PLANETALIA
Formación y Consultoría

MANUAL DE ESTILO DE PROGRAMACIÓN

Rev 9.01

La última versión de este manual está disponible siempre en
http://www.ahristov.com/tutoriales/Blog/Estilo_de_Programacion.html

1 ÍNDICE

2 INTRODUCCIÓN

2.1 Presentación y Objetivo

En cualquier proyecto en el que intervenga más de una persona se hace necesario seguir unas guías comunes de desarrollo para asegurar la correcta comprensión de todo el código, así como su mantenimiento posterior por personas que no han participado en el desarrollo. La disparidad de estilos, así como el uso - por parte de programadores con poca experiencia - de prácticas desprestigiadas por la industria, tienen un efecto mortalmente sedante en la salud del proyecto : Inicialmente aparentan ser productivas, ingeniosas, eficaces, pero a largo plazo, cuando llega el inevitable momento de corregir errores que ocurren en entornos o condiciones no previstas, o cuando hay que hacer alguna ampliación del sistema, o cuando un programador nuevo necesita incorporarse al proyecto; únicamente entonces se pueden ver los verdaderos efectos negativos de esas prácticas.

En un hipotético mundo ideal, cuando alguien aborda un problema, realiza una completa investigación sobre el tema, examina los diferentes aspectos formales, aprende la teoría subyacente y el historial de casos previos, así como las decisiones tomadas en el pasado, sus motivaciones, consecuencias y revisiones. Antes del primer CREATE TABLE, un DBA-modélico debería conocer las bases del modelo relacional de Codd, los problemas de SQL y su divergencia con el modelo relacional y un sinfín de detalles "finos" adicionales.

Está claro que esta opción no es viable, sin embargo. En el mundo real, existen aspectos prácticos que constituyen limitaciones a esta búsqueda de la solución perfecta. No es casualidad el viejo *adagio* que afirma "*Lo mejor es enemigo de lo bueno*". Los proyectos deben acabarse en un tiempo escaso (y desde luego finito), existe una rotación de personal importante, no existen sistemas de gestión del conocimiento y a veces ni siquiera sistemas de gestión de la calidad o del proceso.

Tampoco es viable pretender que cada programador novel que se incorpore a un proyecto lea e interiorice el equivalente a una pequeña librería de un centenar de libros antes de escribir una única línea de código, motivo por el que el modo "cookbook" que tanto detestan algunos autores (Pascal 2007) es tan popular.

Este manual pretende ser una especie de torniquete - una solución de emergencia, relativamente asequible y conciso, que pueda aplicarse de manera ***temporal y transitoria*** mientras el programador investiga y explora las bases y ramificaciones de las reglas expuestas y explora la amplia literatura existente. El manual no debe considerarse un sustituto a una cura duradera en forma de conocimiento completo, de la misma forma que un torniquete nunca puede ser una solución permanente a una hemorragia.

Por lo tanto, este manual tiene una forma esquemática : intenta destilar de forma muy sintética conjuntos de buenas prácticas extraídas del sector, recomendaciones procedentes de multitud de libros sobre ingeniería del software y resultados de métricas y estadísticas de miles de proyectos. El manual únicamente *indica qué y cómo hacer*, pero no tiene como objetivo convencer o justificar cada una de las recomendaciones dadas. A los interesados en conocer el origen y el porqué de las normas, o las consecuencias de violarlas, les recomiendo encarecidamente la lectura de la bibliografía en el capítulo final.

2.2 Historia de este manual

En (casi) todos los proyectos en los que he participado desde hace más de 20 años, siempre me ha obsesionado el tema de la calidad del desarrollo, y dada mi inclinación científica, siempre he buscado cosas que de forma demostrable (a diferencia de "opinable") condujesen a una mayor calidad (verificable) en el software. Por ello siempre me ha interesado todo lo relacionado con los sistemas de aseguramiento de la calidad o las métricas de software y los sistemas de predicción de defectos.

Una de las piezas claves de esta "obsesión" es un manual de buenas prácticas a nivel de codificación que elaboré en 1997 para una empresa en la que dirigía varios proyectos, llamada PSD (Personal Systems Design).

Hasta ahora, nunca se me había ocurrido que este tipo de guías pudieran interesar a nadie salvo a mí. Uno siempre supone que cualquier empresa medianamente seria dispone de estándares para homogeneizar la forma de construir software y que probablemente esos estándares sean mucho más precisos y apropiados que cualquier guía que haya podido humildemente crear yo. Sin embargo, de forma sistemática mi experiencia en diferentes sitios siempre ha sido la contraria. Salvo en un par de ocasiones, la inexistencia de estándares de codificación era notoria y dolorosa, llevando a tener que depurar una y otra vez los mismos errores causados por las mismas malas prácticas.

Y no se trataba de que los responsables no supiesen dónde estaban los problemas, sino del mal endémico de siempre en el sector - la falta de tiempo. Tiempo para sistematizar el conjunto de buenas prácticas, tiempo para expresarlo en forma de estándar, tiempo para implantarlo y tiempo para llevar un seguimiento del mismo.

Hace varios meses, me topé accidentalmente con el blog[1] de un antiguo compañero de trabajo - Nicolás Aragón (alias "NicoNico"), en el que relataba su experiencia en diferentes empresas y - refiriéndose entre otras cosas a este manual - comentaba básicamente la misma situación.

Así que dije - ¡qué demonios!, por qué no darle un repaso y publicarlo por si a alguien le pudiese servir de ayuda. Y aquí está - la versión 9.0 revisada y adaptada de aquel manual de estilo de programación, con licencia Creative Commons.

[1] **http://espira.net/nico/personal_systems_design**.

2.4 Iconos utilizados

Las reglas tienen el siguiente formato:

P1.1   Regla

Donde:

- La primera columna indica el identificativo de la regla, para poder ser referenciada con facilidad. Las reglas de cada capítulo están numeradas secuencialmente.
- La segunda columna indica el tipo de regla, siendo posibles las siguientes:
  -  Prohibición - señala ejemplos de código o formas de proceder que son peligrosas, y que por lo tanto no están permitidas dentro de un programa bajo ningún concepto y sin excepción.
  -  Cautela - señala ejemplos de código o formas de proceder que son dudosas y cuyo uso no se recomienda por constituir un potencial problema en el futuro, si bien su uso es tolerable **cuando está justificado, siendo además pocas las situaciones justificables**.
  -  Obligación - señala especificaciones que forman parte del estándar mínimo común a utilizar durante el desarrollo, y que por lo tanto se deben aplicar **siempre**, sin excepción.
  -  Sugerencia - indica especificaciones que, si bien no son obligatorias, son muy recomendables.
  -  Definición - indica una definición.
- La tercera columna contiene un icono que indica la aplicabilidad de la regla:

Icono	Significado
	Regla aplicable a Java
	Regla aplicable a Delphi y lenguajes similares (Object Pascal, Modula-2,etc)
	Regla aplicable a los lenguajes de la familia .NET (C#, VB)
	Regla universal
	Regla aplicable a lenguajes con punteros y memoria no gestionada

- Por último, la cuarta columna contiene la propia regla.

2.5 Convenciones de texto

En este manual, el código fuente, así como los resultados en pantalla de la ejecución de dicho código (cuando no son gráficos), se representan de la siguiente forma:

```
Ejemplo de texto de código
```

Estos números de línea se utilizan exclusivamente para comentar el código fuente en el texto que le sigue, pero en ningún caso deben ser introducidos como parte del programa.

Igualmente en ocasiones en el interior del código fuente se utiliza la elipsis:

```
public class Hola {
  public static void main(String[] args) {
    ....
  }
}
```

lo cual indica que en ese lugar puede ir cualquier código o un código ya comentado anteriormente (el caso concreto dependerá del texto).

Cuando se hace mención a métodos, clases o palabras clave dentro del texto se utiliza el siguiente estilo : **`Integer.parseInt`**.

En algunos ejemplos se utiliza la sintaxis **`<concepto>`**, por ejemplo de la siguiente forma:

```
for (int i = <valor inicial>; i != <valor final>; i++)
```

Esto significa que la totalidad de la expresión encerrada entre < y >, incluyendo los propios símbolos, ha de ser sustituida por el valor apropiado. Por ejemplo, si el valor inicial es 5 y el final es 20, entonces la expresión anterior quedaría escrita así:

```
for (int i = 5; i != 20; i++)
```

La aplicabilidad de las reglas se indica mediante los siguientes iconos:

2.6 Licencia y Copyright

Este manual es Copyright © 1994 - 2007 por Alexander Hristov y se distribuye en formato PDF bajo licencia Creative Commons, en la modalidad "Attribution, noncommercial, no-derivatives" y su versión 3.0. Un *resumen humano* de la licencia es:

El texto completo de la licencia puede encontrarse en **http://creativecommons.org/licenses/by-nc-nd/3.0/legalcode**
El texto legal presente en el enlace indicado tiene prioridad sobre cualquier otra formulación.

A los que deseen citar este manual les rogaría que proporcionasen un vínculo a la página web en lugar de a un PDF concreto, ya que como todo conjunto de buenas prácticas, las mismas pueden evolucionar y es recomendable que se enlace siempre a la última versión. Forma de cita:

```
Alexander Hristov, Manual de Estilo de Programación, 2007 -
http://www.ahristov.com/tutoriales/Blog/Estilo_de_Programacion.html
```

2.7 Acerca de Planetalia

Planetalia (**www.planetalia.com**) desde hace más de 10 años se especializa en **formación a medida y consultoría de proyectos** basados en tecnologías de vanguardia - desde Web 2.0 hasta XBRL, desde SOA hasta DNI-E.

En la actualidad, cubrimos las siguientes tecnologías y plataformas:

- ✓ **Lenguajes :** Java, PHP, Ruby, C#, VB.NET, ActionScript para Flash MX, Python, Perl
- ✓ **SOA :** Servicios Web (Microsoft WCF, JAX-WS, Axis), ESB (JBI, Mule, Apache ServiceMix, Bea Aqualogic),
- ✓ **Metodologías :** XP, Agile, SCRUM, RUP, TDD
- ✓ **Plataformas** : J2EE, .NET
- ✓ **Servidores :** Weblogic, Websphere, Glassfish, OAS, JBoss, Tomcat, IIS, Sharepoint
- ✓ **Entornos de Desarrollo :** Eclipse, WSAD, NetBeans, JBuilder, Visual Studio.NET
- ✓ **Arquitecturas y Frameworks :** Struts, Hibernate, JDO, AOP, Ruby on Rails
- ✓ **Diseño :** UML 2.0, Patrones
- ✓ **Web 2.0** : AJAX, Adobe Flash, Adobe Flex, Microsoft Silverlight
- ✓ **Usabilidad y Accesibilidad**
- ✓ **Tecnologías XML,** incluyendo **XBRL**
- ✓ **Tecnologías embebidas, inalámbricas y móviles -** J2ME, JavaCard
- ✓ **Seguridad :**Criptografía, PKI, Implantación de soluciones basadas en DNI-E, JCA, JAAS, Kerberos

Para más información, no dude en dirigirse a nosotros.

2.8 Acerca del Autor

Alexander Hristov acumula más de 20 años de trayectoria profesional en el mundo de las Tecnologías de la Información, de los cuales más de 10 en dirección de proyectos para numerosas empresas. Paralelamente a su labor puramente técnica, siempre ha estado interesado en la formación y divulgación, tanto en el sector técnico como de la ciencia en general. Ha colaborado durante muchos años con la sección técnica de PC Actual, ha sido profesor externo en la UNED, colaborador del CNICE/Ministerio de Cultura para el proyecto Arquímedes, ponente el II congreso nacional de Sociedades de Valoración y autor de numerosos artículos divulgativos.

Actualmente es profesional independiente, ofreciendo servicios de formación, consultoría y dirección de proyectos. Se puede contactar con él en su página web - **www.ahristov.com**

2.9 Comentarios, Sugerencias y Versiones Actualizadas

Como siempre, se agradecen todos los comentarios, correcciones, críticas o sugerencias sobre este manual. . Para cualquier comentario o sugerencia, por favor dirigirse a **www.ahristov.com** o **www.planetalia.com**

La última versión de este manual siempre puede obtenerse de
http://www.ahristov.com/tutoriales/Blog/Estilo_de_Programacion.html

3 REQUERIMIENTOS

O3.1 ☞ Todos los requisitos de un proyecto deben estar disponibles en una única ubicación central de fácil acceso y actualización, y que permita una búsqueda rápida de los requerimientos

3.1 Lenguaje

O3.100 ☞ Cada requerimiento contendrá al menos la siguiente documentación

- Un identificativo único y corto ("clave primaria") que identifique unívocamente al requerimiento, que permita localizarlo con facilidad, que sea fácil de escribir, copiar y pegar, y que se pueda utilizar sin problemas o interferencias dentro del código fuente, documentos y pruebas para referirse a este requerimiento
- Fecha del requerimiento
- El requerimiento en sí
- Su prioridad, preferentemente en una escala *reducida*
- La persona del proyecto que ha tomado este requerimiento
- La persona del cliente que ha indicado este requerimiento y a la que se pueda referir uno en caso de dudas.
- Requerimientos de los que depende, pero **no** a los que afecta (ya que esto es una información derivada que debe ser mantenida automáticamente)

O3.101 ☞ Un buen requerimiento debe ser (IEEE Software Engineering Standards Commiteee 1998)

- Correcto y claro
- No ambiguo (es decir, con una única posible interpretación)
- Consistente con el resto de requerimientos y no redundante (es decir, no derivable de ellos)
- Verificable
- Modificable
- Trazable (proporcionando trazabilidad tanto hacia atrás : dependencias de otros requerimientos, origen del requerimiento, etc. como hacia adelante : documentos y código originados por el requerimiento).
- Priorizable (tener una prioridad claramente definida en relación con los demás requerimientos)

S3.102 ☺ Siempre que sea posible, se intentará que la verificabilidad de los requisitos pueda realizarse de forma automatizada y no manual.

S3.103 ☺ Siempre que sea posible, se intentarán expresar los requerimientos de manera formal

La expresión formal ***no*** sustituirá la indicación dada por el cliente, sino que únicamente la complementará. La expresión formal suele tener el siguiente aspecto:

$$\bigwedge_i P_i \xrightarrow{E} \bigwedge_j Q_j$$

donde

- P_i es el conjunto de precondiciones (antecedentes)

- E es el estímulo que produce la transición
- Q_j es el conjunto de postcondiciones (consecuentes, o efectos de la operación)

Por ejemplo : "*Los usuarios autenticados con privilegios de administración, pulsando la opción 'ver estado' visualizarán el total de conexiones disponibles*".

Antecedentes :

P_1 = Usuario Autenticado

P_2 = Usuarios con privilegios de administración

E = Pulsar "Ver Estado"

Asunciones

$P_2 \rightarrow P_1$

Consecuentes

Q_1 = Total de conexiones disponibles mostrado en pantalla

S3.104 ☺ Si se dispone de expresión formal de los requerimientos, se deberían verificar los mismos mediante interrogación al contrario del cliente:

$$\forall i, ¿ \neg P_i \rightarrow \neg \left(\bigwedge_j Q_j \right) ?$$

En el ejemplo anterior:

- ¿Si el usuario no está autenticado, entonces es que no puede ver las conexiones disponibles?
- ¿Si el usuario no tiene privilegios de administración, entonces no puede ver las conexiones disponibles?

3.2 Requisitos precisos

O3.200 ☞ Todos los requerimientos concernientes a magnitudes físicas deberán indicar unidades de medida y tolerancias de error

O3.201 ☞ Todos los requerimientos que afecten a resultados visuales (*informes, pantallas, estadísticas, etc.*.) deben indicar los datos que aparecerán, su ordenación y sentido y su formato.

O3.202 ☞ Todos los requerimientos relacionados con la interfaz con sistemas externos deben indicar los formatos de intercambio, los protocolos, las temporizaciones y el manejo de las situaciones de error

3.3 Lenguaje

La falta de requerimientos claros está la raíz de en entre el 13% y el 20% de los casos de proyectos fallidos. (Alexander and Stevens 2002)

S3.300 ☺ Siempre que sea posible, se expresarán los requerimientos en forma positiva. Por ejemplo "*El proceso X debe tardar 5 seg o menos*" frente a *"El proceso X no debe tardar más de 5 segundos"*.

S3.301 ☺ Cada requerimiento debe expresarse de la forma más sencilla y corta posible, sin utilizar florituras lingüísticas, sin divagaciones, sin jergas de ningún tipo y preferentemente mediante una única frase.

S3.302 ☺ Los requerimientos no necesitan estar auto justificándose.

O3.303 ☞ Se debe utilizar el vocabulario y terminología de forma coherente, sin que pueda ocurrir que en diferentes requerimientos el mismo concepto aparezca expresado de formas diferentes ("informe - reporte", "formulario - cuestionario", etc.)

O3.304 ☞ Se debe utilizar el vocabulario y terminología del cliente y no sustituirlos. Toda la terminología utilizada se definirá de forma explícita y las definiciones se verificarán con el cliente.

P3.305 ☠ Evitar el *"wishful-thinking*" (utopías imposibles) en los requerimientos: "El sistema será 100% tolerable a intrusiones y detectará cualquier intento de acceso reportando al atacante"

P3.306 ☠ En los requerimientos, nunca se utilizarán expresiones idiomáticas vagas o ambiguas.

Por ejemplo:

- Costumbrismos : *generalmente, habitualmente, casi siempre, con frecuencia, uso típico, típicamente, preferiblemente*
- Condiciones Subjetivas : *aproximadamente, adecuado, apropiado, conveniente, fácil, poco, mucho, muy, cómodo, rápido, flexible, confiable, actualizable, seguro, estético, bueno, malo, conveniente* (y todos sus sinónimos y antónimos)
- Listas incompletas : *etc., entre otros, en particular*
- Condicionales *: debería, si ("si fuese necesario"), podría, sería, posiblemente, probablemente*
- Conjunciones que enmascaran la presencia de varios requisitos en uno, o de ambigüedades : *y, o, con, al igual que,*

P3.307 ☠ No se utilizarán cuantificadores ambiguos: *Mucha memoria, poco tiempo, rápido, lento.* Todos los requerimientos relativos a métricas deberán O cuantificarse O eliminarse. La cuantificación puede ser exacta, o puede proporcionarse con márgenes de tolerancia o cotas ("*El proceso X debe tardar menos de 5 seg"*)

C3.308 💣 No utilizar los requerimientos para hacer diseño. Síntomas frecuentes : *nombres de componentes, materiales, procedimientos y metodologías de software, nombres de propiedades o estructuras de bases de datos o clases* (*"R34 : El número de pedido quedará almacenado en la base de datos en el campo NUMPEDIDO"*)

P3.309  

No utilizar los requerimientos para hacer planificación de proyecto. *("R87 : Los usuarios autenticados podrán ver las estadísticas de conexiones, estando esta funcionalidad disponible antes del 3 de Febrero")*

C3.310  

No utilizar atributos sin indicar explícitamente a quién pertenecen (Por ejemplo, utilizar *"fecha del pedido"* frente a solamente *"fecha"*, *"estado de la compra*" frente a *"estado*", o *"nombre del cliente"* frente a *"nombre"*), ni siquiera cuando los mismos "pudieran" estar claros a través del contexto.

4 NOMENCLATURA

En esta sección se exponen reglas generales sobre nomenclatura aplicables a cualquier tipo de identificador (ya sean clases, enumeraciones, variables, métodos, etc…). La mayor parte de las reglas proceden de (ISO/IEC 11179 - 1 1999)

O4.100 Salvo por necesidades especiales *y objetivas* que se especificarán **antes** del comienzo de un proyecto (como por ejemplo, participación internacional en el mismo) , todos los nombres estarán en castellano, ya que el grado de conocimiento y fluidez del inglés de los participantes en el proyecto no se puede asegurar.

O4.101 Salvo en constantes o en identificadores creados por generadores de código, no se utilizará el guión bajo (_) en los nombres de los identificadores.

P4.102 No se utilizará la abreviación cuando el único ahorro conseguido es de uno o dos caracteres en el nombre completo. Por ejemplo, *Hora* nunca podrá ser abreviado a *Hr*. No así *HoraEmisión*, que si podrá ser abreviado a *HrEmision*

P4.103 No se usarán nombres cuya única diferencia sea un numeral en cualquier posición. Por ejemplo: *Fichero1* y *Fichero2*.

S4.104 Estadísticamente está demostrado que la longitud óptima para el nombre de una rutina es entre 7 y 15 caracteres, por lo que esta es la longitud recomendada, si bien se admiten otras cuando los nombres sean suficientemente claros.

P4.105 No se utilizará la capitalización para diferenciar entre identificadores distintos

P4.106 Las siguientes partículas están prohibidas en los nombres de identificadores:

- Determinantes de cualquier tipo :
  - *Artículos : el, la, los, las, uno, unos, unas,*
  - *Determinantes Demostrativos : este, ese, aquel, …*
  - *Determinantes Posesivos : mi, tu, su,…, cuyo, cuyos, cuyas*
  - *Cardinales (uno, dos,..),*
  - *Ordninales (segundo , tercero),*
  - *Multiplicativos (triple, cuádruple)*
- Pronombres de cualquier tipo
  - *Personales : yo, tú, el, …, me, te, se,…*
  - *Demostrativos : este, ese*
  - *Posesivos : mi, tu, su…*

Las únicas excepciones a esta regla son el ordinal *Primer* y el multiplicativo *doble*.

C4.107 Salvo causas justificadas, no se recomienda el uso de preposiciones o conjunciones en un nombre

P4.108 No se utilizarán ni sufijos ni prefijos en un nombre cuyo único propósito sea indicar el tipo de identificador. Por ejemplo: `CEvento`, `EventoClass` o `EnumOpciones`, o `varAlumno`.

O4.109 ☞ Únicamente se utilizará el guión bajo como primer carácter de un identificador (por ejemplo: _foo) cuando dicho identificador haya sido generado de manera automática por una herramienta o framework (ya sea en tiempo de compilación o ejecución)

O4.110 ☞ Si el lenguaje dispone de una guía de estilo en cuanto a capitalización, se aplicará (Java, .NET). En el resto de casos, se utilizarán los criterios que figuran a continuación:

O4.111 ☞ En todo momento se utilizarán nombres que sean claros, concretos y libres de ambigüedades. (Por ejemplo, "fechaNacimiento" vs solamente "fecha", o "estadoImpresora" vs "estado")

5 CLASES

5.1 Diseño de Clases

P5.100  

De forma sistemática se ha constatado que es muy complicado anticipar requerimientos futuros y codificar de acuerdo a ellos (Fowler 1998) . Por ello, el diseño y codificación de clases se limitará a hacer *la tarea actual, de la forma más clara, inmediata y sencilla posible*. No se realizará codificación alguna "en anticipación" a ninguna clase de requerimiento hipotético futuro.

"A complex system that works is invariably found to have evolved from a simple system that worked.... A complex system designed from scratch never works and cannot be patched up to make it work. You have to start over, beginning with a working simple system" (Gall 1977)

Esta actitud además evita un problema *muy* frecuente que es el **`Gold Plating`** (Brown y Malveau 1998)

P5.101  

El **`Poltergeist`** (Brown y Malveau 1998), o la "clase perezosa" (Fowler 1998) es una clase con una funcionalidad limitada que no está muy clara por qué existe como clase, que no representa ningún concepto ni entidad del vocabulario del problema y que realiza un trabajo muy escaso. Este tipo de clases deben ser eliminadas. Síntomas de Poltergeists son:

- Existencia de caminos de navegación redundantes
- Clases sin estado
- Asociaciones temporales entre clases
- Clases u objetos de duración muy corta
- Clases con nombres asociados a procesos o verbos.

O5.102 

Las clases deben modelar preferentemente objetos del mundo real o abstracciones. En ocasiones se pueden definir clases con el propósito de reducir o aislar cierta complejidad o detalles implementativos.

O5.103  

Las clases cuyo nombre es un verbo o un proceso son sospechosas de ser realmente "rutinas glorificadas". Debe reconsiderarse si su sitio no es en forma de rutina en una clase distinta.

S5.104  

Siempre que se puedan, se diseñarán clases que den lugar a objetos inmutables.

La inmutabilidad de una clase proporciona importantes beneficios (Bloch 2001), tanto a nivel de rendimiento como a nivel de testeo, reducción de complejidad, ausencia de conflictos en entornos multihebra, etc. Por ello, siempre que exista la posibilidad, debe considerarse la opción de diseñar y hacer clases inmutables. La inmutabilidad debe asegurarse:

- Declarando todos los campos como privados y no sobrescribibles/heredables (**`final`**, en Java)
- No proporcionando ningún método modificador
- Declarando la clase como no extensible (**`final`**, en Java).

5.2 Métricas de Clase

D5.200 📖 Se define como **interacción entre dos clases** (o entre un método y una clase) las siguientes circunstancias:

- La clase o método crea instancias de la otra clase
- La clase o método utiliza métodos de la otra clase
- La clase o método utiliza variables de instancia de la otra clase

Existen varias métricas frecuentes en el diseño de sistemas orientados a objetos universalmente aceptadas (Chidamber y Kemerer 1994):

D5.201 📖 **Weighted Methods Per Class (WMC)** . Existen varias formas de calcular esta métrica: Como la suma de la complejidad ciclomática (McCabe) de los métodos de una clase, o como el número de métodos introducidos en una clase (= suma de métodos totales - WMC del ancestro)

O5.202 ☞ El WMC (utilizando complejidad ciclomática) de una clase debe ser inferior a 100 (Rosenberg, Stapko and Gallo 1999)

D5.203 📖 **Coupling Between Objects (CBO)** se define como el número de clases con las cuales la clase actual interactúa.

S5.205 ☺ Un valor nominal de CBO está entre 1 y 4

D5.206 📖 **Response for A Class (RFC)** se define como el número de métodos que se pueden ejecutar como máximo como resultado de la recepción de un mensaje (invocación) de cualquier miembro de cualquier objeto de esa clase.

O5.207 ☞ El RFC debe ser inferior a 100, e inferior a 5 veces WMC

D5.208 📖 **Class Responsability (CR)** se define como el porcentaje de métodos con pre o post condiciones respecto al total de métodos de la clase.

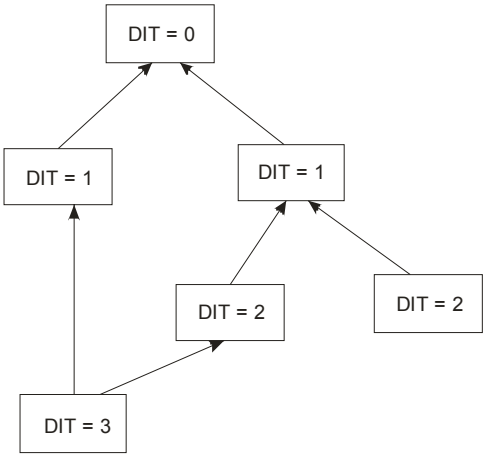
$$CR = \frac{\sum(PRE + POST)}{2\ x\ NOM}$$

S5.209 ☺ Un valor nominal de CR está entre 20% y 75%.

5.3 Herencia

O5.300 ☞ Las clases deben diseñarse pensando de antemano en la herencia ***o*** bien deben diseñarse prohibiendo la herencia. No hay punto medio.

D5.301 🕮 *Depth of Inheritance Tree (DIT)* se define como la longitud más larga de la cadena de herencia que termina en el módulo actual. Para lenguajes que no admiten herencia múltiple, es simplemente el número de ancestros :



S5.302 ☺ Un valor nominal de DIT está entre 0 y 3. Un DIT alto está directamente correlacionado con un número elevado de fallos (Basili 1995), (Glasberg 2000)

O5.303 ☞ Adicionalmente, existe una correlación positiva entre la profundidad a la que se encuentra una clase y la probabilidad de que existan fallos en la misma. Por ello, el grado de inspección y análisis debe ser tanto más detallado cuanto mayor sea esta profundidad (Basili 1995)

C5.304 💣 Las jerarquías de clases en las que existe una única clase hija son siempre sospechosas

P5.305 ☠ No se diseñarán jerarquías en las que existan elementos con DIT > 7

5.4 Nomenclatura

O5.400 ☞ Dado que las clases representan "cosas" y no "acciones", cada clase tendrá como nombre un nombre en el sentido gramatical, y nunca un verbo.

O5.401 ☞ Los nombres de clases que representan excepciones terminarán en el sufijo `Exception`.

O5.402 ☞ El nombre de una clase estará en singular salvo que una instancia de esa clase represente una multiplicidad de cosas

P5.403 ☠ El nombre de la clase no contendrá detalles sobre la implementación interna de la misma.

Por ejemplo, un nombre como `ArrayAlumnos` sería inapropiado.

5.5 Clases Anónimas

O5.500 ☞ Las clases anónimas deben utilizarse de manera limitada, y en ningún caso tener una longitud superior a media pantalla

P5.501 ☠ Una clase anónima no tendrá más de dos métodos.

6 ENUMERACIONES

6.1 Reglas generales

O6.100 ☞ El nombre de las enumeraciones estará en singular.

Por ejemplo:

```
public enum Ubicacion {
   ARCHIVO,
   BASE_DE_DATOS,
   INTERNET
}
```

O6.101 ☞ Los nombres de los miembros de una enumeración estarán en mayúsculas, separando las diferentes palabras por guiones bajos (_)

O6.102 ☞ En enumeraciones de tipo bitfield (**`Flags`**), el nombre de la enumeración estará en plural

Por ejemplo,

```
[Flags]
public Enum Estilo {
   NEGRITA,
   CURSIVA,
   SUBRAYADO,
   TACHADO
}
```

S6.103 ☺ En general suele ser buena idea dotar a la enumeración de una propiedad descriptiva que contenga una explicación legible sobre el valor de la propiedad . La explicación puede estar presente de forma directa, o bien mediante claves de recursos para aplicaciones internacionalizables.

7 RUTINAS Y MÉTODOS

7.1 Número de métodos y grados de responsabilidad

D7.200  

`Import Coupling (IC)` se define como el número de interacciones entre la clase actual y todas las clases con las que interacciona. Esta definición es similar a CBO, pero cuenta las interacciones individuales en lugar de las clases acopladas.

C7.201  

Existe una alta correlación entre IC y el número de fallos de una clase (Briand, y otros 1998). Por tanto, el número de dichas interacciones debe estar limitado.

O7.202 

El Número de métodos de una clase debe ser inferior a 40. (Rosenberg, Stapko y Gallo 1999)

7.2 Nomenclatura

S7.701  

Un buen nombre para una rutina es aquel que describe *todo* lo que la rutina hace.

S7.702  

Es recomendable que los nombres de **métodos que no devuelven valores ("procedimientos")** consistan en un verbo, seguido del objeto al que afecta el verbo. Por ejemplo: *ImprimirFichaInmueble* o *CalcularVAN*.

S7.703  

Para los nombres de **funciones o métodos que devuelven valores**, es recomendable que describan el valor devuelto. Por ejemplo: *ImpresoraLista*, o *VAN*.

C7.204  

En general es peligroso que un método incluya en su nombre detalles internos sobre el funcionamiento del mismo, o sobre las interioridades de la clase.

P7.205  

En ningún caso se usarán verbos genéricos "aplicables a todo" como: *Procesar*, *Manejar*, *Gestionar*, etc. Por ejemplo : *ProcesarInmueble*, *GestionarCliente*. En este caso, el verbo no aclara el cometido real del procedimiento o función.

P7.206  

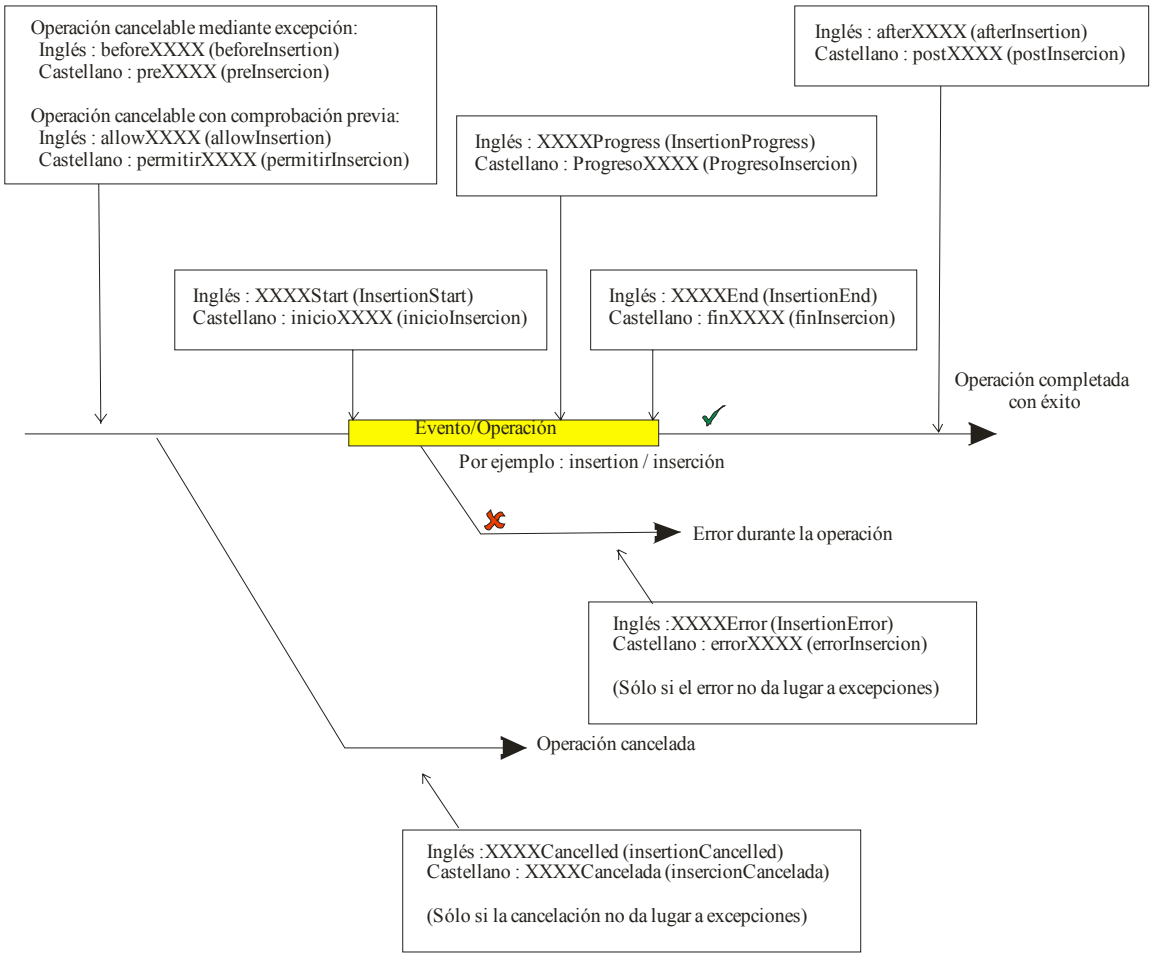
De forma análoga, en ningún caso se usarán nombres genéricos como Entrada, Datos, Salida, Informe, etc. Por ejemplo : *LeerDatos*, *ImprimirInforme*, *VisualizarSalida*. En este caso, el nombre qué tipo de informe ni qué tipo de salida se está produciendo, ni cómo deben ser los datos que se "leen".

O7.207 

Cuando existan grupos de funciones que realicen operaciones similares con pequeñas diferencias, se deberá establecer un sistema de creación de nombres coherente.

Por ejemplo, en el caso de introducción de datos se podría utilizar *Obtener* para una lectura destructiva y *Leer* para una lectura no destructiva. De esta forma *ObtenerMensaje* y *ObtenerCaracter* leerían un mensaje de la cola de mensajes o un carácter del buffer del teclado, destruyendo el dato original, mientras que *LeerMensaje* y *LeerCaracter* devolverían los mismos datos sin destruirlos.

O7.208 ☞ Los nombres de los métodos que representen gestiones relacionadas con la ocurrencia o progreso de un evento o un proceso seguirán las siguientes reglas:



En el diagrama anterior, la diferencia entre los sucesos ***pre*** e ***inicio*** por un lado y ***post*** y ***fin*** por otro es que los primeros no se consideran parte de la operación y por lo tanto un fallo en ellos no implica un fallo en la operación en sí, mientras que un fallo en los segundos conduce a considerar la propia operación como fallida.

7.3 Cohesión - Cohesion

D7.300   El término **cohesión** hace referencia a la fuerza con la que las distintas partes de una rutina están relacionadas entre sí. Por ejemplo, la función *sin()* es altamente cohesiva porque realiza una única operación. Por otro lado, la función *sinYlogatirmo()* es débilmente cohesiva porque realiza dos operaciones que no guardan entre sí ninguna relación.

Existen diferentes tipos de cohesión, algunos de los cuales son aceptables y otros no:

S7.301   **Cohesión Funcional**. Esta es la relación que existe cuando la rutina realiza una única operación. Por ejemplo: *Sin(), LeerNombreCliente(), ImprimirSubtotales()*, etc.

S7.302   **Cohesión Secuencial**. Esta es la cohesión que se da cuando la rutina contiene elementos que deben ser realizados en un orden específico o comparten datos. Por ejemplo, una rutina que abre un fichero, lee unos datos y calcula un total tiene dos pasos con cohesión secuencial : lectura de los datos y cálculo del total. Esta cohesión no es fácil de distinguir de la funcional. Habitualmente, si se puede dar a la rutina un nombre de tipo verbo-objeto, entonces probablemente la rutina tiene cohesión funcional y no secuencial.

S7.303   **Cohesión Comunicativa**. Esta es la cohesión que se da cuando la única relación entre dos pasos dentro de una rutina son los datos sobre los que operan. Por ejemplo, *LeerNombreYTelefonoCliente()* tiene cohesión comunicativa si el nombre y el teléfono del cliente forman parte de una estructura común de datos.

C7.304   **Cohesión Temporal**. Esta cohesión se da cuando las operaciones en una rutina se realizan al mismo tiempo en la escala del programa. Por ejemplo, *InicializarSistema()* es una rutina con cohesión temporal, ya que sus operaciones se realizan durante la fase de inicialización.

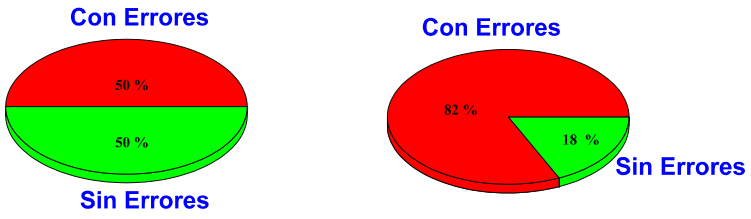
P7.305   **Cohesión Operacional**. Este tipo de cohesión se da cuando los pasos de una rutina se ejecutan en un orden determinado, pero a diferencia de la cohesión secuencial, en este caso las operaciones no se refieren a un mismo conjunto de datos. Por ejemplo, *ImprimirPagina1YPagina2*.

C7.306   **Cohesión Lógica**. Esta cohesión se da cuando una rutina puede efectuar diferentes operaciones dependiendo de un parámetro de control. Por ejemplo : *LeerDato(byte queDato)*. Habitualmente estas funciones tienen una estructura lógica como *Switch* o *If* que aglutina partes que no tienen nada que ver entre sí.

P7.307   **Cohesión Coincidental**. La cohesión coincidental es el caso en el que las diferentes operaciones de una rutina no tienen ninguna relación entre sí. Es un sinónimo a la ausencia total de cohesión.

O7.308  Una buena rutina debe tener un grado de cohesión alto.

IMPORTANCIA DE LA COHESIÓN



ALTA COHESIÓN **BAJA COHESIÓN**

Resultados comparativos sobre los errores contenidos en 450 rutinas con alta y baja cohesión

O7.309 ☞ Se aislarán en rutinas los siguientes aspectos de un programa:

- Áreas de cambio probable o frecuente.
- Operaciones con datos complejos.
- Segmentos de código con algoritmos o lógica compleja.
- Código que haga uso de particularidades del ordenador, sistema operativo o lenguaje.

S7.310 ☺ Las áreas de cambio probable o frecuente suelen ser:

- Dependencias del hardware.
- Entrada/Salida : tanto ficheros generados como salidas escritas.
- Áreas cuyo diseño fue especialmente difícil.
- Variables de estado, puesto que el número y conjunto de estados posibles se altera con frecuencia.
- Limitaciones en el tamaño de los datos.
- Áreas de la programación relacionadas con la estructura o requisitos de procesos empresariales. Por ejemplo, cualquier código relacionado con leyes, procedimientos, políticas empresariales, regulaciones, flujos de trabajo, etc.

7.4 Acoplamiento – Coupling

D7.400  

El *grado de acoplamiento* entre dos subsistemas (donde un subsistema puede ser algo tan pequeño como una rutina o tan grande como un programa completo) hace referencia al grado de interconexión entre las mismas, o a la cantidad de información y datos que uno de ellos necesita conocer del otro para operar correctamente. Así como la cohesión describe la fuerza con la cual están interrelacionadas las interioridades de un subsistema, el grado de acoplamiento describe la forma en que el subsistema se interrelaciona externamente con las demás.

Un grado de acoplamiento bajo significa que dos rutinas son relativamente independientes entre sí : la modificación de una de ellas no afectará a la otra. Por el contrario, un grado de acoplamiento alto significa que dos rutinas dependen la una de la otra.

Al igual que la cohesión, existen diferentes grados de acoplamiento:

S7.400  

Acoplamiento por datos simples. Este es el acoplamiento que existe cuando el único intercambio de información que existe entre dos rutinas es en una lista de parámetros no estructurados. Por ejemplo, si una función *Tangente(Angulo)* llama en su interior a *Seno(Angulo)* y *Coseno(Angulo)*.

S7.401  

Acoplamiento completo por datos estructurados. Análogo al caso anterior, se da cuando el intercambio de información es a través de una lista de parámetros que contiene datos estructurados y la mayor parte de los campos de los datos estructurados se usa. Por ejemplo, si una función *ImprimirInforme()* llama a *ImprimirCliente(Cliente)*, siendo *Cliente* un registro, e *ImprimirCliente* usa todos o gran parte de los campos del registro *Cliente*.

S7.402  

Acoplamiento simple por objetos. Un subsistema está acoplado de forma simple a un objeto si crea instancias de este objeto. En principio este acoplamiento no es problemático.

C7.403  

Acoplamiento incompleto por datos estructurados. Es el caso anterior, pero cuando solamente se usa una parte pequeña de toda la estructura. Por ejemplo, si *ImprimirCliente()* llama a *ImprimirNombre(Cliente)*. En este caso, la función *ImprimirNombre* solamente usa el nombre del cliente, pero sin embargo recibe como parámetro toda la estructura. Este tipo de uso es **solamente** permisible cuando se prevea ampliación del uso de los datos por parte de la rutina llamada.

C7.404  

Acoplamiento de control. Estrechamente relacionado con la cohesión lógica, este acoplamiento ocurre cuando una rutina llama a otra indicando como parámetro la clase de operación que debe realizar.

S7.405  

Acoplamiento de lectura-lectura por datos globales. Este acoplamiento ocurre cuando la relación entre dos rutinas consiste en que ambas leen del mismo conjunto de datos globales.

P7.406  

Acoplamiento de lectura-escritura por datos globales. Este acoplamiento ocurre cuando una rutina modifica datos globales, mientras que otra rutina lee los mismos.

P7.407 **`Acoplamiento semántico`**. Este acoplamiento ocurre cuando un módulo A hace uso de información no pública de otro módulo B en su interacción con el mismo.

P7.408 **`Acoplamiento patológico`**. Este tipo de acoplamiento ocurre cuando una rutina modifica el código o los datos locales de otra rutina.

O7.409 Una buena rutina debe tener un grado de acoplamiento bajo.

S7.410 No obstante, no debe caerse tampoco en la tendencia a evitar el acoplamiento por todos los medios posibles, especialmente en lugares donde de todas formas existe un contrato que define la interacción entre los subsistemas, sino únicamente en aquellos sitios susceptibles de cambios.

"It is not high coupling per se that is the problem; it is high coupling to elements that are unstable in some dimension, such as their interface, implementation, or mere presence ... But, if we put effort into lowering the coupling at some point where in fact there is no realistic motivation, this is not time well spent" (Larman 2004)

7.5 Codificación

No existe relación demostrada entre la longitud de una rutina y su eficiencia o número de defectos.

S7.500 ☺ Se recomienda que una rutina abarque no más de una página y media impresa (equivalente a 80 líneas o 4 pantallas de texto). En esta longitud se incluye todo lo que concierne a la rutina, como comentarios, líneas en blanco, etc.

P7.501 ☠ No se crearán rutinas superiores a las 200 líneas

S7.502 ☺ En general, si un bloque de código de una rutina o método necesita un comentario interior (inline), habría que plantearse si no sería mejor que dicho bloque se constituyese en una rutina aparte con una contracción del comentario como nombre.

P7.503 ☠ Cuando una parte de un método deje de ser útil o cambie su planteamiento, se eliminará totalmente del método, en lugar de dejarla comentada "por si acaso".

O7.504 ☞ Todo método debe ser usado por alguien.

7.6 Interfaz e Interacción con el exterior.

D7.600 🕮 Las **precondiciones** de un método son (Webre 2004) , (Meyer 1997) el conjunto de condiciones que deben darse a la entrada de un método.

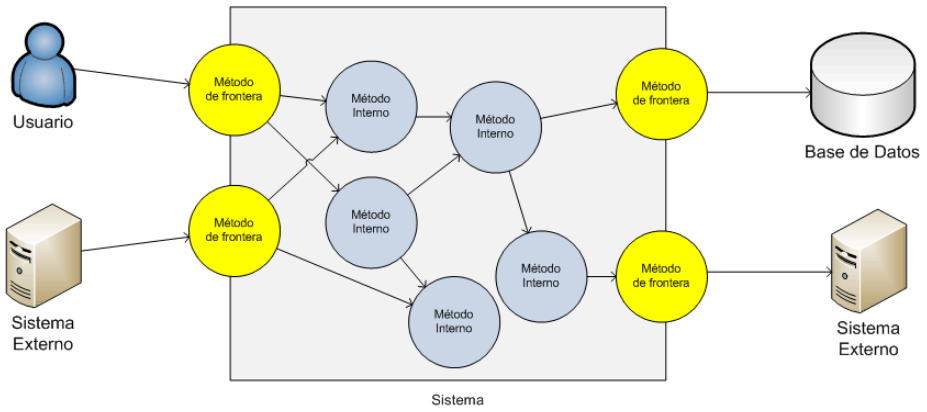
D7.601 🕮 Las **postcondiciones** de un método son el conjunto de condiciones que deben darse a la salida de un método.

S7.602 ☺ Una buena práctica es considerar, a todos los efectos, las cosas que un método ***no cambia*** como parte de las postcondiciones. (A diferencia de la costumbre habitual de considerar únicamente las cosas que sí cambia)

O7.603 ☞ (**Principio de no redundancia**) Las precondiciones y postcondiciones se comprobarán en un único lado (llamante o llamado) de la invocación (si bien el lado puede ser distinto para cada una de ellas).

O7.604 ☞ (**Principio de publicidad**) Cada elemento que figura en una *precondición* de un método debe ser accesible para todos los clientes para los cuales el método es accesible. Dicho de otra forma, una precondición no puede estar basada en elementos que no son visibles para los clientes del método. No es obligatorio el uso de este principio en las postcondiciones.

D7.605 🕮 Se define como método en la frontera aquellos cuya invocación está fuera del control del sistema, o que a su vez invocan métodos fuera del control del mismo:



C7.606 💣 Pese a que puede parecer lo contrario, ***no*** es una buena práctica que los métodos internos verifiquen explícitamente sus precondiciones, ni que los llamantes internos verifiquen las postcondiciones. (ver (Meyer 1997), por ejemplo)

O7.607 ☞ Todos los métodos de frontera a la entrada de un sistema verificarán sus precondiciones. En particular se verificarán siempre los siguientes datos:

- Datos que procedan del usuario.

- Datos que procedan de otros programas.
- Datos que procedan de ficheros que no fueron creados por el propio programa, especialmente ficheros de intercambio de datos entre aplicaciones.
- Datos que se hayan recibido a través de un canal de comunicación inseguro que pueda contener ruido, por ejemplo: una red global, una conexión entre módems, etc.
- Datos compartidos entre aplicaciones. Por ejemplo, datos en ficheros usados por más de un programa.

O7.608 ☞ Adicionalmente, los siguientes métodos verificarán sus datos siempre, con independencia de si están situados en la frontera del sistema o no:

- Rutinas de inicialización y finalización.
- Rutinas de grabación del trabajo o resultados.
- Rutinas que afecten al entorno, a la configuración del ordenador o a otros programas.
- Rutinas de tratamiento de errores o excepciones.

D7.602 🕮 Los **`invariantes`** de clase son el conjunto de expresiones que son ciertas en cualquier momento del ciclo de vida de una instancia de esa clase. Los invariantes de clase involucran únicamente el estado de la clase (sus propiedades)

D7.603 🕮 El **`contrato`** de un método es la descripción pública de lo que hace. Esta descripción pública incluye:

- La signatura del método (parámetros aceptados y resultados devueltos)
- Las precondiciones
- Las postcondiciones
- Excepciones lanzadas

D7.609 🕮 Un contrato se considera **`completo`** cuando el invariante y las precondiciones de un método identifican la totalidad de las entradas válidas al método, y cuando el invariante y las postcondiciones siempre detectan un estado incorrecto como resultado de la ejecución del método. (Benoit y Jézéquel 2006)

O7.610 ☞ Todas las funciones y métodos documentados deben incluir en su documentación las precondiciones y postcondiciones.

O7.611 ☞ Todas las clases deben documentar sus invariantes, o documentar explícitamente la carencia de los mismos.

O7.612 ☞ Todos los métodos que operen con cualquier tipo de E/S (ya sea flujos, invocaciones remotas, bases de datos,etc.) deberán documentar, como parte de su contrato, su sincronicidad y condiciones de cancelación (es decir, si operan de forma síncrona o asíncrona, y si es el primer caso, los tiempos de bloqueo síncrono por defecto)

O7.613 ☞ Todos los métodos que operen con datos bases de datos deberán documentar, como parte de su contrato, su comportamiento frente a transacciones, que puede ser uno de los siguientes:

Contrato	Significado
REQUIRES	Requiere la existencia de una transacción, bien pre-existente, bien exclusivamente creada para el método
REQUIRES NEW	Requiere la existencia de una transacción para uso exclusivo por parte del método
NEVER	No es ejecutable bajo una transacción. La existencia de una transacción previa constituye una violación del contrato
SUPPORTS	Indiferente, con el método uniéndose a la transacción si existe una
NONE	Indiferente - el método no participa en transacciones, con independencia de que haya o no una iniciada previamente.

S7.614 ☺ Los métodos de consulta en general no deben tener precondiciones.

O7.615 ☞ Las precondiciones se expresarán de forma que sean claras y *verificables objetivamente*. La siguiente precondición es un ejemplo **mal** expresado de una precondición: "*NumCliente contiene un código de cliente válido*". ¿Cómo se sabe que es válido?.

O7.616 ☞ En el uso de un método, no se realizarán ninguna clase de suposiciones salvo las que estén explícitamente documentadas en su contrato.

S7.617 ☺ En un caso ideal, el contrato de un método estará expresado de forma totalmente desacoplada respecto a su implementación (por ejemplo, mediante interfaces). En casos similares, se utilizará siempre la interfaz y nunca la implementación (principio de "**`Code To Interfaces and Not to Implementations`**")

D7.618 🕮 El **`principio de sustitución de Liskov`** (Liskov 1994) tiene que ver con las pruebas de corrección en jerarquías de objetos, y se formula de manera sencilla : Si P(x) es un predicado cierto para instancias x de un tipo T, entonces P(y) debe ser cierto para objetos y de S, donde S es un subtipo de T.

A nivel más práctico, el LSP tiene la forma de las siguientes reglas, aplicables a los métodos de S que sobrescriben métodos de T:

P7.619 ☠ Un método que sobrescribe a otro no puede aumentar las precondiciones de este último, donde por "aumentar" se entiende hacer más estrictas, imponer mas condiciones, etc. En particular, un método sobrescrito debe aceptar, como mínimo, todos los valores de parámetros que aceptaría el método original.

P7.620 ☠ Un método que sobrescribe a otro no puede relajar las postcondiciones de este último, donde por "relajar" se entiende eliminar postcondiciones, condicionar el cumplimiento de las postcondiciones, etc. En particular, un método sobrescrito no puede devolver valores fuera del conjunto de valores que devolvería el método original.

P7.621 ☠ Un método no puede disminuir los invariantes.

P7.622 ☠ Cuando un método sobrescribe a otro, sus excepciones deben ser un subconjunto (no necesariamente propio) de las excepciones del método sobrescrito.

O7.623 ☞ Cuando la precondición sea un requisito no - trivial hacia un parámetro, se utilizarán *aserciones* para verificarla. Si el lenguaje no dispone de una función o procedimiento *Assert()*, como parte **inicial** del desarrollo se creará una función de este tipo. "No-trivial" se define aquí como un requisito que, si no se cumple:

- Interrumpe el programa.
- Causa "escapes" o pérdidas de memoria.
- Afecta a otros programas.
- Afecta a datos globales o de configuración.
- Afecta a la misión general del programa: por ejemplo, un error que afecte a la pantalla no es crítico para una hoja de cálculo, pero sí lo es para un programa de tratamiento de imágenes.

P7.624 ☠ No se utilizarán aserciones como sustitución de comprobaciones de error. Por ejemplo, el siguiente código es incorrecto:

```
File f = new File("test.txt")
assert f.exists();
```

IMPORTANCIA DE LA INTERFAZ

Número de Errores debidos a fallos de comunicación entre dos rutinas

39%

D7.625 🕮 **`Ley de Demeter`** - (Lieberherr, Holland y Riel 1988), también llamada regla de "No hables con extraños". Para toda clase C y para todos los métodos M asociados a C, todos los objetos con los cuales M interactúa deben ser :

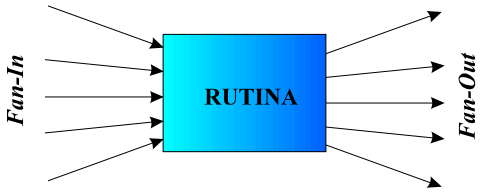
- Instancias de las clases a los que pertenecen los argumentos de M
- Instancias de C
- Objetos creados directamente por M, o por los métodos a los que M invoca.
- Objetos globales.

Otra forma de expresar la Ley de Demeter es mediante la analogía de los juguetes:

- Puedes jugar contigo mismo
- O con los juguetes que te han dado
- O con los juguetes que construyas tú.

S7.626 ☺ La Ley de Demeter es un principio útil que minimiza el acomplamiento entre clases. Sin embargo, su aplicación con celo excesivo conduce a un exceso de métodos en las clases intermedias. Debe considerarse como una guía similar a la del principio de ocultación de la información.

D7.627 🕮 El **`fan-in`** es un concepto que indica el número de rutinas que llaman a una determinada rutina. Por otra parte, el **`fan-out`** indica el número de rutinas que una rutina en concreto llama:



S7.628 ☺ Las características de un buen diseño son un fan-in *alto* y un fan-out *bajo o medio*.

7.7 Parámetros

O7.700 ☞ Los parámetros de una rutina deben estar ordenados como sigue:

- Primero los parámetros de entrada, o solo lectura.
- Después los parámetros de "transporte", o de lectura/escritura.
- Finalmente, los parámetros exclusivamente de salida.

O7.701 ☞ Si varias rutinas usan parámetros similares, el orden de los mismos en las listas de parámetros debe ser coherente. Por ejemplo, la mayoría de las funciones del API de Windows requieren un *Handle* como parámetro. En todas ellas, dicho parámetro es el primero de la lista.

O7.702 ☞ Se deben usar todos los parámetros pasados.

O7.703 ☞ Los parámetros - resultado tipo *Estado* o *Resultado* deben ser los últimos.

C7.704 💣 No se utilizarán los parámetros como variables de trabajo o temporales. Por ejemplo, el siguiente fragmento de código es inadmisible:

```
Procedure Ejemplo(Entrada : Integer; Var Salida : Integer);
Begin
  Entrada := Entrada + Funcion1(Entrada);
  Entrada := Entrada + Entrada / 2 + 1/3;
  ...
  Salida := Entrada;
End;
```

La única situación en la que se permite la modificación de los parámetros de entrada es para normalizar su valor como preparación a su uso:

```
public void emitirFactura(String razonSocial, ...) {
   razonSocial = razonSocial.trim().toUpperCase();
   ...
}
```

S7.705 ☺ El número máximo recomendado de parámetros en una rutina es 7. Un valor superior a este dificulta enormemente la comprensión de la rutina. En general este número es aplicable a otras situaciones. 7 ±2 es un límite experimental de la capacidad cognitiva, a partir del cual el grado de retención y comprensión de las cosas decrece rápidamente (Miller 1956)

S7.706 ☺ Es conveniente que las rutinas se diseñen de forma que contengan (limiten) los fallos ocasionados por los niveles inferiores, de forma que no sean propagados. Contención no significa que los mismos sean barridos bajo la alfombra o ignorados, sino que se pueda hacer algo respecto a ellos.

P7.707 Ninguna rutina se programará asumiendo un mecanismo de paso de parámetros concreto (convención C, Pascal, paso mediante registros, etc.).

P7.708 Ninguna rutina se programará asumiendo un mecanismo de llamada concreto (lejana, cercana, por medio de interrupción, etc.)

7.8 Valores de Retorno

O7.800 ☞ Todas las funciones que devuelvan estructuras de datos de cualquier tipo (listas, grafos, árboles, mapas, etc.) devolverán una estructura vacía para indicar ausencia de resultados y nunca `nil/null`.

O7.801 ☞ Cuando exista ambigüedad, debe documentarse en las rutinas que devuelven cadenas si el uso pretendido es para consumo humano (visualización, traza, envío de alertas, etc.) o para uso por parte de otros procesos. Por ejemplo, el uso de `toString()` en Java es claramente de consumo humano, mientras que el uso de `.trim()` es claramente para otros procesos.

P7.802 Nunca se utilizará el retorno de una rutina para consumo humano para hacer proceso y control de flujo en base al mismo. En particular, nunca se utilizará el resultado de `.toString()` (en un objeto) o de `.getMessage()` (en una excepción) para realizar decisiones de proceso o para extraer información que afecte al proceso en curso.

7.9 Métodos especiales

S7.900 ☺ Siempre es buena idea sobrescribir el método **`toString()`** proporcionando información compacta sobre el estado del objeto, *para uso humano*. En todos los casos esta información deberá incluir el nombre simple de clase (sin espacios de nombres, paquetes, módulos o similares, salvo cuando haya posibilidad de confusión).

S7.901 ☺ En el caso de objetos persistidos, el método **`toString()`** deberá incluir como mínimo la clave primaria del objeto. Si hubiese uno o más códigos de referencia que el cliente final utilizase para identificar el objeto, se incluirán todos ellos indicando explícitamente en el resultado cual es cual:

```
public class Alumno {
   ...

   public String toString() {
      return "Alumno "+getNombre()+" ID:"+getId()+ " DNI:"+getDNI();
      ...
```

S7.902 ☺ Las clases que representen primordialmente datos con una ordenación intrínseca deberían implementar **`Comparable`**. Sin embargo, esto no es buena idea cuando los criterios de ordenación no estén claros o cuando haya diferentes criterios igualmente válidos para ordenar. Por ejemplo, la clase Distancia debería implementar **`Comparable`**; la clase **`Punto3D`** o **`Alumno`** - no.

S7.903 ☺ En el caso anterior, cuando haya diferentes criterios de ordenación, suele ser buena práctica proporcionar comparadores con los diferentes criterios dentro de la propia clase, en forma de clases estáticas o anidadas que implementan **`Comparable`**. Las instancias de estas clases deberían crearse por métodos factoría de la clase principal:

```
public class Alumno {
   ...

   public static Comparator<Alumno> getComparadorPorNombre() {
      ...
   }
   public static Comparator<Alumno> getComparadorPorEdad() {
      ...
   }

   ...
}
```

O7.904 ☞ Si se sobrescribe el método **`.equals()`**, debe sobrescribirse igualmente el método **`.hashCode().`**

P7.905 Nunca se utilizará, dentro de equals() comparación *exacta* de tipos de datos:

```
public boolean equals(Object arg) {
   if (arg.getClass() == MiClase.class)
      ...
}
```

en su lugar, se utilizará **`instanceof`**. Pese a las insistencias de algunos en Internet, es perfectamente posible cumplir el contrato de **`equals()`** sin tirar a la basura el polimorfismo.

C7.906 En el cálculo de **`.hashCode()`**, no debe caerse en la tentación de excluir partes del objeto en aras a un mayor rendimiento.

P7.907 En los lenguajes con gestión de memoria integrada (Java, .NET, etc.) no se utilizarán los finalizadores.

O7.908 Después de la llamada a un constructor, un objeto debe ser utilizable inmediatamente, sin que existan acoplamientos de tipo (*"queremos tener un constructor predeterminado, pero también es necesario que el dato X esté inicializado para que se pueda operar con el objeto"*)

P7.909 No se utilizará **`new String("foo")`**, construcción que es válida en muy pocas ocasiones.

O7.910 Los métodos que modifican propiedades del objeto (por ejemplo **`setXXXX()`** en Java) deben poder invocarse en cualquier orden.

O7.911 El método **`clone()`** debe implementarse invocando a **`super.clone()`** en lugar de **`new`**.

```
public class Cliente {
  public Object clone() {
    Cliente c = (Cliente)super.clone();
    …
   }
}
```


8 MÓDULOS (PAQUETES Y NAMESPACES)

8.1 Creación

D8.100 📖 Todos los lenguajes modernos incorporan el concepto de módulo para aislar conjuntos de clases muy relacionadas entre sí y que colaboran para la consecución de un objetivo o funcionalidad común. Un módulo puede ser un paquete en Java, un namespace en .NET, un Unit en Delphi / Modula, etc.

O8.101 ☞ Se deberá seguir un enfoque modular y de ocultación de información en cualquier desarrollo. Este último principio quiere decir que la información debe estar muy cohesionada con las rutinas que la manejan y no ser visible desde el exterior.

P8.102 ☠ No se accederá a las interioridades de ninguna estructura abstracta de datos. Los siguientes ejemplos de código no son admisibles:

❶
```
Cliente := Cliente^.Next; { Lista enlazada }
```

❷
```
Cliente := Cliente^.HijoIzquierdo;{ Arbol binario }
```

O8.103 ☞ En lugar de la forma anterior, se crearán procedimientos o funciones para el acceso a los datos.

❶
```
Cliente := ObtenerSigCliente(Cliente);
```

❷
```
Cliente := ObtenerClSubordinado(Cliente);
```

S8.104 ☺ No se debe confundir los datos globales de un módulo con los datos globales de un programa. El uso de los primeros por las rutinas de un módulo aumenta la ocultación de información y la legibilidad general del código, además de prevenir el acoplamiento patológico entre módulos. El uso de los datos globales, por otra parte, es totalmente desaconsejable.

8.2 Nomenclatura

O8.200 ☞ La nomenclatura de los módulos o Namespaces será:

```
com.empresa.proyecto.componente1.componente2
```

donde componente2 será opcional. Componente1 podrá ser opcional únicamente en proyectos de muy poca entidad (hasta 15 clases)

9 TIPOS DE DATOS Y VARIABLES

9.1 Definición de tipos de Datos

P9.100  

Aunque el lenguaje lo permita, no se redefinirán tipos estándar. Por ejemplo, el siguiente fragmento es inadmisible:

```
typedef integer longint;
```

S9.101  

Cuando se busque portabilidad, es conveniente utilizar definiciones redundantes de los tipos estándar. Por ejemplo, definir *Entero* como *Integer* para que en caso de cambio (por ejemplo, de *Integer* a *LongInt*) baste modificar la definición del tipo *Entero*.

P9.102  

Aun cuando el lenguaje la permita, no se utilizará la declaración implícita. Esta es una forma declaración que es sinónimo a no declarar nada. Ocurre cuando el lenguaje no exige que todas las variables sean declaradas antes de ser usadas (por ejemplo, Visual Basic). En estos casos la variable se declara de forma *implícita* en el momento de su uso.

C9.103  

Se evitará en lo posible la modificación del tipo de una variable durante la ejecución, cuando el lenguaje permita esta posibilidad.

9.2 Nombres de Variables

S9.200  

Igual que en el caso de las rutinas, la longitud recomendada del nombre de una variable está entre 7 y 16 caracteres. Se dará a las variables un nombre de acorde a su *significado* y *nunca* acorde a su tipo.

C9.201  

Se procurará evitar el uso de variables con nombres como *i,j,k,n*, etc en bucles, independientemente de la longitud del bucle.

P9.203  

Salvo para lenguajes basados en punteros y con comprobación de tipos débil (Pascal, C), en la actualidad la **Notación Húngara** constituye una práctica obsoleta, no recomendada ni siquiera por Microsoft (Microsoft Corp. s.f.). Por lo tanto, su uso queda prohibido, limitándose únicamente a la letra mayúscula I para representar interfaces (ISerializable, IClonable, etc.)

O9.204 

Los nombres de macros se escribirán en mayúsculas, precedidos de un subrayado. Las diferentes palabras de una macro se separarán igualmente con subrayados. Por ejemplo: _MAX_EMPLEADOS

P9.205  

No se utilizarán nombres de variables que puedan ser ambiguos. Por ejemplo *Col* es un nombre ambiguo ya que puede ser columna o color. Una guía a seguir es si otra persona, diferente del autor del código, entiende la variable de la misma forma en un fragmento de código como el siguiente:

```
for (int colActual = 0; colActual <= maxCol; colActual++) {
   ...
}
```


9.3 Variables de Estado y Temporales

O9.300 ☞ Para designar variables de estado se utilizarán nombres que hagan referencia al objeto y estado concreto al que se refieran estas variables. Por ejemplo: *ImpresoraLista* en lugar de *EstadoImpresora*, o *Visualizar* en lugar de *EstadoPantalla*.

C9.301 Se desaconseja la presencia de *Estado* en un nombre de una variable de estado puesto que suele indicar un nombre no descriptivo. Por ejemplo, *EstadoImpresora, EstadoPantalla, EstadoSistema* dejan claro que son variables de estado pero no indican a qué clase de estado concretamente se están refiriendo.

C9.303 Se procurará evitar dar nombres sin sentido a variables temporales. Por ejemplo: *Temp, I, Tmp,*, etc.

C9.304 Se desaconseja el uso de variables de tipo booleano como variables de estado. En su lugar es preferible utilizar tipos enumerados o enteros, definiendo como constantes sus posibles valores.

9.4 Variables Booleanas

O9.400 ☞ Las variables booleanas deben tener nombres que sugieran respuestas o contenidos de tipo S/N. Por ejemplo: *Exito*, *Correcto*, *Realizado*, etc.

O9.401 ☞ Los nombres de las variables booleanas deben ser positivos. Por ejemplo : *Encontrado* en lugar de *NoEncontrado*, etc.

C9.402 Mejor No es recomendable el uso de variables booleanas para representar conceptos que solo puedan tomar dos valores, si estos dos valores no son verdadero o falso. Por ejemplo, no es recomendable utilizar **`boolean`** para representar el sexo de una persona, ya que nunca estaría claro cuando se ejecutaría lo siguiente:

```
// ¿Tienen descuento los hombres o las mujeres?
double descuento = sexo?0:0.5;

// ¿No pueden entrar los hombres o las mujeres?
if (sexo) {
   System.out.println("Entrada prohibida aquí. "+
      "Utilice su propia entrada");
   return;
}
```

O9.403 ☞ Se introducirán variables booleanas temporales cuando en una estructura de control (*if*, *case*, *while*, etc) la expresión sea excesivamente compleja.

P9.404 Peligro Cuando una variable booleana pueda contener marcas de ausencia de valor (nulls), no se intentará llegar a una "regla genérica" sobre cómo funcionan estos valores en expresiones booleanas ("true and null, false or null"), ni se intentará crear la enésima versión de 3VL o 4VL[2], sino que se resolverá cada situación funcional de forma *independiente y aislada*. Dicho de otra forma, cada situación en la que tengan que combinarse estos valores se aislará en un método aparte, documentando el comportamiento y los criterios seguidos en lugar de intentar generalizar.

[2] NVL = N-Valued Logic (Lógica de N valores). Por ejemplo, una base de 3VL podrían ser los valores ("verdadero", "falso","desconocido"). El problema de las NVL es que durante mucho tiempo se han intentado crear cuadros de verdad que fueran coherentes bajo ciertos puntos de vista (leyes de DeMorgan, reglas de implicación y equivalencia) y no se ha conseguido satisfactoriamente. El propio E .F.Codd publicó tres revisiones diferentes de su 4VL sin conseguirlo, por lo cual la probabilidad de que cualquier programador o analista sin una amplia formación matemática consiga hacerlo es cero. Ver, por ejemplo, (Date 2006)

9.5 Variables Artificiales

D9.500  

Son **variables artificiales** aquellas que nacen como consecuencia de la estructura de código de una función. Un ejemplo clásico es una variable "resultado" con la que se pretende que una función tenga un único punto de salida y que no se debe a una necesidad operativa de manejar el resultado de alguna forma (Roberts 1995). Por ejemplo, en la siguiente función que busca un número en una lista:

```
function Search(var list: IntList;
                n, key: integer) : integer;
 var
   i: integer;
   found: boolean;
 begin
   i := 1;
   found := FALSE;
   while (i <= n) and not found do
     if list[i] = key then
       found := TRUE
     else
       i := i + 1;
   if found then
     Search := i
   else
     Search := 0
 end;
```

la variable *found* es artificial. El empecinamiento a arrastrar el resultado de un cálculo hasta el final únicamente complica una rutina que podría haber sido infinitamente más sencilla:

```
function Search(var list: IntList;
                n, key: integer) : integer;
var
  i: integer;
begin
  for i := 1 to n do
    if list[i] = key then return i;
  return 0
end;
```

Este tipo de situaciones ya fueron exploradas por (D. Knuth 1974)

P9.501  

Dado que las variables artificiales dificultan la legibilidad general del programa, están prohibidas en cualquier situación para la cual exista alternativa.

S9. 502  

En la definición de funciones, es conveniente salir utilizando **return** en el momento en que se conozca el resultado que debe devolver la función, en lugar de andar arrastrando el resultado por toda la rutina con el objetivo de tener un único punto de salida[3]

O9.503 

Se utilizará siempre **break** en lugar de variables de finalización, salvo en situaciones en las que es necesario salir de más de un bucle

[3] La regla del "único punto de salida" tiene sus orígenes en un pasado muy anterior al establecimiento de la programación estructurada, por motivos muy diferentes a los que se supone. (Ver, por ejemplo, **http://forum.java.sun.com/thread.jspa?threadID=5194210&messageID=9894427**). Las circunstancias que motivaron dicha regla hoy en día ya no están presentes, pero la regla - que es un lastre para la legibilidad del código y obliga a complicaciones innecesarias - se niega a desaparecer.

9.6 Arrays y cadenas

S9.600 ☺ Es recomendable tratar los arrays (matrices) como estructuras *secuenciales*. Es decir, estructuras a las que no se puede acceder al elemento N si antes no se ha pasado por los elementos 1, 2,..., N-1. El acceso indiscriminado a los elementos de un array es parecido a los saltos indiscriminados utilizando *gotos*. Esta sugerencia es válida únicamente cuando los arrays sean homogéneos en cuanto a los datos que contienen, pero no si se utilizan como un sustituto del tipo de datos Registro si el lenguaje no lo tiene.

P9.601 Nunca se declararán longitudes de arrays con constantes literales:

```
Var   Empleados : Array [1..100] of Integer;
```

o

```
int[] empleados = new int[100];
```

O9.602 ☞ En lugar del código anterior se utilizarán siempre constantes con nombre

O9.603 ☞ Las cadenas en C/C++ se declararán con una longitud de *LONGITUD_DESEADA + 1* (debido al carácter 0). Por ejemplo:

```
char Nombre[_LONG_NOMBRE+1];
```

O9.604 ☞ Las cadenas de caracteres tipo C se inicializarán siempre a \0 de forma explícita.

9.7 Punteros

O9.700 ☞ Todos los punteros se inicializarán a **NIL** / **NULL** a menos que se les asigne un valor inmediatamente después de su declaración.

S9.701 ☺ Es recomendable añadir en todas las estructuras de datos dinámicas un campo de comprobación cuyo valor sea siempre fijo, de forma que se pueda comprobar en cualquier momento si el puntero a la estructura es válido. Por ejemplo:

```
Type  PCliente = TCliente;
   TCliente = Record;
      ID : Integer ; { Siempre $ACAC }
      Nombre : String[15];
      Apellidos : String[30];
      ...
    End;
```

S9.702 ☺ Como medida alternativa a la anterior, es conveniente comprobar la validez de los datos a los que apunta el puntero. Por ejemplo, si uno de los campos de TCliente es *edad*, se puede determinar la validez con la comprobación edad>0 and edad <=150.

S9.703 ☺ Aún otra medida alternativa a las anteriores es la redundancia en los datos. De esta manera, la verificación de los datos a los que apunta un puntero consiste en comprobar la igualdad de los datos redundantes. Por ejemplo:

```
   TCliente = Record;
      Nombre : String[15];
      Apellidos : String[30];
      Edad : Integer;
      ...
      EdadVerif : Integer;
    End;
```

S9.704 ☺ Es conveniente aislar la utilización de punteros en rutinas de más alto nivel.

P9.705 ☠ No se permite más de dos de-referencias por operando. Por ejemplo:

```
Lista*.Siguiente*.Anterior*.Anterior = NULL;
```

S9.706 ☺ Se recomienda que haya solamente una dereferencia por operando. Para conseguirlo, se utilizarán variables intermedias.

O9.707 ☞ Después de liberar un puntero, se asignará a NIL / NULL de forma explícita.

C9.708 💣 No se utilizará más de un nivel de referencia (es decir, no se utilizarán punteros a punteros, punteros a punteros a punteros, etc.), salvo que, o bien sea absolutamente necesario y **se**

justifique en el código, o bien lo imponga el sistema operativo (por ejemplo, Windows), el diseño o la plataforma.

C9.709  

Salvo cuando se trate de objetos, se evitará hacer conversiones de tipo de punteros.

S9.710  

Es conveniente tener reservada una zona de *memoria de emergencia*, que sirva para cerrar de forma ordenada el programa cuando el espacio de memoria que ha quedado impide la continuación normal. Esta zona de memoria de emergencia se reserva al principio del programa. Cuando se agota la memoria disponible, la zona se libera y se usa *exclusivamente* para grabar los datos y salir ordenadamente.

9.8 Inicialización y Uso

P9.800 No se utilizarán "significados escondidos" ni valores-marcador en las variables. Por ejemplo: "*PaginasImpresas* es el número de páginas que fueron impresas o -1 si hubo algún error en la impresión".

S9.801 Cuando existan datos globales, se recomienda el uso de rutinas de acceso a las mismas en lugar de usar las propias variables. Así, en lugar de *Pagina++* se usaría una rutina *AumentarPaginas*, que sería la única que tuviese acceso directo a *Paginas*.

P9.802 No se utilizarán valores directos en los programas. Cualquier número o cadena constante deberá ser sustituida por una constante con nombre, salvo los siguientes valores:

- 0 (utilizado como elemento neutro)
- +1, -1 (utilizados como elementos neutros o deltas)
- +2,-2 (utilizados *exclusivamente* como deltas en iteraciones alternadas)
- true / false
- ""

Por ejemplo:

```
velocidad = velocidad + 1;     // aceptable : 1 utilizado como delta
frecuencuia = 1/periodo;       // aceptable : 1 utilizado como elemento neutro del operador *
peso = 1;                      // no aceptable : significado semántico asociado a 1
peso = peso -1;                // aceptable : -1 utilizado como delta
peso = 0;                      // aceptable : 0 utilizado como elemento neutro
indiceMatriz += 2;             // aceptable : 2 utilizado para acceder al siguiente elemento
                               // par o impar
velocidad += 2;                // inaceptable : 2 utilizado como variación con semántica.
```

S9.803 Es conveniente que las divisiones estén protegidas contra el error en el que el denominador es cero.

O9.904 Cualquier variable, parámetro o valor de retorno que en un momento dado pueda valer NULL deberá documentar *clara y explícitamente* qué significa un NULL.

Dos significados ampliamente reconocidos de null *que conducen a tratamientos funcionales diferentes* pueden ser:

- Desconocido (También llamado A-Mark por (Codd 1990))
- No Aplicable (También llamado I-Mark)

P9.905 No se crearán rutinas que acepten o devuelvan nulls con signfiicados polivalentes. Por ejemplo:

```
// Devuelve el sueldo de una persona. Devuelve 0 si esa persona es un
// becario en prácticas, o null si esa persona no tiene sueldo por no pertenecer
// a la empresa, o si no se encuentra en la base de datos
//
public BigDecimal getSueldo(String nombrePersona) {
...
}
```


9.9 Otras reglas

D9.900  

Una variable asociada a un contexto es una variable que es compartida por varios subsistemas por el hecho de que estos subsistemas comparten el contexto de ejecución en el que se encuentra esa variable. Por ejemplo, variables asociadas a contexto son las variables de tipo **`threadlocal`** (variables asociadas a la hebra actual), variables asociadas a **`request`** o a **`session`** en una aplicación web, etc.

O9.901 

Las variables asociadas a contexto deben documentarse como si se tratase de variables globales a la aplicación. En particular, deben documentarse en una sección específica de la documentación.

P9.902  

No utilizar **`float`**. Salvo en aplicaciones para entornos con capacidad de proceso limitada o procesadores con tamaño de palabra inferior al habitual (por ejemplo : móviles, sistemas embebidos, etc.), no utilizar tampoco **`byte`** o **`short`**.

P9.903  

Cuando se esperen respuestas exactas, no utilizar tipos de datos de punto flotante (**`float`**/**`double`**/**`real`**, etc.). Utilizar tipos de datos de precisión fija (si el lenguaje dispone de ellos, por ejemplo **`decimal`** en .NET) o tipos de datos de precisión ilimitada (por ejemplo **`BigInteger`**/**`BigDecimal`** en Java)

O9.904 

Las variables de tipo colección o flujo se declararán utilizando el tipo de datos de la interfaz o la clase ancestro, no utilizando el tipo de datos concreto. Por lo tanto, se utilizará

```
InputStream in = new FileInputStream("foo.txt");
List<String> lista = new ArrayList<String>();
```

frente a

```
FileInputStream in = new FileInputStream("foo.txt");
ArrayList<String> lista = new ArrayList<String>();
```

Se aplicará el mismo criterio en la declaración de parámetros y en la declaración de tipos de retorno.

P9.905  

No se utilizarlá la concatenación con una cadena (""+a) como forma de conversión rápida de un tipo de datos a **`String`**. En su lugar, se utilizará **`String.valueOf()`** o un método más específico, si existiese (**`Integer.toString`**, por ejemplo, para enteros)

10 ESTRUCTURAS DE CONTROL

10.1 Flujo Lineal

O10.100 ☞

Cuando un grupo de sentencias deben ejecutarse en un orden concreto, la dependencia del orden deberá ser obvia. Por ejemplo, en el siguiente fragmento

```
Inicializar();
InicializarImpresora();
```

suponiendo que Inicializar() declara variables globales que son usadas por InicializarImpresora(), la dependencia no es visible y se podría pensar que las rutinas son independientes o que pueden ser llamadas en otro orden. Se debería sustituir por:

```
InicDatosGlobales();
InicImpresora();
```

Observar también que, de acuerdo con lo establecido en la sección referente a nombres de rutinas, se ha modificado el prefijo de *ambas* rutinas para mantener la coherencia, en lugar de hacer simplemente:

```
InicDatosGlobales();
InicializarImpresora();
```

O10.101 ☞

El código debe ser legible secuencialmente de arriba hacia abajo. Cuando las llamadas a las funciones no dependan entre sí, se agruparán a nivel de datos tratados y no a nivel de operación.

El siguiente fragmento muestra una agrupación a nivel de operación (los tipos de operaciones iguales están juntos):

```
InicDatosMensuales()
InicBalance()
InicCuentaResultados()

CalcDatosMensuales()
CalcBalance()
CalcCuentaResultados()
```

El siguiente fragmento muestra la agrupación a nivel de datos (las operaciones referidas al mismo objeto o conjunto de datos están juntas):

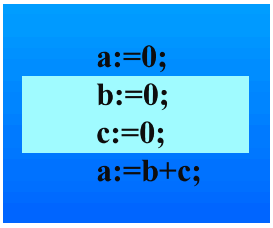
```
InicDatosMensuales()
CalcDatosMensuales()

InicBalance()
CalcBalance()

InicCuentaResultados()
CalcCuentaResultados()
```

D10.102 🕮

Se define la **apertura** de una variable como el número de líneas de código entre dos referencias consecutivas a la variable. Por ejemplo:



Ventana de apertura de a

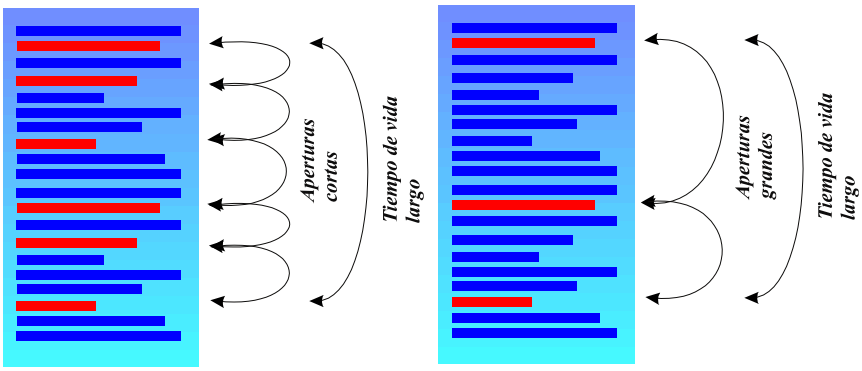
S10.103 ☺ Se debe procurar mantener baja la apertura media de cada variable en un módulo.

O10.104 ☞ En los lenguajes que permitan la declaración de variables en cualquier punto del código, se definirán las variables locales lo más cerca de su primer uso. Se evitará explícitamente el anti patrón (de la época de COBOL) de "diccionario de datos".

D10.105 🕮 Se define el *tiempo de vida* de una variable como la distancia en líneas de código entre su inicialización y su destrucción, o si esto no es aplicable, entre la primera y la última referencia a la variable.

S10.106 ☺ Se debe procurar que el tiempo de vida de las variables sea pequeño.

No confundir tiempo de vida con apertura:



10.2 Flujo Condicional

P10.200 No se utilizará un nivel de anidación superior a 4, salvo en condicionales de la forma

```
If .... Then
   ...
Else If ... Then
   ...
Else If ... Then
   ...
Else If ... Then
   ...
Else ...;
```

P10.201 No se utilizarán cláusulas *Then* vacías:

```
   If ... Then
   Else .... ;
```

O10.202 Las expresiones condicionales complicadas deben ser simplificadas usando llamadas a funciones booleanas o utilizando variables booleanas temporales.

O10.203 En una expresión booleana que enumere posibles casos, los casos más frecuentes deberán ir al principio:

```
   c:=ObtenerCaracter;
   If EsLetra(c) or EsSgnPuntuacion(c) or EsDigito(c) ...
```

Alternativamente:

```
   c:=ObtenerCaracter;
   Case c Of
      ´A´..´Z´ : ....;
      ´.´,
      ´!´,
      ´?´: ... ;
      ´0´..´9´: ... ;
   End;
```

O10.204 Salvo justificación expresa, se utilizará siempre evaluación booleana en cortocircuito (*short-circuit boolean evaluation*)

P10.205 En lenguajes en los que exista el operador condicional (C,C++,Java), no se utilizarán operadores ?: anidados.

P10.206 No se intentarán usar variables falsas o malabarismos para utilizar la sentencia *case* en lugar de una sucesión de *if*s. Por ejemplo, el siguiente código es inadmisible:

```
comando = leerComando();
switch (comando.charAt(0)) {
   ...
   case 'c': Copiar();break;
   case 'i': Imprimir();break;
   ...
}
```

P10.207 No se utilizará el *fall-through* en un *switch*. Si dos casos de un *switch* se repiten, se repetirá el código. Si este es muy grande, entonces su sitio es en una rutina. El siguiente fragmento de código es inadmisible:

```
switch (Token) {
   case Token1 :        ....;
         ...;
   case Token2 :...;
            break;
}
```

O10.208 Si el lenguaje permite tipos enumerados, los switch existentes que operen sobre los mismos deberán considerar, de forma explícita, todos los valores del enumerado. Si el entorno dispone de la opción de notificar avisos o errores en caso de omisión, esta opción se activará.

P10.209 No se utilizarán comparaciones explícitas con booleanos, tales como:

```
      if (a == true) ...

      if ( ( a > b ) == false )...
```

O10.210 En estructuras de control de tipo *if-then-else*, el caso positivo deberá ir primero. Así, se utilizará

```
if (ImpresoraLista) {
      ...
} else {
   ...
}
```

en lugar de

```
if (!ImpresoraLista) {
      ...
} else {
   ...
}
```

10.3 Flujo iterativo (bucles)

S10.300 ☺ Es recomendable tratar el interior del bucle como una rutina o caja negra, aplicando todo lo dicho sobre las rutinas en cuanto a cohesión y acoplamiento con el resto del código.

S10.301 ☺ Las condiciones iniciales y de terminación del bucle deben ser claras por su declaración, sin necesidad de examinar su interior:

```
while( codigo == rs.codPostal && !rs.eof()) {
   cuerpo opaco
}
```

C10.302 💣 Sin embargo, la regla anterior tiene como excepción aquellas situaciones en las que sea necesario introducir variables artificiales o aumentar la complejidad del cuerpo del bucle con el único propósito de situar la condición al principio. En estos casos, es preferible utilizar un bucle sin condición (while (true)) y salir del bucle en el punto donde más convenga, de manera natural.

S10.303 ☺ Cuando los bucles sean infinitos, es preferible indicar cuanto antes este hecho. Así, es preferible *While True* a *Repeat ... Until False*.

C10.304 💣 Se deben evitar bucles con cuerpos vacíos salvo en casos muy triviales que puedan ser leídos como frases, por ejemplo *Repeat Until Keypressed*.

D10.305 🕮 Se define *apertura* del bucle como el número de líneas de código que abarca.

O10.306 ☞ Se debe mantener la apertura de los bucles baja.

O10.307 ☞ La longitud máxima de un bucle en líneas (no confundir con líneas de código, que no incluyen líneas blancas ni comentarios) no podrá ser superior a una página impresa (66 líneas) o 3 pantallas

C10.308 💣 Se debe evitar código en el interior de un bucle *for* que dependa de un valor particular del índice de control. Por ejemplo:

```
for (int posicion = 1; posicion < lista.size(); posicion++) {
   ...
   if (posicion == POSICION_ESPECIAL)
      cosas
   ...
}
```


P10.309 No se utilizará la variable de control de un bucle *for* fuera del mismo.

S10.310 En bucles que trabajan con estructuras abstractas de datos o con flujos de datos en lenguajes sin gestión automática de la memoria, es conveniente considerar un *contador de seguridad* que finalice el bucle en el caso de que los datos estén corruptos. Por ejemplo:

```
int contadorSeguridad = 0;
while( cliente != null) {
   ...
   cliente = cliente.siguiente();
   contadorSeguridad++;
   if (contadorSeguridad > MAX_CLIENTES)
      throw new SomeException(....)

}
```

S10.311 Es recomendable que las sentencias de continuación del bucle (*loop* o *continue*) estén lo más próximamente posible al inicio del mismo

P10.312 No se utilizará un nivel de anidación de bucles superior a 3.

P10.313 No se utilizarán bucles con más de una variable de iteración:

```
for (int a=3,b=8; ... ; a++,b+=2)
```

S10.314 La inicialización de la variable de control de un bucle for se realizará dentro del mismo, y no con anterioridad. Es decir, se utilizará

```
for (int a = foo(); .... ; ... )
```

frente a

```
int a = foo();
for ( ; ... ; ... )
```


10.4 Otros tipos de flujo

C10.400 Se evitará el uso de la recursión cuando la solución iterativa sea igual de *legible*. La legibilidad será el único criterio a considerar salvo en situaciones donde el propio programa tiene consideraciones de otro tipo (rendimiento, tamaño en memoria, etc.). Para decidir entre la variante recursiva y la variante iterativa de una rutina nunca podrá aplicarse un criterio distinto al criterio general para el programa

D10.401 Se define como *apertura* de la recursión el número de rutinas intermedias hasta que se cierra el bucle recursivo. Por ejemplo, si A llama a B, B llama a C y C llama a A, la apertura es de 2 (B y C).

P10.402 No se utilizará una apertura superior a 2.

O10.403 En algoritmos recursivos se deberá controlar que exista suficiente espacio de pila.

10.5 Métricas

D10.600 🕮 Se define (informalmente) la *complejidad ciclomática* o *complejidad de McCabe* como el número posible de caminos distintos en un programa (McCabe y Butler 1989), (Software Engineering Institute, Carnegie-Mellon 2000)

Una forma de calcular la complejidad de McCabe de un programa es la siguiente:

- Comenzar con una complejidad de 1
- Añadir 1 por cada *if,while,do, repeat,loop, for, and, &&, or, ||, xor, ?:*
- Añadir 1 por cada caso en una estructura *case/switch*. Si la estructura no tiene *else/default* (o el equivalente), añadir otro 1 más.

El resultado se evalúa como sigue:

	Complejidad	Significado
☺	0 - 7	La rutina tiene una complejidad aceptable
💣	8-15	Sería conveniente simplificar la rutina
☠	15+	La rutina debe ser partida

11 EXCEPCIONES

11.1 Creación

O11.100 ☞ Los métodos crearán excepciones cada vez que sean incapaces de cumplir su contrato, y en particular cada vez que:

- Detecten violación de las precondiciones o invariantes de clase
- Sean incapaces de finalizar cumpliendo las post-condiciones
- Sean incapaces de finalizar sin violar los invariantes de clase

P11.101 Bajo ningún concepto se utilizará el mecanismo de excepciones como un mecanismo de control de flujo, en sustitución de la comprobación adecuada de límites, parámetros y valores de datos. En particular, nunca se capturarán excepciones de desbordamientos de índices, referencias nulas o similares (en Java : ninguna excepción estándar que descienda de **`RuntimeException`**).

P11.102 Nunca se lanzarán o se declarará en la definición de un método el lanzamiento de excepciones genéricas como **`Throwable`**, **`Exception`** o **`Error`**.

O11.103 ☞ Las excepciones que se originen por error en datos reproducirán en su información de detalle y su mensaje los datos que causaron el error. Si existen dependencias externas (por ejemplo, límites que no se han respetado), se incluirán en la medida de lo posible estas dependencias. Por ejemplo, se utilizará

```
if (fechaFirmaSolicitud.after(hoy))
  throw new SomeException("La fecha de firma de la solicitud
("+fechaFirmaSolicitud.getTime()+")"+
   "no puede ser posterior a la actual
("+Calendar.newInstance().getTime()+")");
```

en lugar de:

```
if (fechaFirmaSolicitud.after(hoy))
  throw new SomeException("La fecha de firma de la solicitud no puede ser
posterior a la actual");
```

O11.104 ☞ Las excepciones que se originen deben corresponderse con el nivel de abstracción del método o clase que las produce. Por ejemplo, si una clase tiene un método **`guardar()`** que almacena el en "algún lado" (opaco al cliente de la clase), es un error que este método propague hacia arriba una **`SQLException`** si ha decidido que el almacenamiento debe ser en una base de datos. En su lugar, debe crearse una excepción propia de la aplicación que encapsule la causa subyacente:

```
public void guardar() {
   try {
      … operaciones con bases de datos…
   } catch (SQLException e) {
      throw new AlmacenamientoException(e);
   }
}
```

P11.105  

Como generalización del caso anterior, nunca se utilizarán *checked exceptions* que revelen detalles sobre el funcionamiento interno de un método. Esta forma de programar es un ejemplo clásico de *leaky abstractions* (Spolsky 2002)

O11.107 

Nuevamente como corolario al caso anterior, si varias excepciones de tipo *checked* significan lo mismo para el cliente, deberán ser encapsuladas con una única excepción. Es decir, utilizar

```
   public void leerConfiguracion(String archivo) throws
ErrorConfiguracionException
```

en lugar de

```
   public void leerConfiguracion(String archivo) throws
      URLMalformedException, IOException, SAXParseException,
FactoryConfigurationError
```

S11.108  

En la medida de lo posible, y para facilitar la recuperación, una operación fallida que lanza una excepción debe dejar al objeto en su estado inicial, antes del inicio de la operación.

11.2 Captura y Tratamiento

S11.200 ☺

Un método sólo debería capturar una excepción si piensa darle algún tipo de tratamiento o si va a intentar recuperar o reintentar la operación. "Tratamiento" en este caso no significa imprimir la excepción en consola y olvidarse de ella:

```
try {
   .. operaciones que pueden fallar ...

} catch (AlgunaExcepcion e) {
   e.printStackTrace();
}
```

P11.201

Nunca se utilizarán bloques catch vacíos

```
try {
   ...
} catch (AlgunaExcepcion e) {}
```

ni sus equivalentes:

```
try {
   ...
} catch (AlgunaExcepcion e) {
   e.printStackTrace();
}
```

que no hacen nada, ni procesan la excepción, ni producen ninguna clase de traza persistente en ningún lado.

Las únicas excepciones permitidas a esta regla son en situaciones donde de forma *garantizada* no se producirá la excepción, como por ejemplo en:

```
URL url = null;
try {
   url = new URL("http://www.microsoft.com");
} catch (MalformedURLException ex) {
   // no puede ocurrir
}
```

y aún en esos casos es recomendable añadir un assert false.

C11.202

Salvo muy contadas ocasiones, no es procedente ni adecuado lanzar y capturar una excepción dentro del mismo ámbito. Casi siempre esto indica que se están utilizando las excepciones como mecanismo de control de flujo:

```
try {
   ...
   if (...)
      throw new AlgunaExcepcion();
   ...

} catch (AlgunaExcepcion e) {
   ...
}
```

P11.203 Nunca se utilizará un **`return`** en una cláusula **`finally`** ni se lanzarán excepciones desde estas cláusulas.

O11.204 Todos los recursos cuya gestión no realice de manera automática por el entorno o la librería se liberarán en cláusulas **`finally`** o **`using`** (en .NET o si el lenguaje dispone de esta construcción) En todos los casos esto incluye archivos, flujos de entrada/salida, conexiones con diferentes servicios, sockets, etc. Además, en los lenguajes sin gestión automática de la memoria esto incluye a todas las zonas de memoria reservadas durante el proceso. Los bloques de liberación de recursos deberán comprobar si el recurso ha sido inicializado.

C11.205 Como regla general, en aquellas plataformas que admitan excepciones anidadas (Java 1.4+, .NET) ***O*** se registra la excepción en una traza ***O*** se propaga, pero no ambas:

```
try {
   ...
} catch (FooException e){
   logger.error(e);            // O ESTO
   throw new BarException(e);  // O ESTO OTRO
}
```

P11.206 ***Salvo*** que esté explícitamente en el contrato del método, null no es un sustituto para una excepción, *en ningún caso*:

```
try {
   ...
} catch (FooException e){
   logger.error(e);
   return null;                // AAGHH!
}
```

P11.207 **Nunca** se sustituirá la anidación real de excepciones por la pseudoanidación consistente en quedarse con el mensaje de la excepción, olvidándose de lo demás

```
try {
   ...
} catch (FooException e){
   throw new BarException("Esto… error! "+e.getMessage());
}
```

S11.208 Siempre que sea posible, ante una excepción irrecuperable se intentará por todos los medios que el trabajo del usuario sea almacenado en algún lado de alguna forma (aunque no fuera la manera inicialmente prevista ni en el formato inicialmente previsto).

Por ejemplo, si un programa no es capaz de almacenar un formulario introducido por el usuario en una base de datos, la opción no es mostrar el error y dejarle plantado en el mismo formulario sin más alternativas que seguir reintentando la grabación o perderlo todo. El programa debería guardar automáticamente los datos (en binario, en XML, etc.) en un sitio confiable (un archivo, por ejemplo) e informar al usuario sobre este hecho.

Como continuación de esta regla, siempre que sea posible, el programa debe ofrecer opciones para importar estos datos más adelante y reintentar la petición.

11.3 Clases de excepción propias

P11.300 Nunca se crearán clases de excepción que desciendan directamente de `Throwable`.

C11.301 No se crearán clases de excepción que dupliquen excepciones existentes del sistema si o que se desea expresar es lo mismo. Por ejemplo, en Java ya existen las siguientes excepciones para situaciones frecuentes:

Clase	Situación
IllegalArgumentException	Un parámetro o dato de entrada tiene un valor incorrecto según las precondiciones.
IllegalStateException	Se está invocando a un método estando el objeto en un estado no apropiado
NullPointerException	Un dato de entrada tiene un valor de `null`, cuando esto es contrario a las precondiciones.
IndexOutOfBoundsException	Un índice está fuera de rango
UnsupportedOperationException	El objeto no admite este tipo de operación
ArithmeticException	Error en una operación aritmética

O11.302 Las clases de excepción que incluyan propiedades adicionales deben sobrescribir el método `toString()` o equivalente y volcar *la totalidad* de estas propiedades adicionales en la cadena resultante.

12 ORDENACIÓN Y ESTILO

12.1 Reglas Generales

O12.100 ☞ Se seguirá una convención coherente sobre:

- Capitalización de palabras clave
- Capitalización de variables locales, variables globales y constantes.
- Nivel de sangrado (número de espacios usados por cada nivel)
- Ordenación de las declaraciones de tipos compuestos (registros, objetos, etc.)
- Ordenación de las sentencias *case*.

P12.101 Bajo ningún concepto se seguirá un estilo de codificación que induzca a interpretaciones erróneas sobre el código, como por ejemplo:

```
x =  2+3 * 4+5;
```

P12.102 La longitud de una línea no podrá superar la anchura de una página impresa (80 columnas).

S12.103 ☺ Se recomienda utilizar espacios en blanco en los siguientes casos:

- Antes y después de operadores : *a + 3* en lugar de *a+3*.
- Después del corchete inicial y antes del corchete final de selección de elemento de un array : *a[3]* en lugar de *a[3]*.
- Después del paréntesis inicial y antes del paréntesis final de llamada a una función o procedimiento: *Imprimir(Informe)* en lugar de *Imprimir(Informe)*.
- Después de las comas que separan parámetros en una llamada a una función o un procedimiento: *Imprimir(a, b, c)* en lugar de *Imprimir(a,b,c)*.
- Antes y después del operador de asignación: *a := b* en lugar de *a:=b*.
- Antes y después de un paréntesis en una expresión: *(2 + 3)* en lugar de *(2 + 3)*

12.2 Alineación y Sangrado

P12.100 Los sangrados se realizarán siempre con tabuladores y nunca con espacios.

O12.101 Se sangrará un único tabulador por nivel. Los diferentes editores permiten configurar posteriormente el tamaño en caracteres de ese tabulador

O12.102 Se deberá usar uno de los dos estilos siguientes de sangrado:

a) Bloques puros:

```
if (A > 3) {
   ...
}
```

b) Bloques begin-end:

```
if (A > 3) {
   ...
   ...
   }
```

S12.103 Se recomienda el sangrado por bloques puros. En Java, este sangrado es obligatorio dado que coincide con la recomendación oficial de Sun (Sun Microsystems 1999).

C12.103 Se tolera el uso de cláusulas **Then** continuadas en la misma línea cuando no excedan la anchura de la página:

```
Pascal:     If A > 3 Then WriteLn("A es mayor que 3");

Java:    if ( a > 3 ) System.out.println("A es mayor que 3");
```

S12.104 En las declaraciones de tipos compuestos, se recomienda sangrar respecto al inicio de la línea y no respecto a la palabra clave o el nombre de tipo. Es decir, usar:

```
Type TListaClientes = Record
  Cliente : PCliente;
  Siguiente : PListaClientes;
End;
```

en lugar de:

```
Type TListaClientes = Record
           Cliente : PCliente;
           Siguiente : PListaClientes;
         End;
```

o de :

```
Type TListaClientes = Record
   Cliente : PCliente;
   Siguiente : PListaClientes;
     End;
```

12.105 ☺ En las sentencias *switch*, se recomienda sangrar respecto a la palabra que selecciona la acción en lugar de respecto al inicio de la acción. Es decir, usar:

```
Case Color of
  clRojo : Begin
   WriteLn(´Color rojo´);
   ...
   End;
  clVerde : Begin
   ...
```

en lugar de:

```
Case Color of
  clRojo : Begin
         WriteLn(´Color rojo´);
         ...
            End;
```


12.3 Sentencias Continuadas a más de una línea

O12.300 ☞ Cuando una sentencia tenga que partirse en más de una línea, el punto de partición se elegirá de forma que quede claro que la primera línea tiene continuidad en la siguiente. Por ejemplo:

```
TotalSemestre = Enero + Febrero + Marzo +
   Abril + Mayo + Junio;
```

en lugar de

```
TotalSemestre = Enero + Febrero + Marzo
  + Abril + Mayo + Junio;
```

O12.301 ☞ Solo se escribirá una sentencia por línea.

12.4 Agrupamiento

O12.400 ☞ Los trozos de código dentro de una rutina o bloque deberán separarse con líneas en blanco cuando el objetivo o intención de cada uno de ellos sea diferente.

O12.401 ☞ En las sentencias *If*, *While*, *Until*, cuando las expresiones condicionales sean muy largas, se agruparán de forma que cada sub-condición esté en una línea aparte, sangrada al nivel del inicio de la primera condición:

```
If (Caracter >= 'a' ) and (Caracter <= 'z') or
   (Caracter >= '0' ) and (Caracter <= '9') Then
```

P12.402 ☠ En las declaraciones de variables, no se harán agrupaciones por tipos. Es decir, cada variable deberá estar con su tipo. Por ejemplo, el siguiente código no es admisible:

```
    Persona comprador, vendedor, intermediario;
```

debiendo sustituirse por:

```
    Persona comprador;
    Persona vendedor;
    Persona intermediario;
```

S12.403 ☺ En una rutina, los casos especiales es recomendable que se traten solo una vez. Por ejemplo el siguiente código trata el mismo caso particular varias veces:

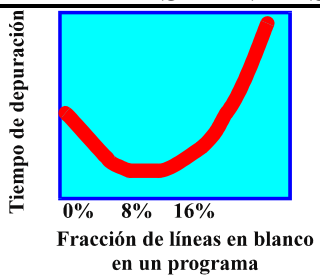
```
Procedure LeerComando;
...
   If Comando = CMD_COMANDO_VACIO Then Begin
      ...
   End.

   { Código común }

   If Comando = CMD_COMANDO_VACIO Then Begin
      ...
   End;

   { Más código común }
End;
```

IMPORTANCIA DE LAS LÍNEAS EN BLANCO



12.5 Comentarios

O12.500 ☞ Los comentarios deberán tener el mismo nivel de sangrado que el código que comentan.

D12.501 🕮 La métrica CP (Comment Percentage) se define como el número de líneas de comentario dividida por el número de líneas no blancas de código. Un nivel apropiado para esta métrica está en torno al 20% (Rosenberg y Hyatt 1995)

O12.502 ☞ Los comentarios deben escribirse de forma que sean fáciles de modificar. Por ejemplo, los siguientes comentarios son difíciles de cambiar:

```
/*******************************************************
 *                                                     *
 * Este programa realiza un cálculo del Valor Actual   *
 * Neto de la promoción                                *
 *                                                     *
 *******************************************************/
```

el motivo es que el más mínimo cambio requerirá un tedioso trabajo de volver a recuadrar con asteriscos el resultado.

Otro ejemplo de comentario difícil de mantener es:

```
Cálculo del Valor Actual Neto
/*************************/
```

P12.503 ☠ Los comentarios no deben repetir el código ni formularlo de otra manera, sino que deben *explicar la intención* del código.

C12.504 💣 No se recomiendan los comentarios de fin de línea puesto que tienden a repetir el código y a contener abreviaciones difíciles. Sí son útiles, sin embargo, en la declaración de variables.

O12.505 ☞ En cada módulo se mantendrá una parte de comentarios destinada exclusivamente a documentar aquellas decisiones tomadas "sobre la marcha" relativas al funcionamiento o limitaciones del código, o cualquier cosa que se asuma sobre parámetros, entorno o variables globales.

O12.506 ☞ Cada rutina importante debe tener un bloque de comentarios que describa :

- Su cometido
- Los parámetros que acepta y sus rangos de valores válidos. Precondiciones
- El resultado que devuelve. Postcondiciones
- Excepciones que provoca
- Cualquier efecto secundario, incluyendo cambios en variables globales.
- Es recomendable incluir también un ejemplo de *uso* (no de llamada, puesto que esto último es obvio a partir de la declaración). Por ejemplo:

```
/* getchar - equivale a getc(stdin)
*
*  getchar devuelve el siguiente carácter introducido
*  desde el teclado. Cuando se produce un error o no
*  hay ningún carácter pendiente de leer, devuelve la
*  constante EOF. Un caso típico es:
```

```
*
*  int caracter;
*  if (( car = getchar() ) != EOF)
*       // correcto - caracter tiene el carácter siguiente
*  else
*    // fallo - usar la función ferror(stdin) para ver
*    // el error concreto que se ha producido
*
*/
```

En plataformas como Java o .NET, donde existe un estándar de documentación (Javadoc), gran parte de estos requisitos se cumplen por la propia definición del estándar.

O12.507 ☞ Todas las funciones documentadas que devuelvan referencias a objetos deben indicar explícitamente si el valor **`nil / null`** es un valor de retorno válido, en qué circunstancias y cual es el significado de ese null ("inexistente", "sin dato", "no aplicable", etc…)

O12.508 ☞ Todas las funciones documentadas que traten con porcentajes o permiles (a la entrada o a la salida) deberán indicar explícitamente en su documentación el rango del porcentaje (0.0 a 1.0 o 0 a 100)

O12.509 ☞ Todas las funciones documentadas que traten con magnitudes físicas deberán indicar explícitamente la unidad de medida.

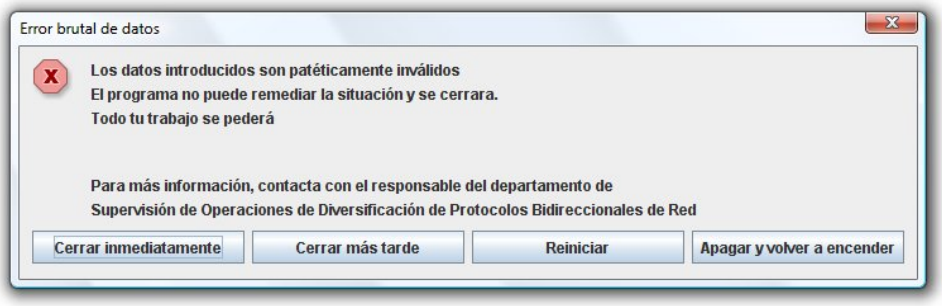
P12.510 ☠ Salvo en el caso concreto en el que un fragmento de código debe desactivarse por una situación temporal, no se dejará en el código fuente fragmentos de código comentados. En caso de desactivación temporal de un bloque de código, debe comentarse junto al mismo por qué se desactiva, cuando se realiza la desactivación, en base a qué requerimiento y cuando se prevé la reactivación.

S12.511 ☺ Es recomendable incluir comentarios antes de cada bucle explicando su propósito:

```
// Buscamos el comando "Fin de párrafo" en la cadena
while ( cadena.charAt(posicion) != CMD_FIN_PARRAFO) {
   ...
}
```

13 INTERACCIÓN CON EL USUARIO

13.1 Notificación de Errores

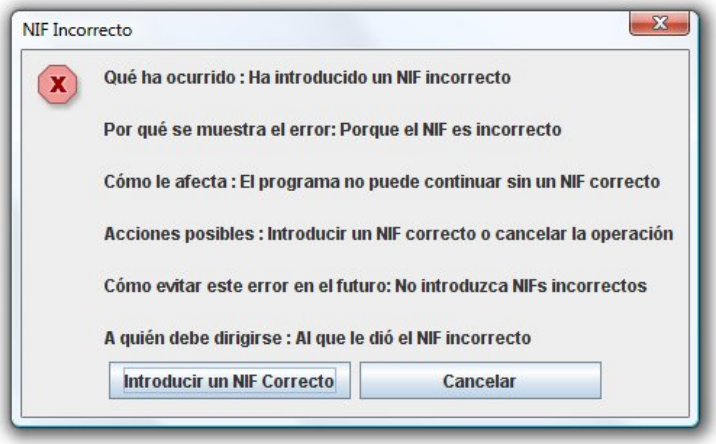


S13.100 ☺ Como dicen (Cooper y Reimann 2003), es un axioma que cualquier mensaje de error es una humillación para el usuario, por lo tanto cualquier comunicación de errores debe ser sutil, educado y no echar nunca la culpa al usuario

O13.101 ☞ Cuando una excepción se propaga hasta la capa de interfaz y por lo tanto no quede más opción que mostrarla, debe ofrecerse siempre la siguiente información que sea aplicable:

- Qué es lo que ha ocurrido (sin entrar en tecnicismos)
- ¿Por qué se está mostrando este mensaje de error?
- Cómo afecta al usuario el error
- Qué acciones son posibles a partir de ahora
- Si el error es evitable en sucesivas ocasiones, qué se puede hacer para evitarlo.
- A quién debe dirigirse el usuario para solucionar el problema (sin tecnicismos ni suponiendo la existencia de departamentos, ni tampoco suponiendo que el usuario actual *no es* aquel al que teóricamente debería dirigirse uno).

La guía anterior no debe seguirse como una plantilla, sino a nivel conceptual. Esto significa, por ejemplo, que si con una frase se cubren varios puntos, no hace falta repetir la misma frase en diferentes "epígrafes" como si el usuario fuese tonto:



O13.102 ☞ Toda la información técnica del error (excepción, mensaje de detalle, traza, volcados varios, etc.) debe poder guardarse en un archivo de texto que pueda ser enviado por EMail o comunicado de cualquier otra manera. El archivo no se generará automáticamente, sino que el usuario deberá poder elegir si se genera y dónde y con qué nombre se guarda.

S13.103 ☺ Siempre que sea posible (.NET / .Java), se ofrecerá al usuario la posibilidad de enviar un informe de error por vía informatizada, preferentemente EMail. Esta vía deberá venir pre-configurada en el programa, sin preguntas técnicas previas al envío.

S13.104 ☺ Siempre que sea posible, se adoptará una política preventiva de control de datos, y no una notificación de errores a posteriori. Esto incluye tanto impedir que se introduzcan datos erróneos como implementando políticas de normalización de datos (incluso a nivel semántico, como sustituir "Fco" por "Francisco")

P13.105 ☠ Nunca se implementará envío de información fuera del equipo del usuario sin que medie consentimiento expreso por parte del mismo. Dicho consentimiento deberá obtenerse indicando claramente:

- Qué información se envía
- A dónde se envía
- El uso que se dará a esta información
- El tiempo de custodia de la información y su eliminación

14 DATOS EXTERNOS Y BASES DE DATOS

Gran parte de las normas de esta sección provienen de (Celko 2005) y (Pascal 2007)

14.1 Bases de Datos

O14.100 ☞ El acceso a bases de datos externos se realizará siempre mediante pools de conexiones y nunca mediante conexiones directas.

P14.101 El acceso a bases de datos externos se realizará siempre mediante pools de conexiones y nunca mediante conexiones directas.

En aplicaciones que permitan usuarios concurrentes y paginación de los resultados, nunca se guardarán la totalidad de los datos recuperados en la sesión o asociados al usuario. En su lugar, se almacenará la consulta utilizada para producir el resultado y se efectuará una re-query contra la base de datos por cada página.

S14.102 ☺ Todos los principios enumerados para el acceso a bases de datos serán de aplicación para cualquier otra fuente de datos, ya sean consultas contra un Active Directory Service, recuperación de datos de un servicio web, lectura de datos de un archivo externo, etc.

O14.103 ☞ Cuando el acceso a un almacén de datos sea transaccional, el bloque de código que inicia la transacción deberá ser en el encargado de cerrarla.

P14.104 No se asumirá que las transacciones no cerradas terminan de forma automática en `rollback`. El rollback deberá ser siempre explícito.

14.2 Definición de Datos (DDL)

P14.200 No se utilizarán tipos de datos propietarios, ni tipos de datos que dependan de la implementación. Para tipos de datos numéricos, se utilizará siempre la sintaxis explícita NUMERIC(p,q) y nunca FLOAT, DOUBLE, INT o tipos de datos similares.

P14.201 Como clave primaria siempre se utilizará un valor:

- No accesible para el usuario. No visible, ni editable.
- Generado por el servidor (preferentemente) o por el programa
- Inmutable

O14.202 Salvo las claves primarias, todas las demás CONSTRAINT introducidas en una base de datos deben tener un nombre legible y fácilmente reconocible.

O14.203 Los nombres de todas las tablas que el programa maneje directamente (y no a través de una capa ORM, por ejemplo), estarán en plural, con la única excepción de aquellas tablas que contengan una única fila (como por ejemplo, una fila de Constantes)

P14.204 Se evitará asignar nombres diferentes al mismo concepto en diferentes tablas.

C14.205 Salvo que no exista alternativa, se evitará convertir al servidor de bases de datos en un mero servidor de archivos mediante el uso intensivo de campos BLOB para almacenar datos sobre los cuales no cabe posibilidad alguna de proceso relacional : imágenes, sonido, vídeo, etc. En estos casos, se almacenará únicamente una referencia al archivo, mientras que los propios datos se almacenarán externamente.

P14.206 No se crearán tablas de "atributos genéricos" salvo que en los requerimientos realmente figure de manera explícita la necesidad de definir atributos dinámicamente.

(Conocido también como el anti-patrón EAV - **`Entity-Attribute-Value`**, consistente en crear tablas de metainformación con columnas tales como entidad, atributo, valor, con el objetivo de conseguir una supuesta flexibilidad futura del modelo de datos).

P14.207 Nunca se utilizarán campos donde NULL pueda tener varios significados diferentes.

O14.207 ☞ Las tablas deberán estar normalizadas, con independencia de si se crean de manera directa O se crean mediante autogeneración a partir de un modelo ORM

Forma Normal	Situación
Primera Forma Normal (1NF) (Grupos de campos repetidos y datos no atómicos)	☞ (Obligatoria)
Segunda Forma Normal (2NF) (Dependencias funcionales directas)	☞ (Obligatoria *salvo* que exista la posibilidad de que una dependencia funcional cambie en el futuro)
Tercera Forma Normal (3NF) (Dependencias funcionales transitivas)	☞ (Obligatoria *salvo* que exista la posibilidad de que una dependencia funcional cambie en el futuro)
Forma Normal de Boyce-Codd (BCNF)	☞ (Obligatoria *salvo* que exista la posibilidad de que una dependencia funcional cambie en el futuro)
Cuarta Forma Normal (4NF) (Múltiples relaciones 1:m m:n por tabla)	☺ (Recomendable)
Quinta Forma Normal (5NF)	☺ (Recomendable)
Forma Normal de Dominio/Clave (DKNF) Derivabilidad de las restricciones de dominio	☺ (Recomendable. Sin embargo, muy difícil de conseguir en la práctica)

14.3 Sentencias y Consultas (DML)

P14.300 Nunca se utilizará composición manual de sentencias:

```
   sql.executeQuery("SELECT * FROM BAR WHERE ID="+id+" AND
NAME='"+name+"'")
```

En su lugar, se utilizarán siempre sentencias paramétricas

P14.301 No se utilizarán parámetros posicionales ("?"). Todos los parámetros de una consulta SQL deberán ser parámetros por nombre. En lenguajes donde no haya más remedio (Java-JDBC), se utilizará o bien una arquitectura de persistencia que proporcione parámetros con nombre (Hibernate, iBatis, etc.) o bien se crearán una serie de rutinas con el mismo efecto.

P14.302 No se accederá a las columnas de una consulta por su posición, sino siempre por su nombre. A todas las columnas que no sean columnas físicas (como por ejemplo : columnas calculadas, literales, etc..) se les asignará un alias y el acceso a las mismas se realizará exclusivamente mediante este alias.

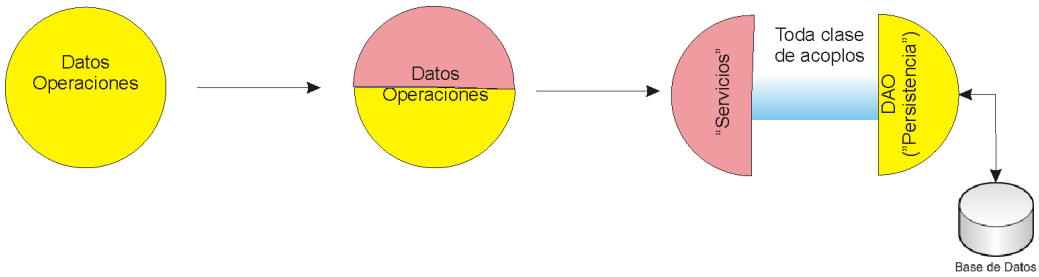
S14.303 Salvo que una consulta esté seleccionando una única fila, cada query deberá asumir que puede recuperar un conjunto potencialmente enorme de datos, por lo que tomará las medidas oportunas para protegerse de esta situación.

P14.304 Nunca se utilizarán directamente en el código localizadores físicos de filas : IDENTITY, ROWID, GUID, etc.

14.4 Bases de Datos y Objetos - ORM

D14.400 🕮 🌐 ORM (**`Object-Relational Mapping`**) es el conjunto de técnicas que permiten establecer una equivalencia entre el modelo relacional y el modelo de objetos.

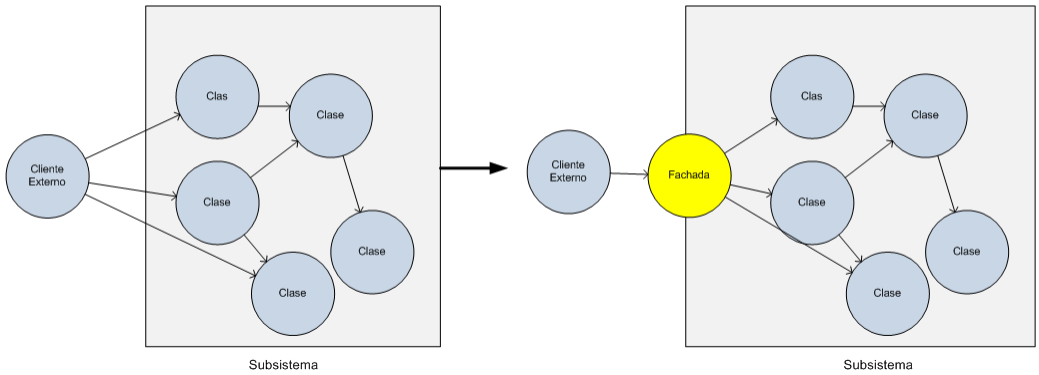
P14.401 ☠ 🌐 El antipatrón más frecuente en un sistema con ORM es el "modelo de dominio anémico" (Fowler, Anemic Domain Model Antipattern 2003), consistente en una involución desde la programación orientada a objetos hacia la programación procedural, camuflada en forma de "capa de servicios" (AKA procedimientos) y "DAO" (AKA datos):



No se utilizará esta forma de planteamiento del sistema ORM. Cada objeto del dominio de negocio se responsabilizará de sus datos *y* de los métodos de negocio que conciernen a la entidad que representa, incluyendo:

- Lógica de creación y destrucción
- Lógica de validación
- Reglas de negocio aplicables a esa entidad
- Reglas de cálculo

P14.403 ☠ 🌐 No confundir el anti-patrón anterior con el patrón Fachada (Gamma, y otros 1995), cuyo propósito es simplificar *frente a un cliente externo a un subsistema* la interacción *con un conjunto de clases del mismo* . Una Fachada *no* es un mero dispatcher



P14.404 ☠ 🌐 En un sistema ORM no se utilizará filtrado ni ordenación programática salvo cuando sea completamente imposible o inviable expresar el conjunto de datos buscado mediante una consulta SQL (o en el lenguaje de consultas del motor de persistencia : EJB-QL, HQL, etc.) .

15 RENDIMIENTO

O15.100 ☞ Como decía D. Knuth,"*Premature optimization is the root of all evil*" (Knuth 1974) . Por lo tanto, nunca se optimizará *antes* de que una rutina/módulo/clase o algoritmo haya sido finalizado, revisado y su funcionamiento verificado por pruebas unitarias.

O15.101 ☞ En la actualidad, la pila de software y hardware que ejecuta un determinado programa es de una complejidad muy superior a la que habitualmente se supone, y los problemas de rendimiento casi nunca se pueden descubrir de forma deductiva. Por ello, este tipo de problemas deben *medirse* y únicamente cuando se ha visto la cusa de la falta de rendimiento, depurarse. Cualquier especulación sobre los motivos de que un software sea así o asá de lento será, casi con total probabilidad, errónea y además una pérdida de tiempo.

O15.102 ☞ La medición de rendimiento en lenguajes con compiladores sobre la marcha (JIT) deberá hacerse teniendo en cuenta que los programas ejecutados bajo esas máquinas virtuales necesitan un tiempo inicial (*"warming time"*) para alcanzar un estado estacionario. El tiempo de precalentamiento no debe computarse puesto que es dependiente de multitud de factores que nada tienen que ver con el algoritmo cuyo rendimiento se deseae verificar.

16 TRAZA

16.1 Formato General

O16.100 ☞ Dado que muchos errores provienen del proceso incorrecto de caracteres espurios (espacios, tabuladores, etc.), la impresión de cualquier variable del programa se realizará entre símbolos delimitadores apropiados (recomendablemente corchetes), permitiendo detectar caracteres adicionales

```
   logger.info("ID del expediente ["+id+"]");
```

S16.101 ☺ Para evitar disrupción del formato de los históricos, se evitarán trazas que contengan retornos de carro o saltos de línea, salvo cuando se esté volcando la traza de una excepción

O16.102 ☞ En aplicaciones que admitan usuarios concurrentes (por ejemplo, aplicaciones web), se utilizará siempre el patrón de traza NDC (*Nested Diagnostic Context)*, (Martin, Riehle y Buschmann 1997) o en su defecto se incluirá en *cada* información de traza el ID de la hebra actual.

O16.103 ☞ Cada traza debe ser autocontenida y mostrar toda la información relevante. Una sentencia de traza ***no*** asumirá que el usuario ha visto otra sentencia de traza anterior:

```
   logger.info("Iniciando proceso de la solicitud "+referencia);

   ... cosas ...

   if (alguncaso)
      logger.info("El proceso de la solicitud ha sido incorrecto");    //
NO!
```


17 CODIFICACIÓN DE PRUEBAS

D17.000 🕮 Análisis de cobertura (*coverage*) es una técnica de prueba (habitualmente gestionada por herramientas automáticas) que consiste en asegurarse que cada instrucción del programa ha sido ejecutada en al menos una prueba

D17.001 🕮 *Boundary Value Analysis (BVA)* o análisis basado en valores en la frontera es una técnica de prueba que examina el comportamiento de los sistemas frente a entradas que están a) dentro de los límites nominales, b) *sobre* los límites nominales ("en la frontera") y c) *fuera* de los límites nominales. Por ejemplo, para una función que valida que una clave tenga 8 letras, BVA proporciona tres casos de pruebas : cadenas con menos de 8 letras, con 8 letras exactas y con más de 8 letras. Formas más estrictas de BVA utilizan la regla "Min+,Max-", que añade dos casos más: justo por encima del nominal mínimo y justo por debajo del nominal máximo.

D17.002 🕮 *Equivalence Partitioning (EP)* o técnica de particiones de equivalencia consiste en definir clases de equivalencia entre los conjuntos de datos de prueba y realizar una única prueba por cada clase de equivalencia

O17.003 ☞ La técnica del BVA es el doble de efectiva que las técnicas basadas en particiones de equivalencia o datos aleatorios (Reid 1997)

17.1 Reglas generales

S17.100 ☺

Todos los desarrollos deberán tener dos o tres clases de prueba generales que engloben todas las demás pruebas:

- La primera clase debería englobar las pruebas críticas que comprueban la funcionalidad básica de la aplicación (llamada en ocasiones *Smoke Test*)
- La segunda clase debería englobar el resto de pruebas no críticas
- La tercera clase debería englobar las pruebas de carga

C17.101

No se recomiendan pruebas unitarias que:

- Requieran una preparación manual de cualquier tipo (crear bases de datos, arrancar servidores, etc.)
- Estén limitadas en cuanto al número de ejecuciones o la forma de las mismas
- Requieran un tiempo o instante concreto de ejecución.
- Requieran ser ejecutadas en un orden o secuencia concreta respecto a otras pruebas.

O17.102

Las pruebas asumirán como punto de inicio un entorno "base". A partir de este punto común, cada prueba preparará su entorno según sus necesidades, y a su finalización (tanto con éxito como sin él) lo dejará en el estado en que lo encontró. La definición de "base" dependerá de cada proyecto, pero lo importante es que será común y documentada para todas las pruebas. Una configuración "base" puede ser el estado inicial ("recién instalado") del entorno, con los datos mínimos que se insertan durante la instalación.

O17.103

Cada vez que se encuentre un error en ejecución en una aplicación considerada como "estable", en *primer lugar* se codificará una prueba unitaria que reproduzca el error, a continuación se resolverá el error y se verificará su solución mediante la ejecución de la prueba unitaria. La prueba unitaria de detección se incorporará a la batería general de pruebas

O17.104

Todas las asunciones se harán explícitas mediante una prueba unitaria que las verifique, y cuyo fallo haga que sea inmediatamente claro qué asunción se ha violado. La asunción se documentará en el punto de código de negocio donde se realiza. En la prueba únicamente se referirá a ese punto.

C17.105

Salvo excepciones justificadas, las pruebas deberán contener en su interior los datos que necesitan, en lugar de confiar en recursos externos.

17.2 Pruebas y Requerimientos

O17.200 ☞ Debe haber *como mínimo* una prueba por cada requerimiento de usuario, que verifique el cumplimiento de dicho requerimiento.

S17.201 ☺ Cada requerimiento del usuario $\bigwedge_i P_i \overset{E}{\rightarrow} Q_j$ deberá tener las siguientes pruebas unitarias:

- Prueba de éxito (precondiciones : P_i, postcondiciones Q_j)
- Pruebas inversas:

$$\forall i : \neg P_i \rightarrow \neg\left(\bigwedge_j Q_j\right)$$

o, alternativamente

$$\forall i : \neg P_i \rightarrow \exists j \mid \neg Q_j$$

S17.202 ☺ Es recomendable vincular cada requerimiento con sus pruebas, de forma que se pueda consultar cada uno a través del otro.

S17.203 ☺ Idealmente, debe utilizarse una herramienta de cobertura del código para que las pruebas unitarias alcancen si no el 100% *del código escrito manualmente* (se excluye el código generado), al menos una cantidad muy alta del código existente.

17.3 Pruebas de Interfaz con sistemas externos

O17.300 ☞ Todas las interacciones con sistemas externos (por ejemplo : envíos/recepciones de EMail, envíos de SFTP, invocaciones de servicios web, etc.) deberán probarse bien utilizando el propio sistema externo o - en su defecto - utilizando un sistema externo simulado que ofrezca una interfaz idéntica.

O17.301 ☞ Todas las interacciones con sistemas externos deberán probar igualmente los siguientes casos (Rotem-Gal-Oz 2003):

- El sistema externo ubicado en una máquina no disponible (pérdida de conectividad)
- El sistema externo no está activo (pérdida de servicio)
- El sistema externo no responde o responde de forma extremadamente lenta
- El sistema externo no sigue el protocolo esperado (violación de contrato)
- El sistema externo falla antes de concluir satisfactoriamente la interacción.
- Pérdida de conectividad durante la interacción

17.4 Datos de prueba

S17.400 ☺

En general, se evitará que los datos a utilizar en las pruebas estén presentes de manera literal en las mismas. Como alternativas, se pueden utilizar los siguientes planteamientos (Dustin 2003), en orden decreciente de preferencia

- Datos en tablas o archivos de configuración fácilmente editables
- Generadores de datos
- Datos aleatorios

O17.401 ☞

Los datos de prueba deben presentar suficiente variabilidad como para asegurar unas pruebas completas. En ausencia de análisis de cobertura, deben realizarse al menos pruebas basadas en el análisis de valores de frontera (*boundary value testing*), que realizan las pruebas con datos dentro de los límites operativos de un método, *sobre* los límites operativos del método y *fuera* de dichos límites

C17.402

La creencia de que un 100% en el análisis de cobertura implica que todos los escenarios de fallo han sido probados es una receta para el desastre. Idealmente, debe aplicarse *tanto* análisis de cobertura como BVA, como otras técnicas de prueba. Por ejemplo en la siguiente rutina:

```
public int getNumArchivos(String ruta) {
   return new File(ruta).list().length;
}
```

una prueba con ruta = "c:\ " daría 100% de análisis de cobertura y no fallaría, sin embargo, la rutina falla mediante **`NullPointerException`** con cualquier ruta que no exista.

17.5 Pruebas de Carga

O17.500 ☞

Todas las aplicaciones dispondrán de una prueba de carga consistente en invocar la operación de negocio más pesada posible durante un número muy alto de veces, de forma consecutiva. Esta prueba deberá además comprobar que el uso de los recursos (memoria, CPU, disco, etc.) alcanza un estado estacionario aceptable. El número de operaciones deberá ser, al menos, 3 veces el número máximo detectado históricamente en el entorno del cliente en una situación similar durante un intervalo de un mes. (Por ejemplo : si a lo largo de su existencia el pico de pedidos por mes de un cliente ha sido 20000, la prueba de carga deberá constar de al menos 60000 pedidos)

O17.501 ☞

Las pruebas de carga pueden omitirse para ejecuciones rutinarias de las baterías de pruebas, pero deben ejecutarse siempre que exista un paso a un entorno de preproducción o producción. Si es posible, este comportamiento deberá automatizarse mediante parámetros de ejecución de la prueba, directivas de compilación, propiedades o cualquier otro mecanismo similar del que disponga el entorno.

O17.502 ☞

Todas las aplicaciones que permitan usuarios concurrentes dispondrán de una prueba de carga consistente en invocar la operación de negocio más pesada posible por un número muy alto de usuarios *de forma concurrente*. El número de usuarios concurrentes deberá ser, al menos, 3 veces el número de usuarios máximo detectado históricamente en el entorno del cliente en una situación similar.

18 REGLAS DIVERSAS

S18.100 ☺

Ante la duda sobre qué alternativa tomar, se aplica la regla de "la conveniencia de la lectura tiene prioridad sobre la conveniencia de la escritura", la cual significa que se debe dar prioridad a la alternativa que facilita la lectura y comprensión del código (que, después de todo, es la actividad más frecuente) frente a la alternativa que facilita la escritura o construcción inicial del código.

S18.101 ☺

Cuando existan cálculos complicados o rutinas que deban ser eficientes, es conveniente diseñar un segundo algoritmo que realice de forma lo más segura posible el cometido del primero. En la fase de depuración, ambos algoritmos se activan de forma paralela y se compara el resultado de cada uno.

- Cualquier cambio, por "pequeño" o "trivial" que sea, deberá ser verificado y cumplir todos los requisitos de este manual.
- Se debe construir el software de forma que no haya casos que sean "raros" o "poco frecuentes". Durante la fase de verificación, cualquier situación o *escenario* debe poder ser reproducido con la misma facilidad.
- Cualquier código se trazará línea a línea cuando el depurador del lenguaje permita esta posibilidad. Esto no significa que se tenga que ejecutar el programa secuencialmente (especialmente los programas de Windows), sino que una verificación mínima es crear situaciones de forma que se pase por todas y cada una de las líneas del programa. "Pasar por" significa verlo con el depurador, no suponer que se está haciendo.
- Se debe escribir código para el "programador medio". En España, el programador medio no sabe inglés, a pesar de su opinión contraria al respecto.

P18.102

No se deberán utilizar "trucos de programación", puesto que los errores se suelen concentrar principalmente en las rutinas que los tienen

S18.103 ☺

Todas las operaciones que causan bloqueo deben estar protegidas por timeout()s. Esto incluye, en particular:

- `Thread.wait()`
- todos los `read()`
- `URL.connect()`

19 COMPILACIÓN Y GENERACIÓN

O19.100 ☞ Cada elemento del proyecto debe tener como mucho una única versión editable por una persona, y a la vez nada que sea editable por una persona debe ser derivable de otros elementos.

O19.101 ☞ La corrección de errores de codificación se realizará ***únicamente*** una vez que se haya constatado a ciencia cierta la causa del error (si bien para llegar hasta la causa es posible que sean necesarias modificaciones del código para aumentar la traza o visualizar resultados intermedios) y nunca utilizando la política de "a ver si haciendo esto se arregla".

O19.102 ☞ Salvo en las primeras etapas, todo proyecto deberá estar en un estado que permita la generación de una copia para producción en cualquier momento.

S19.103 ☺ Con independencia de la configuración de los entornos de desarrollo de cada programador, debe existir un entorno de pruebas que sea lo más cercano posible en cuanto a configuración al entorno real del cliente. Como mínimo el sistema operativo, la estructura física de directorios, los servidores de aplicaciones, bases de datos, servidores de autenticación, etc. deben ser iguales. Debe existir un mecanismo que permita la ejecución de todas las pruebas de verificación en este entorno. Adicionalmente, la versión de producción del software debe extraerse de este entorno.

S19.104 ☺ En general es muy buena idea disponer en un servidor (puede ser el mismo en el que se ubica el repositorio de código) de un proceso automático (dado de alta en cron, por ejemplo) que realice cada día(noche) todas las pruebas, análisis de cobertura, métricas y publique los resultados de forma fácilmente accesible (por ejemplo, en formato HTML).

20 HERRAMIENTAS RECOMENDADAS

Únicamente se consideran en este epígrafe herramientas open source o en su defecto gratuitas.

20.1 Java

- EclEmma - **http://www.eclemma.org/** es una herramienta de análisis de cobertura en forma de plug-in para Eclipse

- Checkclipse - **http://www.mvmsoft.de/content/plugins/checkclipse/checkclipse.htm** es un plugin de Eclipse para enlazar con CheckStyle - una de las más conocidas herramientas de validación estática de código Java.

- FindBugs - **http://findbugs.sourceforge.net/** es otro analizador estático de código, escrito por la Universidad de Maryland. A diferencia de Checkclipse, sin embargo, funciona a nivel de bytecode y por lo tanto es compatible con cualquier lenguaje que se ejecute sobre JVM.

- PMD - **http://pmd.sourceforge.net/** - es otro analizador de código más, con un gran conjunto de reglas predefinidas y fácilmente extensible (las reglas se pueden escribir incluso utilizando expresiones XPath)

- AspectJ - **http://www.eclipse.org/aspectj/** es una extensión de Java que implementa la programación orientada a Aspectos - extremadamente útil para implementar sobre la marcha y de forma desacoplada sistemas de validación, medición de rendimiento, traza, seguridad o gestión de transacciones.

- GUICE - **http://code.google.com/p/google-guice/** - es una framework para inyección dinámica que no requiere configuración XML.

- ESC/Java2 - **http://kind.ucd.ie/products/opensource/ESCJava2/** - es otro analizador estático de Java.

20.2 .NET

- NUnit - **http://www.nunit.org/** es el equivalente a JUnit en el mundo .NET : Una herramienta de automatización de pruebas unitarias.

21 BIBLIOGRAFIA

- Alexander, Ian F., y Richard Stevens. *Writing Better Requirements.* Pearson Education, 2002.
- Basili, Victor R. *A Validation Of Object-Oriented Design Metrics As Quality Indicators.* 1995. http://citeseer.ist.psu.edu/367915.html .
- Benlarbi, Saida, Khaled EI-Emam, Nishith Goel, y Shesh N. Rai. «Thresholds for Object-Oriented Measures.» *National Research Council of Canada.* 2000. http://citeseer.ist.psu.edu/benlarbi00thresholds.html.
- Benoit, Yves Le Traon Baudry, y Jean-Marc Jézéquel. «Design by Contract to improve Software Vigilance.» *IEEE Transactions on Software Engineering - http://www.irisa.fr/triskell/publis/2006/letraon06a.pdf*, 2006: Vol 32, N8, 571 - 586.
- Bloch, Joshua. *Effective Java Programming Language Guide* . Prentice Hall PTR, 2001.
- Briand, Lionel C., John W. Daly, Victor Porter, y Jürgen Wüst. *A Comprehensive Empirical Validation of Product Measures for Object-Oriented Systems.* 1998. http://citeseer.ist.psu.edu/briand98comprehensive.html.
- Brown, Wiliam, y Raphael C Malveau. *Antipatterns : Refactoring Software, Architectures and Projects in Crisis.* John Wiley & Sons, 1998.
- Carr, James. *TDD Anti patterns.* 2007. http://blog.james-carr.org/?p=44.
- Celko, Joe. *SQL Programming Style.* Elsevier - Morgan Kaufmann, 2005.
- Chidamber, S. R., y C. F. Kemerer. «A Metrics Suite for Object Oriented Design.» *IEEE Transactions on Software Engineering*, 1994: Vol. 20, #6.
- Codd, Edgar F. *The Relational Model for Database Management, v2.* Addison-Wesley, 1990.
- Cohen, Frank. «SOAP encoding's impact on Web service performance.» 1 de 3 de 2003. http://www.ibm.com/developerworks/webservices/library/ws-soapenc/.
- Cooper, Allan, y Robert Reimann. *About Face 2.0: The Essentials of Interaction Design.* John Wiley & Sons , 2003.
- Darwen, Hugh. *How to handle missing information without using NULLs.* 9 de 05 de 2003. http://www.dcs.warwick.ac.uk/~hugh/TTM/Missing-info-without-nulls.pdf.
- —. «The Askew Wall - SQL and the Relational Model.» Enero de 2006. http://www.dcs.warwick.ac.uk/~hugh/TTM/TTM-TheAskewWall-printable.pdf.
- Date, Christopher J. *Date on Database.* Addison-Wesley, 2006.
- Dunn, Robert, y Richard Ullman. *TQM for Computer Software.* McGraw Hill, 1994.
- Dustin, Elfriede. *Effective Software Testing.* Pearson Education, 2003.
- Foote, Brian, y Josph Yoder. «The Big Ball of Mud.» *Department of Computer Science, University of Illinois.* 1999. http://www.laputan.org/mud/.
- Fowler, Martin. «Anemic Domain Model Antipattern.» 25 de 11 de 2003. http://martinfowler.com/bliki/AnemicDomainModel.html.
- —. *Refactoring : Improving the Design of Existing Code.* Addison-Wesley, 1998.
- Gall, John. *Systemantics: How Systems Work & Especially How They Fail* . Random House, 1977.
- Gamma, Erich, Richard Helm, Ralph Johnson, y John Vlissides. *Design Patterns - Eleemnt of Reusable Object Oriented Software.* Addison-Wesley, 1995.
- Genera, Marcelo, Mario Piattini, y Coral Calero. «A Survey of Metrics for UML Class Diagrams- http://www.jot.fm/issues/issue_2005_11/article1/.» *Journal of Object Technology*, 2005: vol. 4, no. 9, pp. 59-92,.

- Glasberg, D., El-Emam, K., Memo, W., Madhavji, N. «National Research Council of Canada.» *Validating Object-Oriented Design Metrics on a Commercial Java Application.* 2000. http://iit-iti.nrc-cnrc.gc.ca/iit-publications-iti/docs/NRC-44146.pdf.
- Holt, Adams. «Best Practices for Web services.» 17 de 2 de 2004. http://www.ibm.com/developerworks/library/ws-best9/.
- Hunt, Andrew, y David Thomas. *Pragmatic Programmer, The: From Journeyman to Master.* Addison Wesley , 1999.
- IBM Center for Software Engineering. «Orthogonal Defect Classification (ODC).» 1 de 2 de 2002. http://www.research.ibm.com/softeng/ODC/DETODC.HTM.
- IEEE Software Engineering Standards Commiteee. «IEEE Std. 830-1998 : Recommended Practice for Software Requirements Specifications.» IEEE Computer Society, 25 de 6 de 1998.
- ISO/IEC 11179 - 1. «Information technology — Specification and standardization of data element - http://metadata-stds.org/11179-1/ISO-IEC_11179-1_1999_IS_E.pdf.» 1999.
- Knuth, D. «Computer Programming as an Art.» *Turing Award Lecture.* 1974.
- Knuth, Donald. «Structured programming with goto statements.» *Computing Surveys*, 1974: December .
- Larman, Craig. *Applying UML and Patterns, 3rd ed.* Addison Wesley International, 2004.
- Li, Wei, y Sallie Henry. «Object Oriented Metrics that Predict Maintainability.» *Technical Report TR-93-05, Computer Science, Virginia Polytechnic Institute and State University.* 1993. http://eprints.cs.vt.edu/archive/00000347/01/TR-93-05.pdf.
- Lieberherr, K, I. Holland, y A. Riel. «Object-Oriented Perspective : A Sense of Style - http://www.ccs.neu.edu/research/demeter/papers/law-of-demeter/oopsla88-law-of-demeter.pdf.» *IEEE Computer*, 1988: 79.
- Liskov, Barbara. *Family Values: A Behavioral Notion of Subtyping.* 1994. http://citeseer.ist.psu.edu/liskov94family.html.
- Maguire, Steve. *Código sin errores - Técnicas de Microsoft para desarrollar programas en C sin errores.* McGraw Hill, 1994.
- Martin, Robert C., Dirk Riehle, y Frank Buschmann. *Patterns for Logging Diagnostic Messages, Pattern Languages of Program Design 3.* Addison-Wesley Professional , 1997.
- McCabe, Thomas J, y Charles W. Butler. «Design Complexity Measurement and Testing.» *Communications of the ACM*, 1989: December 1989: 1415-1425.
- McConnell, Steve. *Code Complete - A practical handbook of software construction.* Microsoft Press, 1993.
- McCune, Tim. «Exception Handling Anti-patterns.» 4 de 6 de 2006. http://today.java.net/pub/a/today/2006/04/06/exception-handling-antipatterns.html.
- Meszaros, Gerard. *xUnit Test Patterns.* Addison-Wesley, 2007.
- Meyer, Bertrand. *Object-Oriented Software Construction.* Prentice Hall PTR, 1997.
- —. «The Grand Challenge of Trusted Components - http://se.ethz.ch/~meyer/publications/ieee/trusted-icse.pdf.» *ICSE 25 - International Conference on Software Engineering.* Portland, Oregon: IEEE Computer Press, 2003.
- Microsoft Corp. «.NET Naming conventions.» http://msdn2.microsoft.com/en-us/library/ms229045.aspx.
- Miller, George A. «The Magical Number Seven, Plus or Minus Two: Some Limits on Our Capacity for Processing Information.» *The Psychological Review*, 1956: vol. 63, pp. 81-97.
- Neward, Ted. «ORM - The Vietnam of Computer Science.» 26 de 6 de 2006. http://blogs.tedneward.com/2006/06/26/The+Vietnam+Of+Computer+Science.aspx.

- Nygard, Michael T. *Release It! - Design and Deploy Production-Ready Software.* The Pragmatic Bookshelf, 2007.
- Pascal, Fabian. *Database Debunkings.* 2007. www.dbdebunk.com.
- Philips Medical Systems. «C# Coding Standard.» 19 de 5 de 2005. http://www.tiobe.com/standards/gemrcsharpcs.pdf.
- Pressman, Roger S. *Ingeniería del Software - Un enfoque práctico.* McGraw Hill, 1993.
- Reid, S.C. «An empirical analysis of equivalence partitioning, boundary valueanalysis and random testing.» *Proceedings of the Fourth International Software Metrics Symposium*, 1997: 64-73.
- Roberts, Eric S. «Loop Exits and Structured Programming.» *Department of Computer Science, Stanford University.* 1995. http://www.cis.temple.edu/~ingargio/cis71/software/roberts/documents/loopexit.txt.
- Rosenberg, L, R Stapko, y A Gallo. «Object Oriented Metrics for Reliability.» *IEEE International Symposium on Software Metrics*, 1999.
- Rosenberg, L., y L. Hyatt. «Software Quality Metrics for Object-Oriented System Environments.» *Software Assurance Technology Center, Technical Report SATC-TR-95-1001, NASA Goddard Space Flight Center.* 1995. http://satc.gsfc.nasa.gov/support/oo_tech.PDF.
- Rosenberg, Linda. «Applying and Interpreting Object Oriented Metrics.» *NASA.* 1998. http://satc.gsfc.nasa.gov/support/STC_APR98/apply_oo/apply.pdf.
- Rotem-Gal-Oz, Arnon. «Fallacies of Distributed Computing Explained.» 2003. http://www.rgoarchitects.com/Files/fallacies.pdf.
- Software Engineering Institute, Carnegie-Mellon. «Software Technology Roadmap - Cyclomatic Complexity.» 12 de 07 de 2000. http://www.sei.cmu.edu/str/descriptions/cyclomatic_body.html.
- Spolsky, Joel. «Leaky Abstractions.» *Joel on Softwaer.* 2002. http://www.joelonsoftware.com/articles/LeakyAbstractions.html.
- Stevens, Wayne P., Myers Glenford J., y Larry L. Constantine. «Structured Design.» *IBM Systems Journal*, 1974: 115-139.
- Sun Microsystems. «Code Conventions for the Java Programming Language.» 1999. http://java.sun.com/docs/codeconv/html/CodeConvTOC.doc.html.
- Webre, Hal. *Eiffel - Design By Contract.* 2004. http://www.eiffel.com/developers/presentations.
- WS-I. «Basic Profile 1.0.» 16 de 4 de 2004. http://www.ws-i.org/Profiles/BasicProfile-1.0-2004-04-16.html.