de construirlo. El modelo omite detalles que no resultan esenciales para la comprensión del original y por lo tanto facilita dicha comprensión. Es un boceto o una guía de ese algo que vamos a construir para que a la hora de elaborarlo no lo hagamos erróneamente. Es útil también para mostrar a los interesados una aproximación de lo que se construirá. Esto les permite opinar acerca de lo que se modela, y también permite modificarlo si es necesario.

El modelo de un fenómeno es una herramienta que se usa para describirlo, interpretarlo, predecir comportamiento en diferentes situaciones específicas, validar hipótesis y elaborar estrategias para tratarlo.

Los modelos son utilizados en muchas actividades de la vida humana: antes de construir una casa el ingeniero realiza un plano, el arquitecto utiliza una maqueta, los músicos representan la música en forma de notas musicales, los artistas pintan sobre el lienzo con carboncillos antes de empezar a utilizar los óleos, etc. Unos y otros abstraen una realidad compleja sobre unos bocetos. Todos ellos son "modelos".

Los modelos permiten una mejor comunicación con el cliente por distintas razones:

- Es posible enseñar al cliente una posible aproximación de lo que será el producto final.
- Proporcionan una primera aproximación al problema, que permite visualizar cómo quedará el resultado.
- Reducen la complejidad del original en subconjuntos que son fácilmente tratables por separado.

En Informática, un modelo es una representación de conceptos del dominio del problema. Es una vista de un sistema del mundo real, es decir, una abstracción de dicho sistema considerando un cierto propósito. Así, el modelo describe completamente aquellos aspectos del sistema que son relevantes al propósito del modelo y a un apropiado nivel de detalle.

A través del modelado se consiguen cuatro objetivos:

- Los modelos ayudan a visualizar cómo será el sistema.
- Los modelos permiten especificar la estructura o el comportamiento de un sistema.
- Los modelos proporcionan plantillas que guían en la construcción del sistema.
- Los modelos documentan las decisiones adoptadas.

Todo modelo del sistema forma parte de la documentación del software y es muy útil en las distintas etapas de un proyecto de desarrollo, por lo cual se debe tratar de producir modelos lo más representativo posible a lo que se espera construir.

Lenguaje de modelado

En la industria del software se ha comprobado que un modelado orientado a objetos proporciona arquitecturas más flexibles y readaptables que otros, por ejemplo orientados a la funcionalidad o a los datos. Desde la aparición de esta tecnología, un gran número de metodólogos hicieron esfuerzos para obtener una

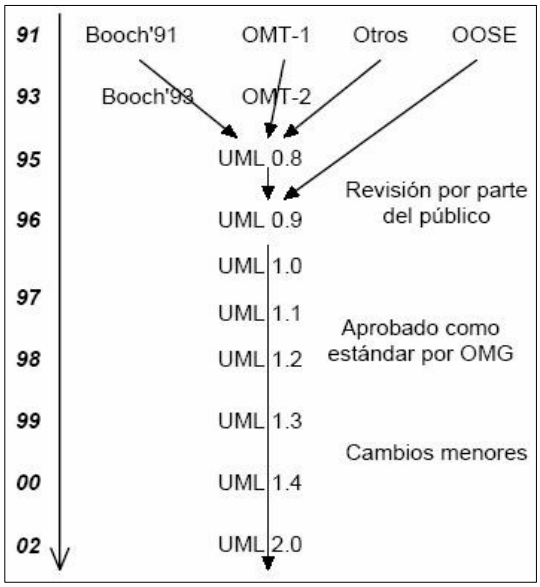
[1] Calidad total se refiere a niveles de calidad tan altos en todas las fases de la vida de un proyecto (ya sea mecánico, industrial, civil...) que no haya que estar modificando en las fases siguientes debido a errores que se debían haber detectado previamente.

notación para representar el modelado. Muchos usuarios de estos métodos tenían problemas al intentar encontrar un lenguaje de modelado que cubriera sus necesidades completamente, alimentando de esta forma la llamada *guerra de métodos*. Aprendiendo de estas experiencias, comenzaron a aparecer nuevas generaciones de métodos, entre los que se destacaron unos pocos de manera muy clara. Entre ellos:

- OOD (Object Oriented Design, Diseño Orientado a Objetos) de Booch
- OOSE (Object-Oriened Software Engineering, Ingeniería de Software Orientada a Objetos) de Jacobson
- OMT (Object Modeling Technique – Técnica de Modelado de Objetos) de Rumbaugh

Cada uno de éstos era un método completo, auque todos tenían sus puntos fuertes y debilidades.
En la primera mitad de los '90, Grady Booch (Rational Software Corporation), Ivan Jacobson (Objectory) y James Rumbaugh (General Electric), empezaron a adoptar ideas de los otros dos métodos, los cuales habían sido reconocidos en conjunto como los tres principales métodos OO a nivel mundial. Estos autores, conocidos como "los tres amigos", comenzaron a trabajar en forma conjunta para crear un lenguaje unificado de modelado, intentando que la industria software termine su maduración como *Ingeniería.*

Es así como en 1996 se publica la primera versión de UML (Unified Modeling Lenguaje, Lenguaje de Modelado Unificado), que luego de varias modificaciones, fue estandarizado por el OMG.



Lenguaje de Modelado Unificado - UML

La aparición de UML significó un gran avance en la ingeniería del software, ya que proporciona las herramientas necesarias para poder obtener los *planos del software*, equivalentes a los que se utilizan en la construcción, la mecánica o la industria aeroespacial.

El UML es una técnica de modelado de objetos y como tal supone una abstracción de un sistema para llegar a construirlo en términos concretos. El modelado no es más que la construcción de un modelo a partir de una especificación.

Es un lenguaje estándar, orientado a objetos, que permite visualizar, especificar, construir y documentar todos los artefactos que componen un sistema con gran cantidad de software.

Abarca todas las fases del ciclo de vida de un proyecto, soporta diferentes maneras de visualización dependiendo de quién tenga que interpretar los planos y en qué fase del proyecto se encuentre.

Es importante destacar que UML es solamente un "lenguaje" para especificar y no un método o un proceso. Es el lenguaje en el que está descrito el modelo. UML se puede usar en una gran variedad de formas para soportar una metodología de desarrollo de software (tal como el Proceso Unificado de Rational), pero no especifica en sí mismo qué metodología o proceso usar. Sin embargo, para que el proceso sea óptimo, debe usarse en un proceso dirigido por casos de uso, centrado en la arquitectura, iterativo e incremental.

UML es un lenguaje por que proporciona un vocabulario y las reglas para utilizarlo, además es un lenguaje de modelado, lo que significa que el vocabulario y las reglas se utilizan para la representación conceptual y física del sistema.

Este lenguaje también intenta solucionar el problema de propiedad de código que se da con los desarrolladores: al implementar un lenguaje de modelado común para todos los desarrollos se crea una documentación, también común, que cualquier desarrollador con conocimientos de UML será capaz de entender, independientemente del lenguaje utilizado para el desarrollo

Ayuda a interpretar grandes sistemas mediante gráficos o mediante texto, obteniendo modelos explícitos que ayudan a la comunicación durante el desarrollo ya que al ser estándar, los modelos podrán ser interpretados por personas que no participaron en su diseño (e incluso por herramientas) sin ninguna ambigüedad.

Debido a su estandarización y su definición completa no ambigua, y aunque no sea un lenguaje de programación, UML se puede conectar de manera directa a lenguajes de programación como Java, C++ o Visual Basic. Esta correspondencia permite lo que se denomina *ingeniería directa* (obtener el código fuente partiendo de los modelos) pero además es posible reconstruir un modelo en UML partiendo de la implementación, o sea, la *ingeniería inversa*.

UML brinda la capacidad de modelar actividades de planificación de proyectos y de sus versiones, expresar requisitos y las pruebas sobre el sistema, representar todos sus detalles así como la propia arquitectura. Mediante estas capacidades se obtiene una documentación que es valida durante todo el ciclo de vida de un proyecto.

UML proporciona distintos puntos de vista de la realidad que modela mediante los distintos tipos de diagramas que posee. Un diagrama es una representación gráfica de una colección de elementos del modelo, que habitualmente toma forma de grafo donde los arcos que conectan sus vértices son las relaciones entre los objetos y los vértices se corresponden con los elementos del modelo. Los distintos puntos de vista de un sistema real que se quieren representar para obtener el modelo se dibujan de forma que se resaltan los detalles necesarios para entender el sistema.

Los diagramas muestran diferentes aspectos de las entidades representadas. Están agrupados en dos tipos diferentes de diagramas: los que dan una vista estática del sistema y los que dan una visión dinámica.

Los diagramas estáticos son:

- Diagrama de casos de uso
- Diagrama de clases
- Diagrama de objetos.
- Diagrama de componentes
- Diagrama de despliegue

Lo diagramas dinámicos son:

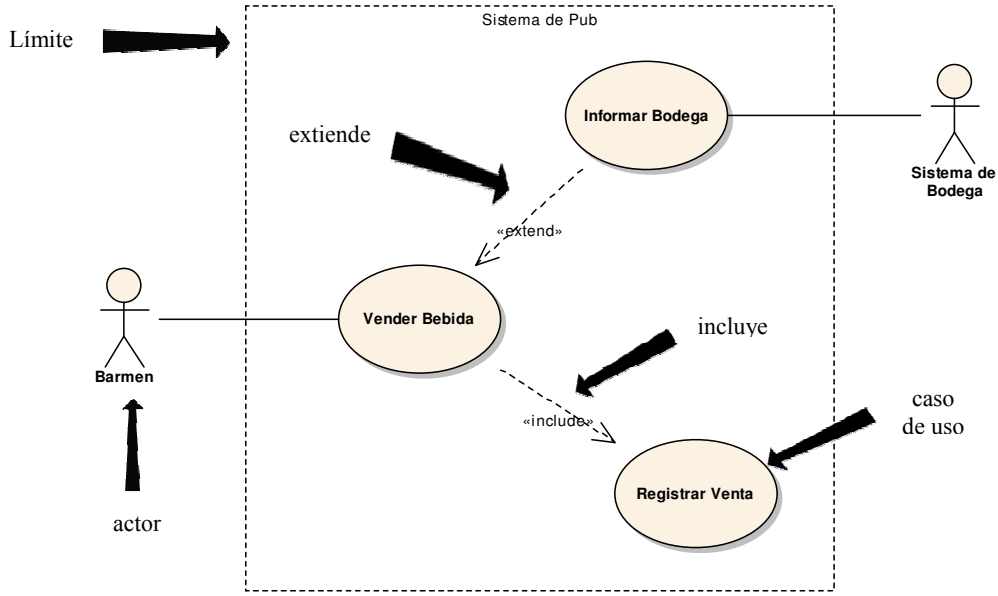
- Diagrama de secuencia
- Diagrama de colaboración
- Diagrama de estados
- Diagrama de actividades

A pesar de tener un número de diagramas muy alto, UML permite definir solo los necesarios, ya que no todos son imprescindibles en todos los proyectos.

1. Diagramas de Casos de Uso

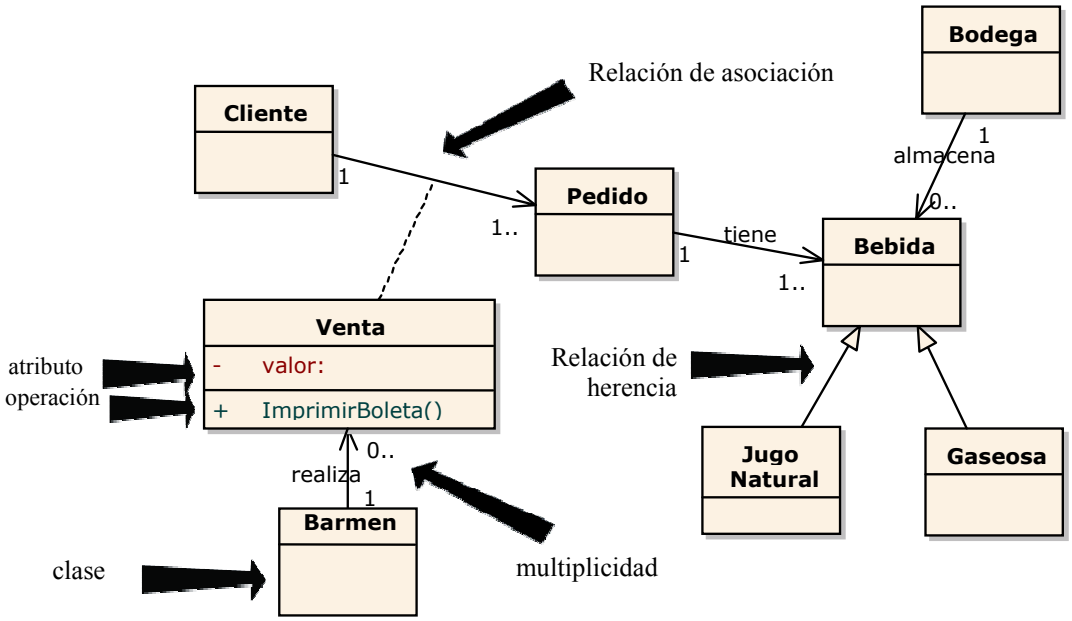
Muestra *quién* puede hacer *qué* y las *relaciones* existentes entre acciones (casos de uso).

- Usados Para Comunicarse con el Usuario Final y el Experto de Dominio
  - Proporciona credibilidad en una etapa inicial del desarrollo del sistema
  - Asegura una comprensión mutua de los requisitos
- Usados Para Identificar
  - Quién interactuará con el sistema y qué deberá hacer el sistema
  - Qué interfaz deberá tener el sistema
- Usados Para Verificar
  - Que se hayan capturado todos los requerimientos
  - Que los desarrolladores hayan entendido los requerimientos



2. Diagramas de Clases

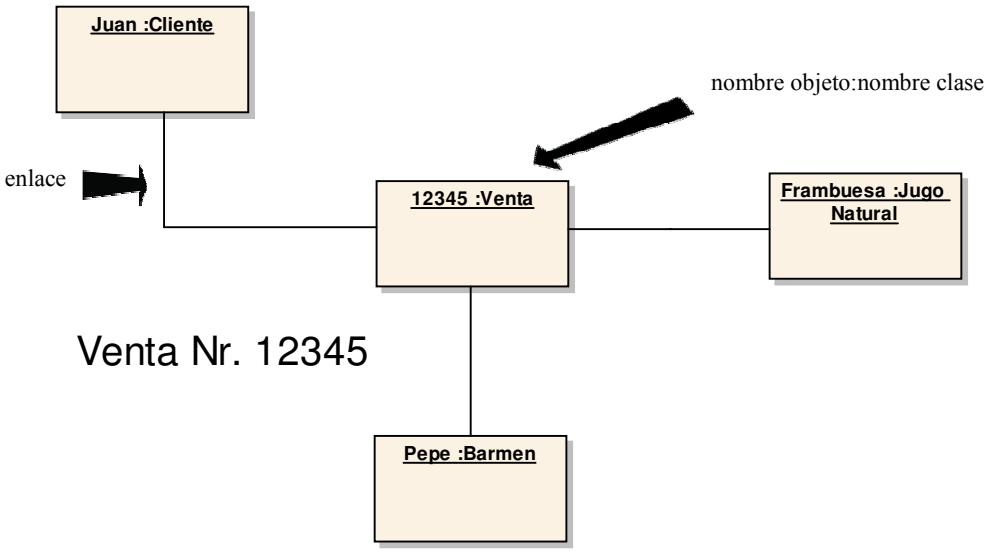
- Usados para mostrar la Estructura Estática de un sistema computacional o una parte relevante del mundo real. Dan una vista estática del proyecto.
- Son los diagramas más frecuentemente usados. Se los puede considerar con distintas perspectivas:
  - Conceptual: muestra las entidades del mundo real con sus relaciones
  - Implementación: el diseño para el código fuente. Representa la visibilidad de atributos y métodos: público (+), privado (-), protegido (#)



3. Diagramas de Objetos

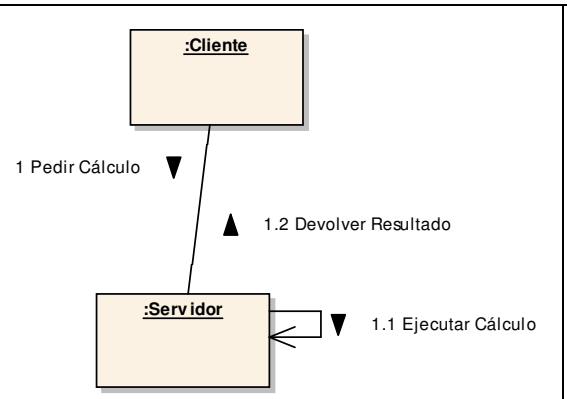
Es un diagrama de instancias de las clases mostradas en el diagrama de clases. Muestra las instancias y como se relacionan entre ellas. Se da una visión de casos reales.

- Usados para mostrar la estructura de objetos en tiempo de ejecución del sistema
- Representan vistas instantaneas (*snapshot*) de una parte del sistema de interés
- Destacan relaciones entre objetos
- Útiles para análisis y diseño preliminar e identificación de clases
- Usados para validar los modelos de clases



4. Diagramas de Colaboración

- Usados para representar el comportamiento del sistema
- Muestran colaboración entre los objetos del sistema
- Destacan:
  - Mensajes enviados entre los objetos
  - Enlaces entre los objetos
  - Un escenario concreto, sin condiciones
- Útiles tanto en análisis (identificación de clases), como en diseño (especificación de componentes)

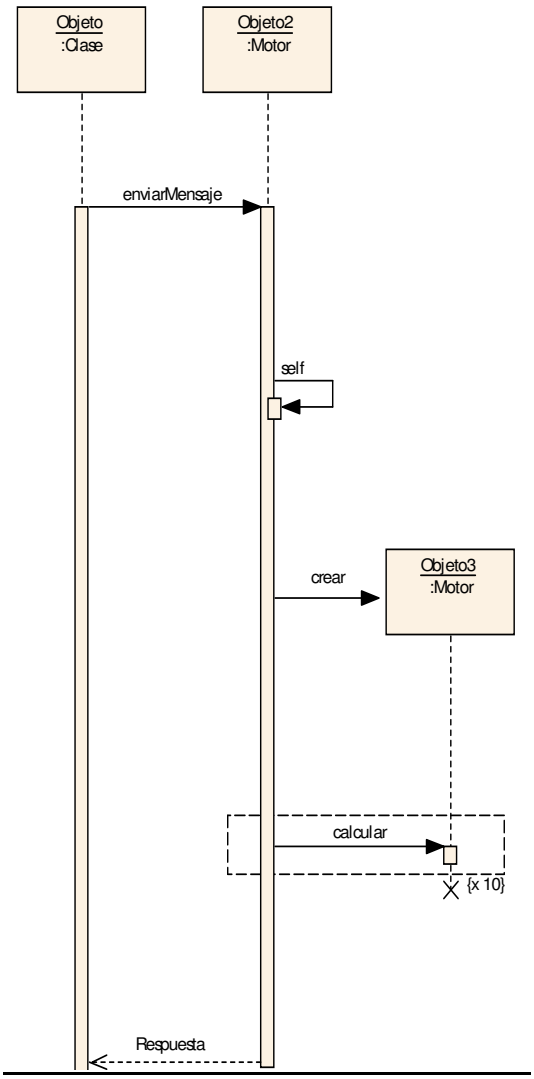


- Un Objeto representa una entidad de una clase en tiempo de ejecución
- Los enlaces conectan los objetos
- Un objeto envía un mensaje al otro El orden de los mensajes se indica con los números
- El mensaje puede ser enviado al mismo objeto

5. Diagramas de Secuencia

Muestran a los diferentes objetos y las relaciones que pueden tener entre ellos, a través de los mensajes que se envían entre ellos. Junto con el de colaboración, constituyen los diagramas de interacción. Son dos diagramas diferentes, que se puede pasar de uno a otro sin perdida de información, pero que dan puntos de vista diferentes del sistema.

- Usados para representar el comportamiento del sistema
- Muestran colaboración a través de mensajes entre los objetos del sistema
- Destacan:
  - Mensajes enviados entre los objetos
  - Orden secuencial entre los mensajes
  - Un escenario concreto, sin condiciones
- Útiles tanto en análisis (identificación de clases), como en diseño (especificación de componentes)



- Objeto representa una entidad de una clase en tiempo de ejecución
- Un objeto envía un mensaje al otro
- El mensaje puede ser enviado al mismo objeto
- Un objeto crea otro objeto
- Varias ocurrencias (iteraciones) de un mensaje
- El objeto muere
- Mensaje de respuesta

¿Secuencia o Colaboración ? ¿ Cuándo usarlos ¿?

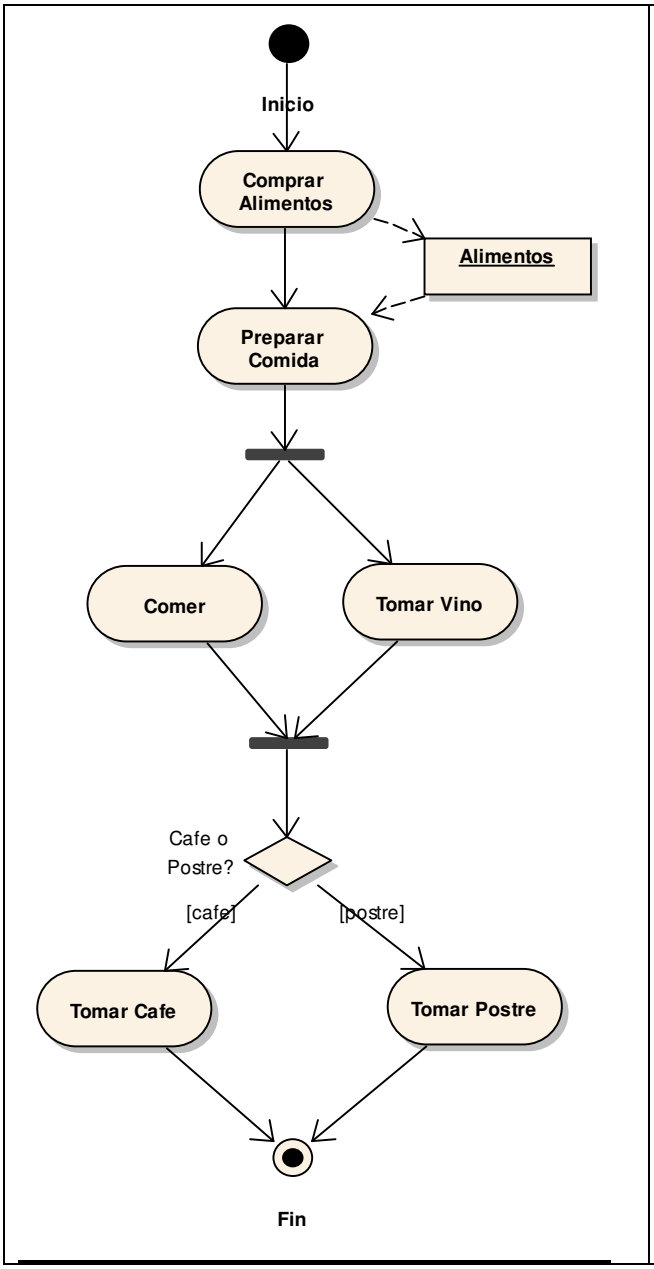
	Secuencia	Colaboración
Muestra comportamiento del sistema	x	x
Indica mensajes intercambiados en su orden	x	x
Destaca un escenario sin flujos alternativos	x	x
Visualiza los enlaces entre los objetos		x
Visualiza secuencia en tiempo	x	

- **Colaboración**: complejas redes de objetos → destacan enlaces
- **Secuencia**: muchos mensajes → los ordenan en tiempo

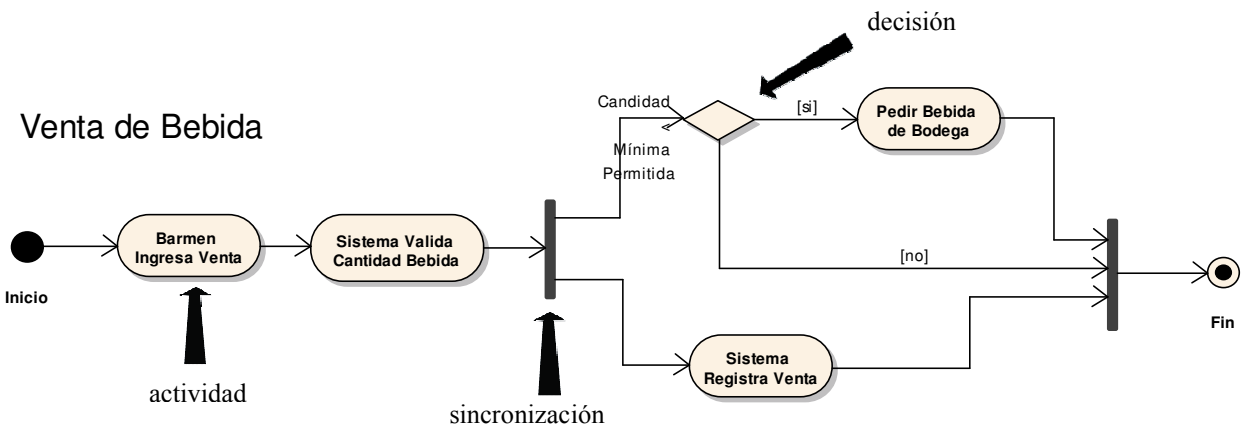
6. Diagramas de Actividades

Es un caso especial del diagrama de estados. Muestra el flujo entre los objetos. Se utilizan para modelar el funcionamiento del sistema y el flujo de control entre objetos.

- Usados para representar el comportamiento del sistema o negocio
- Muestran actividades y procesos
- Destacan:
  - Condiciones y flujos alternativos
  - Tareas y procesos concurentes
  - Responsabilidades sobre ciertas actividades
- Útiles en análisis de negocio para capturar procesos de alto nivel



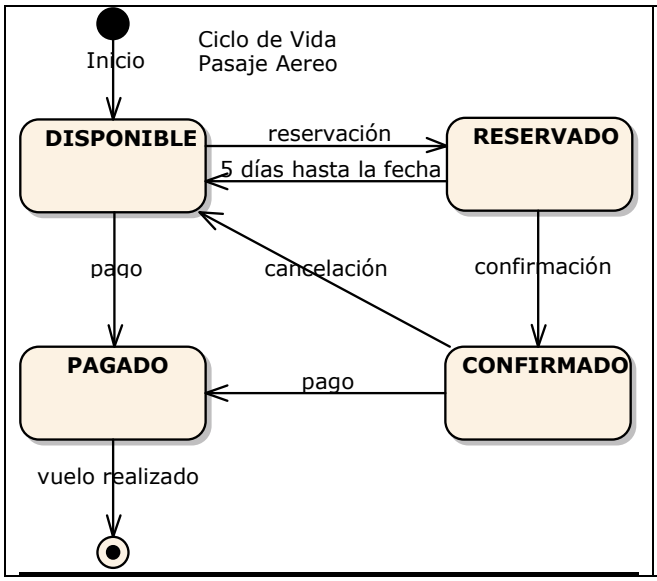
- Inicio de flujo
- Actividades secuenciales
- Actividades producen o consumen datos
- Barra de sincronización
- Actividades concurrentes
- Decisión
- Actividades condicionales
- Fin de Flujo



7. Diagramas de Estados

Muestran los estados, eventos, transiciones y actividades de los diferentes objetos. Son útiles en sistemas que reaccionen a eventos**.**

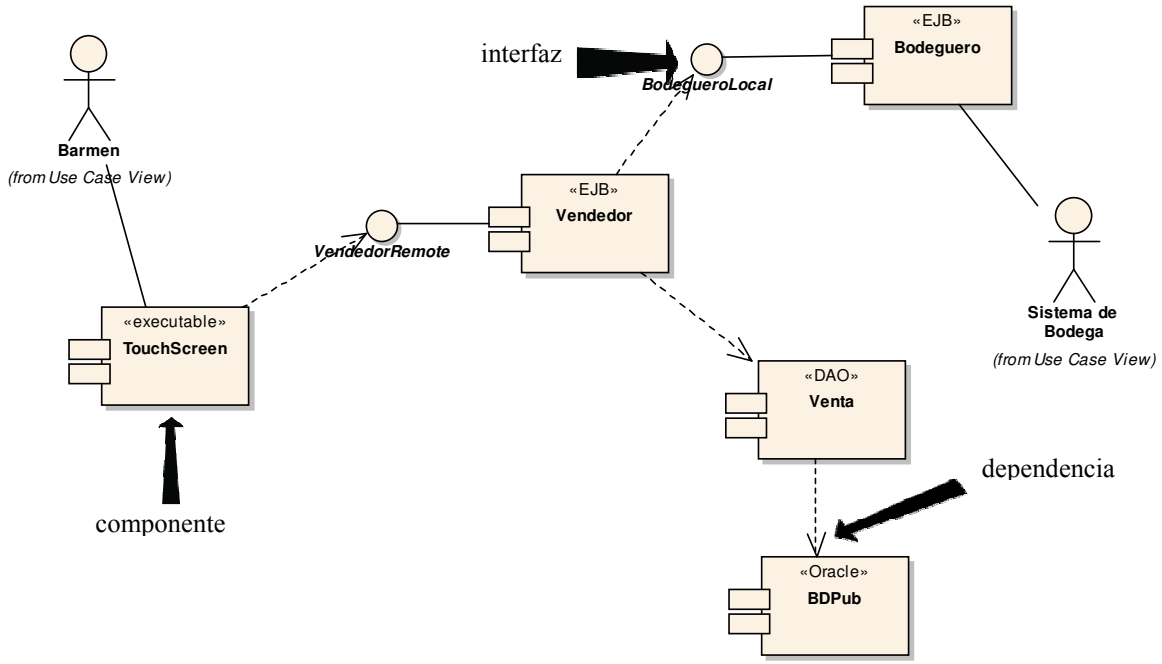
- Usados para representar el comportamiento INTERNO de un objeto de un módulo del sistema
- Muestran estados en los cuales un objeto se puede encontrar
- Destacan:
  - Estados
  - Transiciones y condiciones de las transiciones
  - Actividades realizadas
- Típicamente usados para describer ciclo de vida de un objeto



- Inicio de ciclo
- Estados
- Transiciones
  - Acciones
  - Eventos
- Fin de Flujo

8. Diagramas de Componentes

Muestran la organización de los componentes del sistema. Un componente se corresponde con una o varias clases, interfaces o colaboraciones.



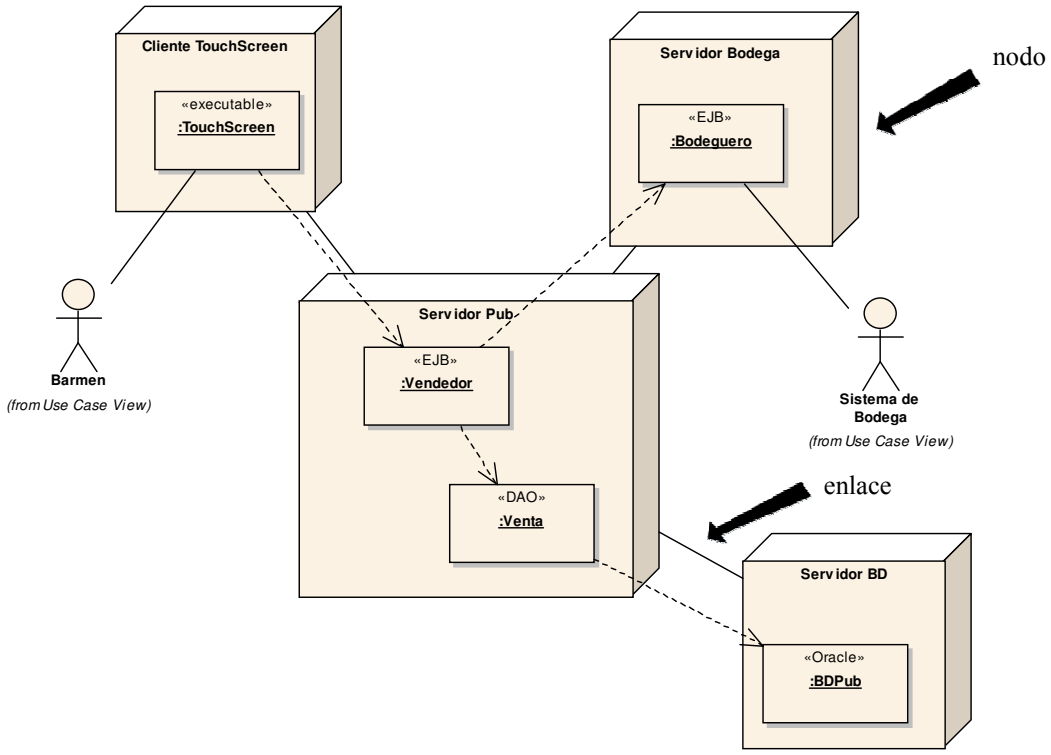
- Usados para mostrar los Módulos Físicos de software:
  - Los ejecutables y librerías dinámicas
  - Las páginas WEB y los scripts
  - Los módulos o funciones, etc.
- Sin embargo se usan más bien para capturar la Organización de los Componentes de Software (EXE, DLL, EJB[2], etc)
- Destacan Dependencias entre los Componentes

9. Diagramas de *Deployment* (despliegue) o implementación

Muestra los nodos y sus relaciones. Un nodo es un conjunto de componentes. Se utiliza para reducir la complejidad de los diagramas de clases y componentes de un gran sistema. Sirve como resumen e índice.

Usados Para Modelar las Relaciones entre el *Software* y el *Hardware*

- Mapeo de los Componentes de Software a los Nodos de Hardware
- Típicamente contienen elementos tales como: Servidores, Procesadores, Impresoras, Redes computacionales, etc.



Documentación Interna

Un programa no solo debe ser entendido por la máquina. Frecuentemente el código es leído por muchas personas para repararlo, ampliarlo o, simplemente, para evaluarlo. Por lo tanto, es fundamental que el programa esté bien redactado, con estilo, para que su significado sea claro y comprensible.

Se logran programas comprensibles

- Aumentando la legibilidad del código
- Aplicando técnicas sistemáticas de programación
- Documentando adecuadamente los programas

[2] Enterprise Java Beans

Las convenciones de código son la mejor manera de asegurar la legibilidad de los programas. Es altamente recomendable atenerse a estándares y convenciones. Existe gran variedad de convenciones, y todas son buenas, siempre que sean respetadas.

Particularmente Sun ofrece para Java una serie de reglas, las "Code Conventions for the Java Programming Language", a las que se puede acceder en **http://java.sun.com/docs/codeconv/.**

Un buen estilo de codificación facilita la corrección y la evolución del código, es decir, favorece una de las reconocidas cualidades del software, la mantenibilidad (reparabilidad-extensibilidad).

Para determinar si un programa está bien escrito se evalúa su legibilidad o **capacidad de comunicar lo que hace** a otros programadores que tengan que leer el código fuente.

Un programador dispone de cuatro mecanismos básicos para comunicarse con sus lectores:

1. El estilo de programación: un orden sistemático facilita la búsqueda al lector
2. Los comentarios
3. La nomenclatura: los nombres de las variables, constantes y métodos
4. La identación: los espacios en blanco

1. El estilo de programación - Reglas de buen estilo

El código fuente debe organizarse de la siguiente forma:

1. documentación (javadoc) de la clase o interfaz
2. *class* o *interface*
3. variables de clase (*static*)
4. variables de instancia u objeto
5. constructores
6. accessors
7. demás métodos

Un orden sistemático facilita la búsqueda al lector.

Sobre las variables

- Las variables de clase u objeto no deben ser públicas → para que nadie pueda alterarlas desde fuera de la clase.
- Las variables deben ver reducida su existencia al mínimo tiempo posible, es decir, declararse en el ámbito más reducido que sea posible → para que nadie se confunda en su uso.
- Se preferirán variables de método a variables de objeto.
- Las variables deben inicializarse inmediatamente → para evitar que la variable contenga un valor fuera de control. Ej: `int nMuestras = 0;`
- Las variables con un ámbito muy amplio deberán tener nombres largos y muy significativos. Las de ámbito muy restringido pueden tener nombres muy cortos → intuitivamente el lector pensará que es una variable sin futuro, que no necesita retener. Ej.: `int i; char c; String s;`
- Las variables no deben tener nombres ambiguos → dificulta determinar para qué sirve.

```
int valor= 0;       // incorrecto → valor de qué? Todos son valores
```

- Las variables jamás deben albergar cosas diferentes → dificulta determinar para qué sirve. Tarde o temprano el programador la usará con el contenido inadecuado.

  Ej.:
```
int numero= 0;
numero= cuentaOvejas();
numero= cuentaSillas();
```

Sobre las expresiones

- Evite expresiones (numéricas o condiciones lógicas) complejas. Si una expresión se complica, considere la creación de variables auxiliares o métodos auxiliares (posiblemente privados).

Sobre las estructuras de programa

- No use las construcciones `break` o `continue`, salvo que sea estrictamente necesario → para que los bucles sean legibles, sin sorpresas.
- Un bucle no debe ocupar más de una pantalla (unas 20 líneas de código) desde su inicio hasta su final, incluyendo líneas preliminares para inicializar variables. Si no fuera así, sería conveniente crear métodos auxiliares con nombre propio → para poder leerlos en pantalla en forma completa.
- Evite el anidamiento excesivo de estructuras `for, while, if, switch`. Si hay más de tres (3) estructuras anidadas, considere la oportunidad de introducir métodos auxiliares con nombre propio.

2. Los Comentarios

Antiguamente, un principio básico del procesamiento de datos señalaba la práctica de mantener archivos independientes, de código y de documentación, en sincronía. Los resultados de hecho confirman que no es una buena práctica. La documentación de los programas es notablemente pobre y su mantenimiento peor. Los cambios hechos en el programa no aparecen pronto, exacta e invariablemente en el documento.

Actualmente se recomienda combinarlos en un solo archivo, conteniendo toda la información incorporados en el archivo fuente, mediante comentarios. Esto es un incentivo poderoso hacia el propio mantenimiento y una seguridad de que la documentación será siempre manejada por el usuario de programa. Dichos programas son llamados auto-documentados.

Todos los lenguajes de programación permiten intercalar comentarios. El contenido de los comentarios puede ser texto arbitrario en lenguaje natural. El compilador ignora los comentarios.

Documentar el código de un programa es añadir suficiente información como para explicar lo que hace, punto por punto, de forma que no sólo los ordenadores sepan qué hacer, sino que además los humanos entiendan qué están haciendo y por qué.

Hay dos reglas que no se deben olvidar nunca:

- todos los programas tienen errores y descubrirlos sólo es cuestión de tiempo y de que el programa tenga éxito y se utilice frecuentemente
- todos los programas sufren modificaciones a lo largo de su vida, al menos todos aquellos que tienen éxito

Qué documentar

Hay que añadir explicaciones a todo lo que no es evidente. No hay que repetir lo que se hace, sino explicar por qué se hace. Esto se traduce en:

- Una descripción de la funcionalidad **(¿qué hace?)** de cada clase y de cada método.
- Una descripción del comportamiento **(¿cómo lo hace?)** de cada clase y de cada método.
- Una descripción del propósito de cada constante, declaración de tipo y variable.
- ¿qué algoritmo se está usando? ¿qué limitaciones tiene el algoritmo? ¿... la implementación?
- ¿qué se debería mejorar ... si hubiera tiempo?

Tipos de Comentarios

Java dispone de tres notaciones para introducir comentarios:

- **javadoc**: Su propósito es generar documentación externa. Comienzan con los caracteres "/**", se pueden prolongar a lo largo de varias líneas (que probablemente comiencen con el carácter "*") y terminan con los caracteres "*/".
- **una línea:** Comienzan con los caracteres "//" y terminan con la línea. Se usa para documentar código que no se necesita que aparezca en la documentación externa (que genere javadoc). Este tipo de comentarios se usa incluso cuando el comentario ocupa varias líneas, cada una de las cuales comenzará con "//".

- **tipo C**: Su propósito es anular código, cuando por algún motivo se desea mantener código que es obsoleto. Para que no se ejecute, se comenta. (En inglés se suele denominar "**C**omment out". De allí su nombre). Comienzan con los caracteres "/*", se pueden prolongar a lo largo de varias líneas (que probablemente comiencen con el carácter "*") y terminan con los caracteres "*/".

Cuándo colocar un comentario

Por obligación (**javadoc**):

1. al principio de cada clase
2. al principio de cada método
3. ante cada variable de clase

Por conveniencia (**una línea**):

4. al principio de fragmento de código no evidente
5. a lo largo de los bucles

Y por si acaso (**una línea**):

6. siempre que se haga poco habitual
7. siempre que el código no sea evidente

Es primordial que cuando un programa se modifica, los comentarios se modifiquen al mismo tiempo, de modo que los comentarios no acaben haciendo referencia a un algoritmo que ya no se utiliza.

Javadoc

Javadoc es una herramienta integrada al SDK de Java, que permite documentar de manera rápida y sencilla las clases y los métodos que se proveen. Genera un conjunto de páginas web (formato html) a partir de los archivos de código. Esta herramienta toma en consideración algunos comentarios para generar una documentación bien presentada de clases y componentes de clases (variables y métodos). Para ello es preciso que los comentarios tengan una sintaxis especial. Deben comenzar por "/**" y terminar por "*/", incluyendo una descripción y algunas etiquetas especiales:

```
/**
 * Parte descriptiva.
 * Que puede consistir de varias frases o párrafos.
 *
 * @etiqueta texto específico de la etiqueta
 */
```

Estos comentarios especiales deben aparecer justo antes de la declaración de una clase, un atributo o un método, en el mismo código fuente. En las siguientes secciones se detallan las etiquetas (tags) que javadoc sabe interpretar en cada uno de los casos. Como regla general, hay que destacar que la primera frase (el texto hasta el primer punto) recibirá un tratamiento destacado, por lo que debe aportar una explicación concisa y contundente del elemento documentado. Las demás frases entrarán en detalles.

Documentación de clases e interfaces

Deben usarse al menos las etiquetas **`author`** y **`version`**

La tabla muestra todas las etiquetas posibles y su interpretación:

@author	nombre del autor
@version	identificación de la versión y fecha
@see	referencia a otras clases y métodos
@since	indica desde qué versión o fecha existe esta clase o interfaz en el paquete
@deprecated	Esta clase no debería usarse pues puede desaparecer en próximas versiones

Documentación de constructores y métodos

Deben usarse al menos las etiquetas:

- **`@param`**: una por argumento de entrada
- **`@return`**: si el método no es *void*
- **`@exception`**: una por tipo de *Exception* que se puede lanzar

La tabla muestra todas las etiquetas posibles y su interpretación:

@param	nombre del parámetro	descripción de su significado y uso
@return		descripción de lo que se devuelve
@exception	nombre de la excepción	excepciones que pueden lanzarse
@since	indica desde qué versión o fecha existe este constructor o método en la clase	
@deprecated	Indica que este método no debería usarse pues puede desaparecer en próximas versiones	

Documentación de atributos

Ninguna etiqueta es obligatoria.

La tabla muestra todas las etiquetas posibles y su interpretación:

@since	Indica desde qué versión o fecha existe este atributo en la clase
@deprecated	Indica que este atributo no debería usarse pues puede desaparecer en próximas versiones

Ejecución de javadoc

La mayor parte de los entornos de desarrollo incluyen un botón para llamar a javadoc, así como opciones de configuración. No obstante, siempre se puede acceder al directorio en que se instaló el JDK y ejecutar javadoc directamente sobre el código fuente Java:

```
...> {directorio de instalación}/javadoc *.java
```

La herramienta javadoc admite muchas opciones. Algunas de las más usadas aparecen a continuación:

```
javadoc [options] [packagenames] [sourcefiles]
```

`-public`	`Muestra solo las clases y miembros públicos`
`-protected`	`Muestra clases y miembros públicos/protegidos (default)`
`-package`	`Muestra las clases y miembros públicos/protegidos/Packaged`
`-private`	`Muestra todas las clases y miembros`
`-sourcepath <pathlist>`	`Especifica donde encontrar los archivos fuentes`

-classpath <pathlist>	Especifica donde encontrar los archivos .class de usuario
-d <directory>	Directorio destino para los archivos generados
-windowtitle <text>	"texto" es el título para la ventana del navegador
-help	despliega un catálogo completo de opciones.

3. Nomenclatura

Los nombres de clases, variables y métodos son un mecanismo básico para transmitir al lector lo que piensa el programador. En principio, cualquier convenio sería bueno, con el único requisito de que se siguiere sistemáticamente.

A continuación se describe el convenio más habitual en la comunidad de programadores Java.

Reglas de nomenclatura

REGLA GENERAL: cuando un nombre conste de varias palabras, se escribirán una tras otra, sin solución de continuidad, comenzando cada palabra por mayúscula:

Variable**D**e**T**res**P**alabras
Numero**D**e**T**elefonos**E**n**L**a**O**ficina

Las excepciones a esta regla general se señalan en cada caso.

- Los nombres para las clases deben ser sustantivos. Se sigue la REGLA GENERAL.
```
ColaLlamadas, Estancia, ListaDeNombres
```
- Los nombres de las variables siguen la REGLA GENERAL; pero la primera letra será minúscula.
```
int numeroDeCarneros;
```
- Las constantes ("variables" final) se escribirán íntegramente con mayúsculas. Si constan de varias palabras, se separarán estas por "_".
```
static final int COLOR_ROJO= 0xFF0000;
```
- Los nombres de los métodos deben ser verbos. Se sigue la REGLA GENERAL; pero la primera letra será minúscula.
```
String getNombre ();
void setNombre(String string);
```
- Si una palabra es un acrónimo, se sigue la REGLA GENERAL de escribir solo el primer carácter con mayúscula (con las salvedades antes apuntadas).
```
class Dni
dniDelSujetoPasivo
getUrl()
```

4. Identación

El identación o i**n**dentación (términos utilizados indistintamente), indican los cambios de línea y espacio en blanco al principio de una línea o párrafo. Es un mecanismo básico para transmitir al lector lo que piensa el programador. Se realiza una alineación de los párrafos para los diferentes elementos del código para mostrar los niveles de anidamiento.

A continuación se describe el convenio más habitual en la comunidad de programadores Java.

- No escriba líneas con más de 70 caracteres. Si una sentencia requiere más del límite, considere partirla en varias líneas, dejando claro en todas ellas (menos la última) que se sigue a continuación.

```
String mensaje= "Este es un mensaje un poco largo que, " +
                "claramente, requiere de más de una línea " +
                "para atenerse a la regla.";

int acumulado= capitulo[1] + capitulo[3] +
               capitulo[5] + capitulo[7] +
               capitulo[11];

funcion(argumento1, argumento2, argumento3,
        argumento4, argumento5);

for (int indice= 0;
     indice < tablaDeRutasDeAcceso.length;
     indice+= 2) {
  ... ... ...
}
```

Reglas de identación

- Utilice dos espacios en blanco como medida básica de identación. Evite el uso de caracteres de tabulación (TAB).

```
int maximo= tabla[0];
for (int i= 0; i < tabla.length; i++) {
  if (tabla[i] > maximo)
    maximo= tabla[i];
}
```

- Para identar clases e interfaces se usará el patrón del ejemplo.

```
class UnaClase extends OtraClase
  implements UnaInterfase, OtraInterfase {
  ... ... ...
  ... ... ...
}
```

- Para identar métodos se usará el patrón del ejemplo.

```
public void unMetodo ()
  throws UnaExcepcion {
  ... ... ...
  ... ... ...
}
```

- El tipo de sentencias condicionales `if-else` deberían adaptarse a alguna de las siguientes formas:

```
if (condicion) {
    sentencias;
}
if (condicion) {
    sentencias;
} else {
    sentencias;
}

if (condicion) {
    sentencias;
} else if (condicion) {
    sentencias;
} else if (condicion) {
    sentencias;
}
```

Las sentencias if ***siempre*** deben incluir las llaves. Si se emiten, esa sentencia está ***sentenciada*** a ser una fuente de errores.

```
if (condicion) // EVITARLO!  FALTAN LAS LLAVES {}!
    sentencia;
```

- Para identar estructuras SWITCH se usará el patrón del ejemplo.

```
switch (condicion) {
case ABC:
    sentencias;
    /* sigue la ejecución */
case DEF:
    sentencias;
    break;
case XYZ:
    sentencias;
    break;
default:
    sentencias;
    break;
}
```

Cada vez que una cláusula **case** deba seguir la ejecución en la siguiente cláusula **case**, es decir, no incluya la sentencia **break**; es imprescindible incluir un comentario en el lugar que ocuparía la sentencia break, tal como se muestra en el ejemplo con el comentario **/* sigue la ejecución */.**

Cada sentencia **switch** debería incluir el caso **default**. Aquí la sentencia **break** es redundante, pero debería incluirse siempre para evitar errores en caso de que se añada una sentencia case posterior y se ejecute sin deber hacerlo.

- Para identar bloques while se usará el patrón del ejemplo.

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

- Para identar bloques for se usará el patrón del ejemplo.

```
for (initializacion; condicion; incremento) {
   sentencias;

}
```

- Para identar estructuras TRY se usará el patrón del ejemplo.

```
try {
    sentencias;
} catch (ExceptionClass e) {
  sentencias;
}
```