Análisis y Diseño Orientado al Objeto para Reutilización

Versión 2.1.1 Octubre de 1997

TR-GIRO-01-97V2.1.1

Francisco José García Peñalvo
Departamento de Ingeniería Electromecánica y Civil
Área de Lenguajes y Sistemas Informáticos
Escuela Universitaria Politécnica, Universidad de Burgos
Av. General Vigón S/N, 09006 Burgos (España)
☎ +34 47 258989
e-mail: fgarcia@ubu.es

José Manuel Marqués Corral
Departamento de Informática
Edificio de Tecnologías de la Información y las Telecomunicaciones
Campus Miguel Delibes
Universidad de Valladolid
Paseo del Cementerio S/N, 47011
Valladolid (España)
☎ +34 83 423000 Ext. 5638
e-mail: jmmc@infor.uva.es

Jesús Manuel Maudes Raedo
Departamento de Ingeniería Electromecánica y Civil
Área de Lenguajes y Sistemas Informáticos
Escuela Universitaria Politécnica, Universidad de Burgos
Av. General Vigón S/N, 09006 Burgos (España)
☎ +34 47 258989
e-mail: jmaudes@ubu.es

Resumen

La reutilización no supone un nuevo concepto en el desarrollo del software. Se ha estado reutilizando bibliotecas de software durante mucho tiempo, pero viendo al código fuente como el único objeto de la reutilización y generalmente empleando técnicas intuitivas. Sin embargo, actualmente, el concepto de reutilización ha evolucionado hacia la idea de que todo el conocimiento y los productos derivados de la producción de software son susceptibles de ser reutilizados en la construcción de nuevos sistemas, surgiendo de esta forma el concepto de asset o de componente software reutilizable.

Cuando en una organización se decide introducir la reutilización en sus desarrollos, aparecen dos actividades perfectamente diferenciadas: ***el desarrollo de componentes software reutilizables*** *(desarrollo para la reutilización) y* ***el desarrollo con componentes reutilizables existentes*** *(desarrollo con reutilización).*

El objetivo de este documento es presentar los principios fundamentales para el diseño de componentes reutilizables, pero desde el prisma de la orientación al objeto.

Para documentar los diseños orientados al objeto que se recogen en este documento se hace uso de UML 1.0.

Palabras Clave

Reutilización de software, componentes software, análisis orientado al objeto, diseño orientado al objeto, principios de diseño, desarrollo para reutilización, UML.

Agradecimientos

Este documento ha sido generado en el seno del grupo GIRO (*Grupo de Investigación en Reutilización y Orientación al Objeto*) compuesto por miembros del Departamento de Informática de la Universidad de Valladolid y por miembros del Área de Lenguajes y Sistemas Informáticos de la Universidad de Burgos. Desde aquí queremos agradecer la inestimable colaboración y las correcciones sugeridas por el resto de los miembros de este grupo.

Tabla de contenidos

Tabla de Figuras

Tabla de Listados

1. Concepto de reutilización

El término reutilización fue originalmente postulado por M.D. McIlroy en la conferencia de la NATO de 1968 sobre Ingeniería del Software [McIlroy76], y desde entonces la reutilización del software ha sido, y sigue siendo, uno de los principales temas de investigación en el campo de la Ingeniería del Software, citándose a menudo como una de las principales técnicas para incrementar la productividad de los desarrolladores de software. Así Mili et al. [Mili95] llegan a afirmar que "*La investigación en estas décadas en los campos de la Ingeniería del Software y de la Inteligencia Artificial ha dejado algunas alternativas, pero la reutilización es la* "única" *aproximación realista para llegar a los índices de productividad y calidad que la industria del software necesita*". Aunque por otra parte, este mismo autor reconoce que no se han conseguido grandes avances en la adopción sistemática de la reutilización en el proceso de construcción del software.

El propósito de la reutilización es mejorar la eficiencia, la productividad y la calidad del desarrollo software. Así la reutilización puede definirse como "*cualquier procedimiento que produce o ayuda a producir un sistema mediante el nuevo uso de algún elemento procedente de un esfuerzo de desarrollo anterior*" [Freeman87b] o como "*la utilización de elementos software existentes durante la construcción de un nuevo sistema software*" [Krueger92].

Se va a denominar componente reutilizable o asset a "*cualquier componente que es específicamente desarrollado para ser utilizado, y es actualmente utilizado, en más de un contexto*" [Karlsson95]. Un componente reutilizable puede pertenecer a cualquier nivel de abstracción de un desarrollo, es decir, puede ser desde una especificación de requisitos a una biblioteca de funciones.

2. Actividades de la reutilización

Cuando la reutilización está presente en un proceso de desarrollo software, este debe integrar todas las actividades necesarias para producir y reutilizar componentes software. Así se distinguen dos actividades principales dentro de la reutilización: *el desarrollo para la reutilización* y *el desarrollo con reutilización*.

2.1 Desarrollo para la reutilización

El desarrollo para la reutilización consiste en la realización que cumplen un conjunto de restricciones sobre su reutilización y su calidad. A la hora de desarrollar componentes reutilizables es fundamental centrarse en criterios de calidad, en detrimento de los costes de producción de los componentes.

Cuando se va a crear un componente reutilizable debe analizarse la variabilidad de los requisitos que satisface dicho componente, de forma que se construya como un componente genérico que pueda ser especializado en el momento de su reutilización para ajustarse a unos requisitos específicos.

2.2 Desarrollo con reutilización

Las técnicas utilizadas en el proceso de desarrollo con reutilización dependen en gran medida de los componentes que se hayan preparado en el proceso de desarrollo para la reutilización.

El desarrollo con reutilización consiste en la generación de nuevos productos software integrando elementos existentes, de forma directa o pasando por un proceso de adaptación. Aparecen así cuatro problemas fundamentales [Business92a], [Krueger92]:

- ***La selección y recuperación de los componentes***
- ***La comprensión y evaluación de los componentes***
- ***La adaptación de los componentes***
- ***La integración de los componentes***

3. Orientación al objeto y reutilización

De forma creciente, los desarrollos con orientación al objeto están empezando a dominar todos los campos de la construcción de software, desde las más simples aplicaciones de información a grandes sistemas financieros distribuidos, desde los applets en las páginas web a sistemas operativos [Lea96].

Uno de los objetivos de la tecnología orientada al objeto es la reutilización. La clase proporciona los mecanismos de encapsulación, abstracción y ocultación de la información, además de ser un componente elemental en la reutilización. La clase proporciona mecanismos de reutilización en dos niveles: como representación de una abstracción de diseño, que se puede extender o especializar, y como *fabrica* de objetos que comparten la estructura y el comportamiento definido por la clase.

La herencia es la propiedad que permite que nuevas clases puedan compartir comportamiento y representación a partir de las clases existentes. Es el concepto del paradigma objetual sobre el que se asientan la reutilización y la extensibilidad del software. A través de la herencia los diseñadores pueden construir nuevos elementos software sobre una jerarquía de elementos existentes, permitiendo abordar el proceso de diseño y construcción del software, sin tener que partir de cero [Marqués95].

Existen dos tipos principales de componentes reutilizables orientados al objeto: ***las jerarquías de herencia*** y ***los frameworks basados en dominio***.

Las jerarquías de herencia son independientes del dominio de aplicación y consisten en un conjunto de clases asociadas por herencia, donde las clases abstractas se encuentran en lo alto de la jerarquía, y las hojas son las clases concretas.

Una clase abstracta es una clase diseñada para ser utilizada como una plantilla para subclases más específicas, no tienen instancias y no están completamente definidas, faltándole la implementación de alguno de sus métodos, que serán definidos en las subclases. Sin embargo, las jerarquías de herencia presentan problemas. No son lo suficientemente eficientes al tener muchas dependencias internas y presentar un acceso bastante pobre.

Por su parte un framework orientado al objeto es un conjunto de clases diseñadas para trabajar juntas para dar solución a un problema u ofrecer alguna capacidad [Sparks96]. Por lo tanto se trata de un tipo de biblioteca de clases, donde las clases tienen un alto grado de acoplamiento, de forma que las clases no pueden sacarse de su contexto, y el framework completo debe ser reutilizado como un componente. Los mensajes son ampliamente utilizados en los frameworks, de forma que las clases ya no están sólo asociadas por herencia.

Un framework describe como implementar todas las partes de una aplicación en un problema particular del dominio. Es una arquitectura general para el dominio y sus

clases describen los objetos en el dominio y la forma en que interaccionan. Una instanciación de un framework es un ensamblado de objetos que trabajan juntos y solucionan un problema particular. Un framework debe ocultar las partes de diseño que son comunes a todas las instancias, y dejar accesibles las partes que deben ser especializadas.

La reutilización de un framework conlleva dos actividades: *definir las nuevas clases que se necesiten* y *configurar un conjunto de objetos*.

4. El proceso de desarrollo para la reutilización

El proceso de desarrollo para reutilización, que se ilustra en la **Figura 1**, dentro de un dominio implica el análisis y la formalización de éste.

Para el análisis de un dominio se recurre al estudio de las aplicaciones existentes y a entrevistas con expertos del dominio. Para la realización de estas entrevistas deben evitarse las reuniones individuales con un solo experto del dominio porque es un mecanismo lento y tedioso que da como resultado la visión del problema de una sola persona. En su lugar se recomiendan técnicas de JAD (*Joint Application Design*), que permiten realizar el proceso en menos tiempo, y solventar inmediatamente las inconsistencias debidas a diferencias de opinión.

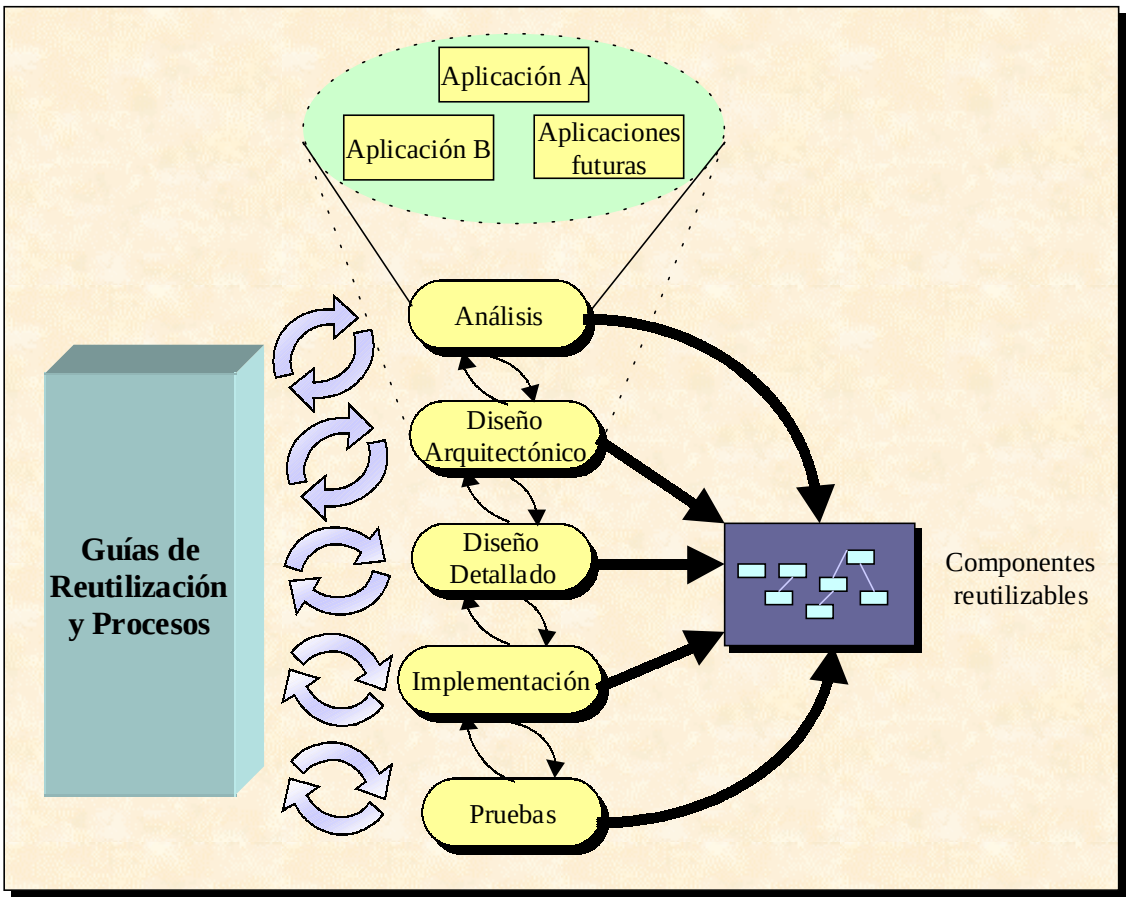

Figura 1. Ciclo de vida general del desarrollo para reutilización.

El desarrollo para la reutilización no implica que se tengan que llevar a cabo una serie de actividades de forma secuencial, sino más una interconexión de actividades que son iteradas. Esta característica se cumple especialmente en el desarrollo de componentes específicos de un dominio.

La vista externa, *especificación de requisitos*, y la vista interna, *reingeniería*, de diferentes aplicaciones se analiza en diferentes momentos del proceso de desarrollo. Las actividades de desarrollo iteran entre los modelos de análisis, los diseños y las implementaciones. Las guías de reutilización fuerzan modificaciones en las versiones actuales de los componentes que repercutirán posteriormente en el análisis de otras aplicaciones. El desarrollo para la reutilización no es una actividad que forme parte del ciclo de vida de desarrollo clásico. El desarrollo para la reutilización debe ir en paralelo con los ciclos de vida de los productos de la organización.

5. Análisis del dominio en el desarrollo para la reutilización

El análisis del dominio es el proceso de identificar y organizar el conocimiento sobre algunas clases de problemas – *el dominio del problema* – para soportar la descripción y la solución a esos problemas [Prieto-Díaz91].

El proceso de desarrollo de componentes reutilizables de un dominio específico comprende las siguientes actividades:

- ***Definición del dominio***
- ***Recolección de aplicaciones del dominio***
- ***Análisis de las aplicaciones del dominio y desarrollo de modelos y de componentes***
- ***Preparación del soporte de la reutilización***

5.1 Definición del dominio

El objetivo es definir el dominio lo más estrechamente posible, pero manteniendo visibles las relaciones con otros dominios adyacentes.

Los componentes específicos del dominio deben permanecer separados de los componentes específicos de un producto y de los componentes de propósito general.

Deben realizarse modelos de dominio que expliquen y describan el dominio y las relaciones entre los diferentes componentes.

5.2 Recolección de aplicaciones del dominio

Se puede encontrar información sobre el dominio en antiguos productos. Las especificaciones, diseños, código fuente, casos de prueba... de antiguas aplicaciones, que posiblemente no serán orientadas al objeto, pueden convertirse en potenciales fuentes de componentes reutilizables mediante un proceso de reingeniería. Esta reingeniería se orienta a separar los componentes propios del dominio de los componentes generales y del producto.

5.3 Análisis de las aplicaciones del dominio y desarrollo de modelos y de componentes

Durante el estudio de la documentación de los productos antiguos, se pueden desarrollar diferentes modelos que pueden servir también como componentes reutilizables. Los modelos de análisis pueden servir como modelos de dominio a la vez que como componentes reutilizables.

5.4 Preparación del soporte de la reutilización

Los ingenieros del software que van a desarrollar con reutilización deben contar con alguna guía que les permita decidir cuando reutilizar y cuando no reutilizar un determinado componente.

Los componentes especialmente complejos, como los frameworks orientados al objeto, deben acompañarse de un manual de reutilización, que describa como adaptarlo y configurarlo para diferentes requisitos.

6. Análisis orientado al objeto y diseño orientado al objeto

La transición entre las fases de análisis y diseño en la orientación al objeto es mucho más suave que en las metodologías estructuradas, no habiendo tanta diferencia entre las etapas [Piattini96]. Es difícil determinar donde acaba el AOO y donde comienza el DOO, siendo la frontera entre el AOO y DOO totalmente inconsistente, de forma que lo que algunos autores incluyen en el AOO otros lo hacen en el DOO.

El objetivo del AOO es modelar la semántica del problema en términos de objetos distintos pero relacionados. Por su parte, el DOO conlleva reexaminar las clases del dominio del problema, refinándolas, extendiéndolas y reorganizándolas, para mejorar su reutilización y tomar ventaja de la herencia. El análisis casa con el dominio del problema y el diseño con el dominio de la solución; por lo tanto el AOO enfoca el problema en los objetos del dominio del problema y el DOO en los objetos del dominio de la solución.

Según Monarchi et al [Monarchi92] los objetos del dominio del problema representan cosas o conceptos utilizados para describir el problema, denominándose objetos semánticos porque ellos tienen el significado del dominio del problema. El análisis se centra en la representación del problema, la identificación de las abstracciones que tienen el significado de las especificaciones y de los requisitos del sistema. El énfasis del diseño está en definir la solución. Las clases semánticas pueden ser extendidas durante el análisis o el diseño. Los objetos del dominio de la solución incluyen: *objetos de interfaz, objetos de aplicación* y *objetos base o de utilidad***.** Estos no forman parte directamente de los objetos del dominio problema, pero representan la vista del usuario de los objetos semánticos.

Se puede definir AOO como *el proceso que modela el dominio del problema identificando y especificando un conjunto de objetos semánticos que interaccionan y se comportan de acuerdo a los requisitos del sistema* [Monarchi92].

Se puede definir DOO como *el proceso que modela el dominio de la solución, lo que incluye a las clases semánticas con posibles añadidos, y las clases de interfaz, aplicación y utilidad identificadas durante el diseño* [Monarchi92].

El AOO y el DOO no deben separarse en fases muy separadas, siendo recomendable llevarlas a cabo concurrentemente, así el modelo de análisis no puede completarse en ausencia de un modelo de diseño, ni viceversa. Uno de los aspectos más importantes del ADOO es la sinergia entre los dos conceptos [Martin93].

6.1 Actividades generales en el AOO para la reutilización

El AOO constituye el acto por el que se determinan las abstracciones que subyacen en los requisitos. Es el proceso de razonamiento que comienza con los requisitos y finaliza

con un conjunto de objetos que expresan dichas abstracciones, así como con los mensajes que soportan los comportamientos requeridos.

Para Monarchi et al [Monarchi92] el proceso del análisis del dominio del problema conlleva tres actividades:

- *La identificación de las clases semánticas, los atributos, el comportamiento y las relaciones (generalizaciones, agregaciones y asociaciones).*
- *El emplazamiento de las clases, atributos y comportamiento.*
- *La especificación del comportamiento dinámico mediante paso de mensajes.*

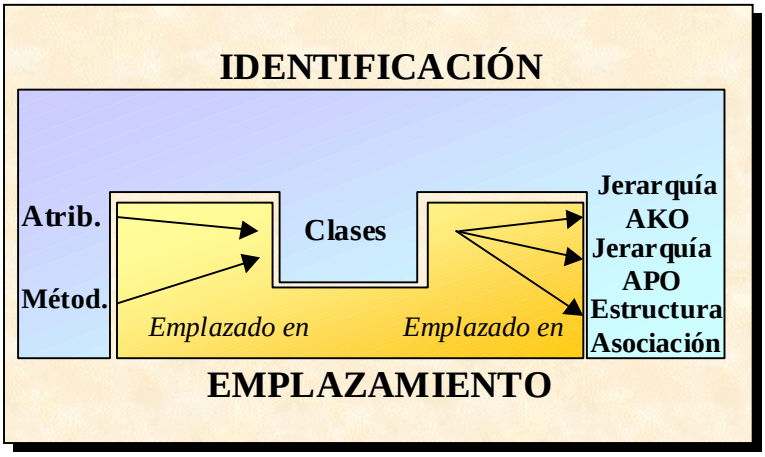

Figura 2. Identificación y emplazamiento

Las recomendaciones que se citan a continuación pueden aplicarse análisis orientado al objeto para la reutilización, viniendo derivadas algunas de ellas del análisis tradicional. Algunas de las consideraciones que se van a realizar no son específicas de un proceso de reutilización, pero influyen en su mejora.

1: ***Recoger información de la mayor cantidad de fuentes posible.***

Un sistema basado en una visión única puede no cumplir los requisitos actuales o futuros. Para conseguir un componente estable, se debe conseguir información de varias fuentes diferentes.

2: ***Captura de todos los requisitos como modelos de análisis.***

La completa comprensión de los requisitos es esencial en un desarrollo software. A la hora de desarrollar para la reutilización toma una especial importancia la captura de los requisitos no funcionales, debido a que con frecuencia contribuyen a aumentar la reutilización de un componente.

3: ***Utilizar una notación que sea fácil de comprender***.

Para facilitar la reutilización de un componente en futuros desarrollos es esencial que los modelos sean fáciles de comprender por el que los va a reutilizar. La notación utilizada para realizar los modelos se convierte así en un factor clave, especialmente si se tiene en cuenta el amplio número de metodologías de ADOO existentes, cada una con sus propias aportaciones en lo tocante a la notación. La comunidad de orientación al objeto está haciendo esfuerzos por convergir las diferentes tendencias en ADOO hacia un punto común. En este sentido cabe

destacar el ***Lenguaje de Modelado Unificado*** (*UML – Unified Modeling Language*) de Rational Software Corporation [Rational97a]. UML se basa en los trabajos de tres de los más conocidos expertos en metodologías orientadas al objeto (*Graddy Booch, Jim Rumbaugh y Ivar Jacobson*), y es una combinación de notación [Rational97d], una semántica [Rational97b], [Rational97c] y un metamodelo para un lenguaje de modelado. Aunque su total aceptación como estándar por parte de OMG (*Object Management Group*) no se ha producido (*y no está claro que se produzca tal cual*), si se puede asegurar que se ha convertido en el estándar de facto de las notaciones en la orientación al objeto [Ohnjec97].

4: ***Refinar y formalizar los modelos de análisis***.

Se debe refinar el modelo conceptual para eliminar clases redundantes y para buscar niveles de mayor abstracción.

5: ***Uso de la herencia como mecanismo de especialización de tipos.***

La estructura de herencia[1] constituye un importante mecanismo para el soporte de la reutilización al permitir la extensión y el refinamiento. Las estructuras de herencia que primero se localizan son propias del dominio del problema.

El concepto de herencia, cuando se corresponde con la relación de especialización/generalización de clases, permite organizar la información en una estructura jerárquica de componentes, en la que los elementos de alto nivel capturan el comportamiento común de todos sus elementos derivados. De esta forma se establece un mecanismo, mediante el cual se permite compartir (*especialización*) y/o factorizar (*generalización*) la información, posibilitando no sólo la reutilización, sino también la extensibilidad del software, al permitir definir nuevos componentes que pueden heredar el comportamiento y la representación de los componentes existentes [Marqués95].

El concepto de herencia puede corresponderse con otros tipos de relaciones diferentes a la especialización/generalización, destacando los procesos de agregación y composición.

Sin embargo, una de las características diferenciales del desarrollo de aplicaciones utilizando la orientación al objeto es su naturaleza incremental. Este aspecto de desarrollo incremental en el que se van incorporando clases a medida que el diseño orientado al objeto progresa, puede dar lugar a jerarquías de clases *sin sentido*, siendo fundamental la obtención de una jerarquía de clases correcta para la comprensión y el seguimiento de los modelos y su reutilización [Marqués96].

6: ***Analizar aplicaciones existentes dentro del dominio para identificar características comunes***.

Si se han desarrollado varias aplicaciones pueden existir varias soluciones al problema. Se debe tratar de obtener las abstracciones adecuadas y colocarlas en un modelo general.

[1] Rumbaugh et al [Rumbaugh96] sostienen que los términos herencia, generalización y especialización se refieren a aspectos de la misma idea y suele ser posible utilizarlos de forma intercambiable. Herencia hace alusión al mecanismo empleado para compartir atributos y operaciones empleando la relación de herencia. La generalización y la especialización son dos puntos de vista distintos de la misma relación vista desde la superclase o desde las subclases. La palabra generalización proviene del hecho consistente en que la superclase generaliza a las subclases. La especialización hace alusión al hecho consistente en que las subclases refinan o especializan a la superclase.

El mayor problema para llevar a cabo esta actividad puede ser la accesibilidad a la información necesaria para analizar las aplicaciones.

7: ***Generalizar de forma que se soporten cambios futuros en los componentes.***

La introducción de abstracciones de alto nivel facilitan el cambio de los componentes sin tener que reestructurar la arquitectura, sino simplemente extendiendo la jerarquía mediante herencia.

8: ***Indicar las variantes de los sistemas existentes y de los sistemas futuros dentro del dominio.***

Mediante el uso de la herencia para señalar las diferencias entre los sistemas, obliga a los analistas a establecer abstracciones estables que permanezcan invariantes durante un largo período de tiempo.

9: ***Realización de una revisión formal de los documentos generados en la fase de análisis.***

El examen de los documentos generados durante la fase de análisis puede dar como resultado la detección de errores y el descubrimiento de nuevos requisitos que pueden ser útiles para obtener un componentes completamente reutilizable.

Las nueve recomendaciones anteriores son lo suficientemente genéricas como para que se tengan en cuenta en cualquier caso, pero no constituyen una guía o metodología para la realización de un análisis orientado al objeto. Por este motivo se van a citar y comentar brevemente los ocho pasos que habría dar para construir un modelo de objetos según una de las mejores metodologías orientadas al objeto, OMT [Rumbaugh96].

a. Identificación de los objetos y de las clases

El primer paso es la identificación de las clases de objetos relevantes en el dominio del problema[2], evitando estructuras de implementación.

Debe comenzarse por la enumeración de los candidatos a clases de objetos que se encuentran en la descripción escrita del problema. En esta primera fase no hay que ser especialmente selectivo. Con frecuencia, las clases se corresponden con los sustantivos de la especificación escrita del problema.

El siguiente paso es descartar las clases innecesarias e incorrectas según los siguientes criterios:

- *Clases redundantes:* Clases que expresan la misma información. Se debe retener la que tenga el nombre más descriptivo.
- *Clases irrelevantes:* Clases que tienen poco o nada que ver con el problema. Deben ser eliminadas.
- *Clases vagas:* Clases con unos límites mal definidos, o con un ámbito excesivo. Una clase debe representar un concepto específico.
- *Atributos:* Los nombres que describen objetos individuales deben recalificarse como atributos. Por ejemplo, nombre, sexo... Sin embargo, si la existencia independiente de una propiedad es importante, se debe crear una clase con ella.

[2] Entidades físicas y conceptos.

- *Operaciones:* Si un nombre describe una operación que se aplica a objetos y que no es propiamente manipulada en sí, entonces no es una clase. Sin embargo, si una operación posee características propias debe modelarse como una clase.
- *Roles:* El nombre de una clase debería reflejar su naturaleza intrínseca, y no el rol que desempeña en una asociación.
- *Estructuras de implementación:* Las estructuras ajenas al mundo real deben ser eliminadas del análisis. Las estructuras de datos tales como listas enlazadas, árboles, matrices y tablas, son casi siempre de implementación.

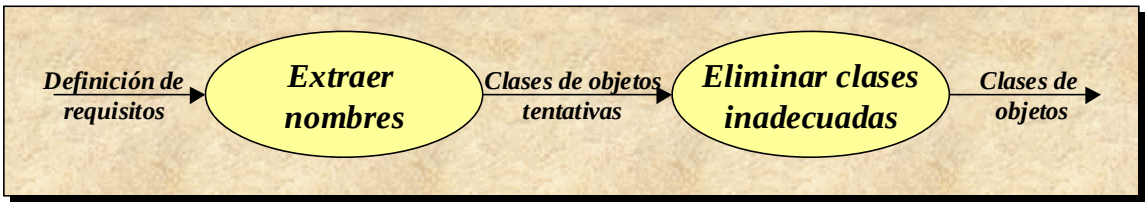

Figura 3. Identificación de clases de objetos

b. Preparación de un diccionario de datos

Debe prepararse un diccionario de datos para todas las entidades del modelo, escribiendo un párrafo que describa con precisión y absoluta claridad cada clase de objetos.

El diccionario de datos también describe las asociaciones, atributos y operaciones.

c. Identificación de asociaciones y agregaciones

Se identifican las asociaciones entre clases. Toda dependencia entre dos o más clases o una referencia de una clase a otra es una asociación.

Las asociaciones suelen corresponderse con verbos de estado o con locuciones verbales[3].

Como en el caso de las clases, deben extraerse todos los candidatos. En este momento no debe perderse mucho tiempo intentando distinguir una asociación de una agregación, se debe utilizar lo que parezca más natural en este momento.

El siguiente paso es descartar las asociaciones innecesarias e incorrectas, siguiendo los siguientes criterios:

- *Asociaciones entre clases eliminadas:* Si se ha eliminado alguna de las clases que interviene en la asociación debe eliminarse ésta o redefinirla en términos de otras clases.
- *Asociaciones irrelevantes o de implementación:* Hay que eliminar cualquier asociación que esté fuera del dominio del problema, o que afecte a estructuras de implementación.
- *Acciones:* Las asociaciones deben describir propiedades estructurales del dominio del problema, y no sucesos transitorios.
- *Asociaciones ternarias:* La mayoría de las asociaciones entre tres o más clases se pueden descomponer en asociaciones binarias.

[3] Ubicaciones físicas, acciones dirigidas, comunicaciones, propiedad, cumplimiento de alguna condición.

- *Asociaciones derivadas:* Deben omitirse las asociaciones que puedan ser definidas en términos de otras porque son redundantes. Por ejemplo *Abuelo* puede definirse en términos de una parejas de asociaciones *Padre de*.
- *Asociaciones de nombre incorrecto:* No debe indicarse cómo o por qué se produce una situación; hay que decir lo que es.
- *Nombres de rol:* Tienen que añadirse donde convenga. Los nombres de rol describen el papel desempeñado por la asociación desde el punto de vista de la otra clase. Por ejemplo, en la asociación *Trabaja para*, la *Compañía* tiene el rol de *empresario* y la *Persona* tiene el rol de *empleado*. Si sólo existe una asociación entre una pareja de clases, y el nombre de la clase describe correctamente su papel, se pueden omitir los nombres del rol.
- *Asociaciones cualificadas:* Normalmente, el nombre identifica al objeto en un cierto contexto; la mayoría de los nombres son únicos desde el punto de vista global. El contexto se combina con el nombre para identificar de forma única al objeto. El cualificador distingue a los objetos del lado "*muchos*" de la asociación.
- *Multiplicidad:* Hay que especificarla, aunque se debe tener presente que la multiplicidad suele variar a lo largo del análisis. Se debe desconfiar de los valores de multiplicidad iguales a "*uno*".
- *Asociaciones inexistentes:* Las asociaciones inexistentes que se vayan descubriendo deben ir añadiéndose al modelo.

d. Identificación de los atributos de los objetos y de los enlaces

El siguiente paso es la identificación de los atributos de los objetos. Los atributos son propiedades de los objetos individuales. Los atributos no deben ser objetos, debe utilizarse una asociación para mostrar la relación entre dos objetos.

Los atributos suelen corresponderse con nombres seguidos por frases posesivas, por ejemplo "*el color de la silla*". Los adjetivos suelen representar valores de los atributos específicos.

A diferencia de las clases y las relaciones, es menos probable que los atributos se describan por completo en la definición del problema. Durante el análisis deben evitarse los atributos que sean sólo de implementación y asegurarse que cada atributo recibe un nombre significativo.

Los atributos derivados debe omitirse o marcarse de forma clara. Por ejemplo, *edad* puede derivarse de *fecha de nacimiento* y de *fecha actual*.

Los atributos de enlace también deben ser identificados. Un atributo de enlace es una propiedad del enlace de dos objetos, en lugar de ser una propiedad de un objeto individual. Por ejemplo, la asociación muchos a muchos entre *Accionista* y *Compañía* tiene como atributo de enlace el *número de acciones*.

A continuación deben filtrarse los atributos con el fin de eliminar aquellos que sean innecesarios o incorrectos siguiendo los siguientes criterios:

- *Objetos:* Si es importante la existencia independiente de una entidad, y no sólo su valor, se trata de un objeto. La distinción depende con frecuencia de la aplicación. Por ejemplo, en una lista de correos *Ciudad* sería un atributo, mientras que en un censo *Ciudad* sería un objeto. Una entidad que tiene características propias dentro del problema es un objeto.

- *Cualificadores:* Si el valor de un atributo depende de un contexto particular, hay que pensar recalificar el atributo como cualificador. Por ejemplo, *número de empleado* no es una propiedad única para una persona que tenga dos trabajos, lo que hace es cualificar la asociación *Compañía que emplea a persona*.
- *Nombres:* Un nombre es un atributo de un objeto cuando no depende del contexto y sobre todo cuando no necesita ser único. Si un nombre responde afirmativamente a las preguntas *¿Selecciona el nombre uno de los objetos de un conjunto? ¿Puede un objeto del conjunto tener más de un nombre?,* se trata de un cualificador de una asociación.
- *Identificadores:* Los lenguajes orientados al objeto tienen la noción de identificador de objetos para hacer alusión no ambigua a un objeto. Estos identificadores no deben enumerarse en los modelos de objetos al ser una propiedad intrínseca del objeto. Sólo deben enumerarse aquellos atributos que existan en el dominio del problema.
- *Atributos de enlace:* Si una propiedad depende de la existencia de un enlace, es un atributo del enlace y no de un objeto relacionado. Los atributos de enlace son evidentes en las asociaciones muchos a muchos, ya que no pueden asociarse a ninguna de las clases como consecuencia de su multiplicidad. Los atributos de enlace son más sutiles en asociaciones uno a muchos, porque se pueden asociar al objeto de la parte "*muchos*" sin pérdida de información.
- *Valores internos:* Si un atributo describe el estado interno de un objeto que es invisible fuera del objeto, hay que eliminarlo del análisis.
- *Detalles finos:* Los atributos menores, que no afecten a la mayoría de las operaciones, deben omitirse.
- *Atributos discordantes:* Son aquellos que parecen ser completamente distintos e independientes de todos los demás; pueden indicar una clase que debiera fragmentarse en dos distintas.

e. Refinamiento mediante herencia

La siguiente actividad es organizar las clases empleando la herencia para compartir una estructura común. La herencia se puede añadir en dos direcciones: *generalizando aspectos comunes de las clases existentes en una superclase*[4] o bien *refinando las clases existentes para obtener subclases especializadas*[5].

Cuando se lleva a cabo una factorización o refinamiento ascendente se buscan clases con atributos, asociaciones u operaciones similares, de forma que en cada generalización se define una superclase para compartir características comunes.

Las especializaciones por refinamiento progresivo suelen verse directamente a partir del dominio del problema,

La herencia múltiple se puede utilizar para compartir características de varios objetos, pero teniendo en cuenta que se está incrementando la complejidad conceptual y de implementación.

[4] Refinamiento ascendente.
[5] Refinamiento descendente.

f. Verificación de la existencia de las vías de acceso adecuadas para las probables consultas

Se deben seguir las vías de acceso por el diagrama de modelo de objetos para comprobar si se obtienen resultados sensatos. Cuando se espera un valor único, ¿hay una vía que lo proporcione? Para la multiplicidad "*muchos*", ¿existe una forma de seleccionar valores únicos cuando sea necesario?

g. Iteración y refinamiento del modelo

Un modelo de objetos rara vez es correcto en la primera pasada. Todo el proceso de desarrollo de software es una continua iteración; las distintas partes del modelo se encuentran en diferentes fases de acabado. Si se encuentra un error hay que volver a la etapa anterior, si es necesario, para corregirlo.

h. Agrupamiento de las clases en módulos

El último paso del modelado es agrupar las clases en folios y módulos. Las clases que estén fuertemente acopladas deben ir agrupadas, pero teniendo en cuenta que un folio tiene una cantidad prefijada de información.

Un módulo es un conjunto de clases (*uno o más folios*) que captura algún subconjunto lógico del módulo completo.

6.2 Actividades generales en el DOO para la reutilización

Aunque los lenguajes OO, los métodos y las herramientas continúan su proceso de maduración, el diseño de las aplicaciones orientadas al objeto retiene algo de ese oscurantismo propio de los desarrollos software [Lea96]. En este trabajo se pretende exponer una serie de recomendaciones generales y básicas del DOO, teniendo como objetivo de fondo la reutilización del software que se genere y de los diseño en sí.

El propósito del diseño es crear una arquitectura para el sistema que va a desarrollarse, y establecer las tácticas comunes que deben utilizarse por parte de elementos dispares del sistema [Booch96].

Durante el diseño orientado al objeto, se ejecuta la estrategia seleccionada durante el análisis y se rellenan los detalles. Se produce un desplazamiento del énfasis, pasando a los conceptos del dominio de la solución. Los objetos identificados durante el análisis sirven como esqueleto del diseño. Las operaciones identificadas durante el análisis deben expresarse en forma de algoritmos, descomponiendo las operaciones complejas en operaciones internas más sencillas. Las clases, atributos y asociaciones del análisis deben implementarse en forma de estructuras de datos específicas. Será preciso introducir nuevas clases de objetos[6] [Rumbaugh96].

En el nivel arquitectónico es importante mostrar el agrupamiento de clases en categorías de clases, *arquitectura lógica*, y el agrupamiento de módulos en subsistemas, *arquitectura física*.

Un problema en la fase de diseño es el rápido incremento de la complejidad, debido al incremento del número de clases, métodos y atributos. Los subsistemas se introducen para manejar el sistema de forma más abstracta.

[6] Objetos de interfaz, objetos de aplicación y objetos base o de utilidad [Monarchi92].

La fuerte cohesión funcional es una propiedad importante que deben cumplir los buenos subsistemas. Un subsistema con una fuerte cohesión funcional es un subsistema que realiza una función u ofrece un comportamiento perfectamente delimitado.

Ivar Jacobson [Jacobson92] establece una serie de criterios para seleccionar los subsistemas:

- Cuando en un subsistema debe hacerse algunos cambios, estos cambios no deben afectar a más de un subsistema de bajo nivel.
- Cada subsistema debe tener una alta cohesión.
- Los subsistemas deben tener un bajo acoplamiento, limitándose este a comunicaciones entre los subsistemas.
- Debe tratarse de ocular el comportamiento o funcionalidad dentro de los subsistemas.

De cara a la reutilización, los diseños de los subsistemas deben cumplir una alta cohesión, de forma que las clases que forman el subsistema se vean afectadas al mismo tiempo por un cambio en los requisitos, y un bajo acoplamiento, minimizando las colaboraciones entre subsistemas de forma que los cambios en un subsistema afecten lo menos posible a otros.

Otra recomendación de cara a la reutilización viene de la mano de crear el mayor número de clases abstractas posible, de esta forma las clases abstractas encapsulan el comportamiento que puede ser reutilizado mediante la herencia. Esta actividad ayuda a encontrar la abstracción más adecuada, ya que al definir tantas clases abstractas como haya sido posible significa que se ha factorizado tanto como haya sido posible el comportamiento común.

La última fase del diseño recibe el nombre de diseño detallado, y es la fase en que las clases y sus métodos son definidos, siendo muy importante cuidar los aspectos de reutilización debido a que las clases y los métodos constituyen la interfaz de los componentes que se utilizarán en los desarrollos con reutilización.

En general se pueden mencionar los siguientes puntos comunes en la fase de diseño detallado:

- La utilización de la herencia conlleva multitud de ventajas: los componentes serán menores, más sencillos de entender y de modificar, y por consiguiente de reutilizar.
- El protocolo lo constituyen las interfaces de las clases, los métodos públicos de las clases. Utilizando un protocolo estándar los objetos mantienen una interfaz similar y una forma común de nombrar los métodos.
- El polimorfismo reduce el número de interfaces diferentes que se deben tener en cuenta.
- Deben evitarse las clases demasiado grandes, transformándose en nuevas abstracciones o en pequeñas jerarquías de herencia.

En forma de *recetario* se pueden dar las siguientes recomendaciones para aplicarlas en el diseño detallado:

1: ***Si se identifican propiedades generales comunes a algunas clases se puede crear una superclase que las reúna.***

Esta es una guía de diseño ascendente. El establecimiento de la jerarquía de clases es preferible hacerla en fases anteriores.

Desde el punto de vista de la reutilización las superclases debe capturar las características comunes en un grado de abstracción mayor, mientras que las subclases deben recoger las diferencias. Los desarrolladores que reutilicen estos diseños podrán encontrar la abstracción más adecuada para extender el software mediante herencia. Si las características comunes no se capturan de forma adecuada en las superclases, el desarrollador con reutilización se verá obligado a realizar grandes modificaciones y redefiniciones.

2: ***Minimizar el uso de la herencia múltiple***

La herencia múltiple puede ser causa de problemas en el momento de reutilizar un componente. Las estructuras de herencia son mucho más sencillas si no es utiliza la herencia múltiple.

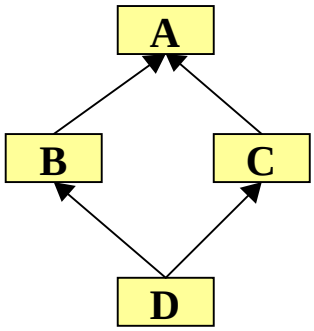

Figura 4. Ambigüedad en la herencia múltiple

Además muchos lenguajes de programación orientados al objeto tienen problemas con la herencia múltiple (*o simplemente no la soportan*) especialmente cuando la estructura de herencia implica ambigüedad, (*ver* **Figura 4**). Para profundizar más en temas de herencia múltiple, en sus problemas y en los algoritmos para solventarlos se recomienda consultar la tesis doctoral del Dr. José Manuel Marqués [Marqués95].

3: ***Si una operación X de una clase se implementa para realizar una operación similar en otra clase, entonces esta operación debe denominarse también X (Introducción de la recursión).***

La introducción de la recursión es importante para conseguir unos protocolos estandarizados que permitan utilizar el polimorfismo de forma más extensa.

Además de esta forma, cuando se vaya a desarrollar con reutilización sólo se tendrán que manejar un conjunto limitado de nombres de métodos.

4: ***Cada método debe realizar sólo una tarea***

De esta forma el protocolo de la clase será más fácil de entender y por lo tanto más fácil de reutilizar.

5: ***Las subclases deben ser especializaciones***

La herencia que no comparte características comunes debe evitarse. Las subclases deben ser autenticas especializaciones de las superclases.

6: ***La cima de una jerarquía de herencia debe ser una clase abstracta***

Es mejor heredar de una clase abstracta que de una clase concreta. Una clase concreta debe ofrecer una definición para la representación de sus datos, y algunas subclases necesitan diferentes representaciones. Dado que una clase abstracta no tiene que ofrecer una representación de sus datos, las futuras subclases podrán

utilizar cualquier representación sin entrar en conflicto con lo que hayan heredado.

7: *Cuidar que el protocolo total de la clase sea pequeño*

El protocolo total de una clase consiste en todos los métodos definidos en la clase y en todos los métodos heredados de sus superclases que no han sido redefinidos o anulados. Si se tienen muchos métodos en las clases de los últimos niveles de la jerarquía se tiene una abstracción muy compleja, y esta jerarquía puede ser dividida en varias jerarquías más pequeñas y menos complejas.

Desde el punto de vista de la reutilización es más difícil reutilizar clases grandes, siendo más fáciles de entender y modificar clases más pequeñas.

8: ***Usar clases abstractas en tantos niveles como sea posible en la jerarquía de herencia***

Una clase abstracta es una clase que no representa completamente un objeto. En su lugar representa un amplio rango de diferentes clases de objetos. Sin embargo, esta representación comprende sólo las características de aquellas clases que tienen objetos en común. Por tanto, una clase abstracta sólo ofrece una descripción parcial de sus objetos [Martin92a].

Una clase abstracta está diseñada para ser reutilizada y actúa como una plantilla para clases concretas. Debe utilizar protocolos estándares y reunir generalidades de una jerarquía de herencia.

Una forma de incrementar la reutilización es utilizar clases abstractas en varios niveles de la jerarquía de herencia.

9: *Las jerarquías de herencia deben ser prudentemente profundas y estrechas*

Si una superclase tiene más de diez subclases directamente dependiendo de ella, es recomendable buscar una nueva abstracción.

La desventaja de tener jerarquías demasiados profundas es la dificultad de comprender su comportamiento.

10: *Búsqueda de la uniformidad en las reglas para los nombres*

Es importante para las personas que vayan a reutilizar los componentes manejar un limitado conjunto de nombres de clases y métodos. Un buen ejemplo de esta característica se tiene en las bibliotecas de Eiffel, donde todos los nombres de operaciones en la biblioteca de estructuras de datos han sido estandarizados.

En caso de no existir una notación estándar en la organización se puede adoptar algún convenio entre los desarrolladores, o emplear alguna notación ampliamente aceptada como puede ser la notación húngara. A continuación, y como referencia, se citan las reglas de Ottinger para nombres de variables y clases:

- *Utilizar nombres pronunciables.*
- *Evitar nombres codificados.*
- *No ser demasiado ingenioso pensando nombres, o sólo serán entendidos por las personas que coincidan en el sentido del humor o que recuerden la broma.*
- *Cuando un concepto pueda representarse por varias palabras elegir una.*
- *Evitar nombres con doble sentido o ambiguos.*

- *Utilizar nombres o frases con algún verbo.*
- *Utilizar nombres del dominio de la solución.*
- *Utilizar nombres del dominio del problema.*
- *Nada es intuitivo.*
- *Evitar nombres o acrónimos que tengan otros significados.*
- *Cuidar que los nombres mantengan su significado dentro cualquier contexto.*
- *No crear contextos artificiales.*
- *Cuidar la forma en que se deshacen las ambigüedades.*

11: *Mantener las signaturas de los métodos consistentes*

Los métodos que realizan funciones similares deben tener los mismos nombres, y devolver el mismo tipo de dato.

Todas las nociones que se han comentado referentes al diseño orientado al objeto se reafirman en lo que Robert C. Martin [Martin97a] denomina los ***principios del DOO***. Estos principios son:

1. El principio de abierto/cerrado.
2. El principio de sustitución de Liskov.
3. El principio de inversión de dependencia.
4. El principio de separación de la interfaz.
5. El principio de equivalencia reutilización/revisión.
6. El principio de cierre común.
7. El principio de reutilización común.
8. El principio de dependencia acíclica.
9. El principio de las dependencias estables.
10. El principio de las abstracciones estables.

7. El principio de abierto/cerrado

> *"Todos los sistemas cambian durante sus ciclos de vida. Esto debe tenerse en mente cuando se pretende que los sistemas desarrollados perduren más de la primera versión"*
>
> *Ivar Jacobson [Jacobson92]*

Existen muchas heurísticas asociadas con el DOO. Por ejemplo: "*todas las variables miembro deben ser privadas*", o "*las variables globales deben evitarse*", o "*la utilización de identificación de tipos en tiempo de ejecución es peligroso*". El principio de diseño que subyace en muchas de estas heurísticas es el principio abierto/cerrado, enunciado por Bertrand Meyer [Meyer88]. Este principio dice lo siguiente:

> Las entidades software (*clases, módulos, funciones...*) deben estar abiertas para su extensión, pero cerradas para su modificación.
>
> *Bertrand Meyer [Meyer88]*

Este es el principio constituye la guía más importante para un diseñador de software orientado al objeto. El diseño de módulos cuyo comportamiento pueda ser modificado sin realizar modificaciones en el código fuente de dichos módulos, es la característica en la que se basan los beneficios del diseño orientado al objeto.

La naturaleza, simultáneamente abierta y cerrada de los sistemas orientados al objeto da soporte a la facilidad de mantenimiento, de reutilización y de verificación. Así, un sistema reutilizable debe estar abierto, en el sentido de que tienen que ser fáciles de extender, y cerrados en el sentido de que deben estar listos para ser utilizados [Graham96].

Cuando un único cambio en un programa produce una cascada de cambios en los módulos dependientes, el programa exhibe unos atributos no deseables debidos a un mal diseño. Así, el programa se convierte en un programa frágil, rígido, impredecible y en consecuencia **NO REUTILIZABLE.** Este principio ataca este vicio de forma directa, expresando la idea de que nunca se debe cambiar el diseño de los módulos. Cuando cambien los requisitos, se extiende el comportamiento de los módulos añadiendo nuevo código, pero nunca cambiando el código que ya funciona [Martin96a].

En resumen, los módulos que cumplen el principio abierto/cerrado tienen dos atributos primarios:

1. **Están abiertos para su extensión**

   Eso implica que el comportamiento de los módulos puede ser extendido. Se puede hacer que un módulo se comporte de formas nuevas cuando los requisitos de la aplicación cambien.

2. **Están cerrados para su modificación**

   El código fuente del módulo es inalterable. No se permite realizar cambios al código fuente.

La clave de este principio está en la abstracción. Se pueden crear abstracciones que sean fijas y que representen un grupo ilimitado de posibles comportamientos. Las abstracciones son un conjunto de clases base, y el grupo ilimitado de los posibles comportamientos viene representado por todas las posibles clases derivadas. De esta forma un módulo esta cerrado para su modificación al depender de una abstracción que es fija. Pero el comportamiento del módulo puede ser extendido mediante la creación de clases derivadas.

La **Figura 5** muestra un diseño simple que no cumple el principio abierto/cerrado. Tanto la clase **cliente** como la clase **servidor** son clases concretas. La clase **cliente** usa a la clase **servidor**. Si se desea que un objeto **cliente** use un objeto **servidor** diferente, se debe cambiar la clase **cliente** para nombrar la nueva clase **servidor**.

Figura 5. Diseño que no se ajusta al principio abierto/cerrado.

La **Figura 6** muestra el diseño que cumple el principio abierto/cerrado. En este caso, la clase **ServidorAbstracto** es una clase abstracta con métodos virtuales puros. La clase **Cliente** usa esta abstracción, y por tanto los objetos de la clase **Cliente** utilizarán los objetos de las clases derivadas de la clase **ServidorAbstracto**. Si se desea que los objetos de la clase **Cliente** utilicen diferentes clases **Servidor**, se deriva una nueva clase de la clase **ServidorAbstracto**, y la clase **Cliente** permanece inalterada.

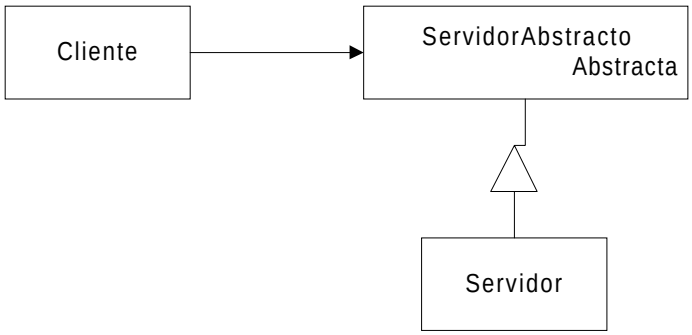

Figura 6. Diseño que cumple el principio abierto/cerrado.

7.1 Ejemplo del principio abierto/cerrado: La abstracción de la figura

Se tiene una aplicación que es capaz de dibujar círculos y cuadrados. Los círculos y los cuadrados deben ser dibujados en un determinado orden, por lo tanto se creará una lista en el orden apropiado de círculos y de cuadrados para que la aplicación recorra la lista en ese orden y dibuje cada círculo o cada cuadrado.

Utilizando una aproximación estructurada que no se ajusta al principio abierto/cerrado, utilizando lenguaje C por ejemplo, se puede resolver este problema como se muestra en el **Listado 1**. Se puede observar que tanto la estructura **Cuadrado** como la estructura **Circulo** tienen el primer elemento en común, que no es más que un identificador del tipo de figura. La función **DibujaFiguras** recorre la matriz de punteros a elementos de tipo **Figura**, examinando el tipo de figura y llamando a la función apropiada.

Sin embargo, la función **DibujaFiguras** no cumple el principio abierto/cerrado porque no está cerrada a nuevos tipos de figuras. Si se quiere extender esta función para dibujar rectángulos, se tiene que modificar la función **DibujaFiguras**.

En el **Listado 2** se presenta la solución al problema planteado, pero de forma que cumpla el principio abierto/cerrado. Como lenguaje de programación se ha elegido C++. En este caso, se ha definido la clase **Figura** como clase abstracta. Esta clase cuenta con un método virtual puro denominado **Dibuja**. Las clases **Circulo** y **Cuadrado** son clases derivadas de la clase **Figura**. Por su parte, la función **DibujaFiguras** no necesita modificarse en el caso de que se añadan nuevas figuras a la jerarquía. De acuerdo con esto se puede extender el comportamiento de la función **DibujaFiguras** sin modificar nada en absoluto de su código, cumpliendo así el principio abierto/cerrado.

Como comentario a este principio se puede decir que nunca se tiene un programa completamente cerrado. En general, no importa lo cerrado que esté un módulo, siempre existe algún tipo de cambio que demuestra que no estaba cerrado. Entonces, dado que un módulo nunca estará cerrado al 100%, el cierre debe ser estratégico. El diseñador debe seleccionar los cambios para los cuales estará cerrado su diseño.

```
#include <stdio.h>
#include <stdlib.h>
#define MAXIMO 10
#define ELEMENTOS 5

enum TipoFigura {circulo, cuadrado};
struct Figura {
        TipoFigura Tipo;
};
struct Punto {
        float x, y;
};
struct Circulo {
        TipoFigura Tipo;
        float Radio;
        struct Punto Centro;
};
struct Cuadrado {
        TipoFigura Tipo;
        float Lado;
        struct Punto SuperiorIzquierda;
};
typedef struct Figura *PunteroFigura;
void DibujaCuadrado(struct Cuadrado *);
void DibujaCirculo(struct Circulo*);
void DibujaFiguras(PunteroFigura lista[], int);

void DibujaCuadrado(struct Cuadrado *c) {
        printf("\nCuadrado");
        printf("\nPunto superior izquierda: %f %f", c->SuperiorIzquierda.x,
                        c->SuperiorIzquierda.y);
        printf("\nLongitud del lado: %f", c->Lado);
}
void DibujaCirculo(struct Circulo *c) {
        printf("\nCírculo");
        printf("\nCentro: %f %f", c->Centro.x, c->Centro.y);
        printf("\nLongitud del radio: %f", c->Radio);
}
void DibujaFiguras(PunteroFigura lista[], int n) {
        int i;
        struct Figura *f;
        for (i=0; i<n; i++) {
                f = lista[i];
                switch (f->Tipo) {
                case cuadrado:
                        DibujaCuadrado((struct Cuadrado*)f);
                break;
                case circulo:
                        DibujaCirculo((struct Circulo*)f);
                break;
                }
        }
}
void main(void) {
        struct Cuadrado *c1;
        struct Circulo *c2;
        int i;
        PunteroFigura lista[MAXIMO];

        for (i=0; i<ELEMENTOS; i++) {
                if (i%2==0) {
                        c1 = (struct Cuadrado *) malloc (sizeof(struct Cuadrado));
                        c1->Tipo = cuadrado;
                        c1->SuperiorIzquierda.x = c1->SuperiorIzquierda.y = i*2;
                        c1->Lado = i*2.5;
                        lista[i]= (struct Figura *)c1;
                }
                else {
                        c2 = (struct Circulo *) malloc (sizeof(struct Circulo));
                        c2->Tipo = circulo;
                        c2->Centro.x = c2->Centro.y = 5.0+i;
                        c2->Radio = 10.0/i;
                        lista[i]= (struct Figura *)c2;
                }
        }
        DibujaFiguras(lista, ELEMENTOS);
}
```

Listado 1. No cumple el principio abierto/cerrado (figuras.c).

```
#include <iostream.h>
#define MAXIMO 10
#define ELEMENTOS 5

class Figura {
public:
	virtual void Dibuja() = 0;
};
class Punto {
	float x, y;
public:
	Punto(float x1, float y1) {x=x1; y=y1;}
	float LeerX(void) { return x;}
	float LeerY(void) { return y;}
};
class Circulo: public Figura {
	float Radio;
	Punto Centro;
public:
	Circulo(float x1, float y1, float r1):Centro(x1,y1) { Radio = r1;}
	virtual void Dibuja() {
		cout << "\nCírculo";
		cout << "\nCentro: " << Centro.LeerX() << " " << Centro.LeerY();
		cout << "\nLongitud del radio:" << Radio;
	}
};
class Cuadrado:public Figura{
	float Lado;
	Punto SuperiorIzquierda;
public:
	Cuadrado(float x1,float y1,float l1):SuperiorIzquierda(x1, y1) { Lado = l1;}
	virtual void Dibuja()  {
		cout << "\nCuadrado";
		cout << "\nPunto superior izquierda: " <<SuperiorIzquierda.LeerX() << " "
<<
			SuperiorIzquierda.LeerY();
		cout << "\nLongitud del lado: " << Lado;
	}
};
void DibujaFiguras(Figura *lista[], int);

void DibujaFiguras(Figura *lista[], int n) {for (int i=0; i<n; lista[i++]->Dibuja());}

void main(void) {
	Figura *lista[MAXIMO];
	for (char i=0; i<ELEMENTOS; i++) {
		if (i%2==0)
			lista[i] = new Cuadrado (i*2.0, i*2.0, i+5.0);
		else
			lista[i] = new Circulo (i*3.0, i*2.5, i + 10.0);
	}
	DibujaFiguras(lista, ELEMENTOS);
}
```

Listado 2. Código que si cumple el principio abierto/cerrado (figura.cpp).

7.2 El principio abierto/cerrado y las heurísticas del DOO.

El famoso principio de Meyer está detrás de las principales heurísticas y convenciones del diseño orientado al objeto. Se pueden destacar:

- ***Todas las variables miembro de una clase deben ser privadas.***

  Esta es una de las convenciones del DOO más difundidas. Las variables miembro de las clases deben ser conocidas exclusivamente por los métodos de las clases que las definen.

Esta afirmación en relación con el principio abierto/cerrado se puede razonar desde el punto de vista que las variables de la clase pueden cambiar, y cada función que dependa de esas variables debe ser cambiada. Por tanto, ninguna función que dependa de una variable puede estar cerrada con respecto a esa variable.

- ***No se deben utilizar variables globales.***

  El argumento de esta aseveración es similar al de tener variables miembro públicas. Ningún módulo que dependa de una variable global puede estar cerrado con respecto a otro módulo que modifique el valor de dicha variable.

8. El principio de sustitución de Liskov

El principio abierto/cerrado es la base para la generación de diseños y código fáciles de mantener y de reutilizar. Los principales mecanismos que emplea dicho principio son la abstracción y el polimorfismo. En los lenguajes estáticamente tipados, como el C++, el mecanismo clave que soporta la abstracción y el polimorfismo es la herencia. Utilizando la herencia se pueden derivar clases que se ajusten las interfaces abstractas y polimórficas definidas mediante las funciones virtuales puras en las clases base abstractas.

Así, las reglas de diseño que gobiernan este uso de la herencia, las características de las mejores jerarquías de herencia y las *trampas* que llevan a la creación de jerarquías de herencias que no cumplen el principio abierto/cerrado, son los temas de que se encarga el principio de sustitución de Liskov [Martin96b].

> Lo que se quiere aquí es algo como la siguiente propiedad de sustitución: Si para cada objeto o_1 de tipo S hay un objeto o_2 de tipo T tal que para todos los programas P definidos en términos de T, el comportamiento de P no cambia cuando o_1 es sustituido por o_2, entonces S es un subtipo de T.
>
> *Barbara Liskov [Liskov88]*

Robert C. Martin enuncia el principio de la siguiente forma: "***Las funciones que usan punteros o referencias a clases base deben ser capaces de utilizar objetos de las clases derivadas sin tener conocimiento de ello***" [Martin96b], y también "***Las clases derivadas deben ser utilizables a través de la interfaz de la clase base, sin necesidad de que el usuario conozca la diferencia***" [Martin97a].

La importancia de este principio es obvia cuando se consideran las consecuencias de violarlo. Si una función no cumple el principio de Liskov, entonces la función que utiliza un puntero o una referencia a una clase base tiene que conocer de forma explícita todas las clases derivadas de dicha clase base. Por lo tanto, esta función violaría el principio abierto/cerrado porque tendría que ser modificada cada vez que se creara una nueva clase derivada de la clase base.

8.1 Ejemplo de violación del principio de Liskov

Supóngase que una aplicación que funciona correctamente y que está instalada en diferentes lugares utiliza la clase **Rectangulo** que se muestra a continuación en el **Listado 3**:

```
class Rectangulo {
    double alto, ancho;
public:
    Rectangulo(double _alto, double _ancho) {alto=_alto; ancho=_ancho;}
    void EstableceAlto(double tmp)  {alto=tmp;}
    void EstableceAncho(double tmp) {ancho=tmp;}
    double Alto()  const {return alto;}
    double Ancho() const {return ancho;}
};
```

Listado 3. Clase Rectangulo.

En un momento dado, los requisitos de los usuarios cambian y exigen que se pueda trabajar también con cuadrados. Si se ve la herencia como la relación semántica **es_un** (***ISA***), cualquier tipo de objeto nuevo que cumpla la relación **es_un** con algún tipo de objeto ya creado, puede derivarse de la clase del objeto ya existente. Claramente, un cuadrado es un rectángulo para todos los propósitos. Dado que se cumple la relación **es_un**, se puede derivar la clase **Cuadrado** de la clase **Rectangulo**, como se muestra en la **Figura 7**.

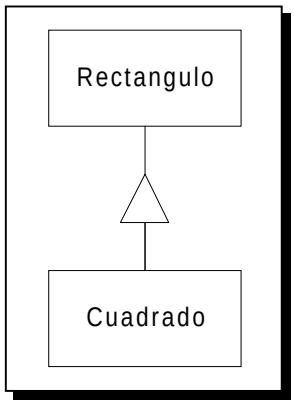

Figura 7. El cuadrado es_un rectángulo.

La utilización de la herencia es considerada por muchos como una de las técnicas fundamentales de la orientación al objeto, así en el ejemplo un cuadrado es un rectángulo, y por lo tanto la clase **Cuadrado** debe derivarse de la clase **Rectangulo**, como puede verse en el **Listado 4**.

```
class Cuadrado: public Rectangulo {
public:
     Cuadrado(double lado):Rectangulo(lado, lado){}
     void EstableceAlto(double lado) {
         Rectangulo::EstableceAlto(lado);
         Rectangulo::EstableceAncho(lado);
     }
     void EstableceAncho(double lado) {
         Rectangulo::EstableceAlto(lado);
         Rectangulo::EstableceAncho(lado);
     }
};
```

Listado 4. Clase Cuadrado.

Sin embargo, esto puede conducir a unos problemas bastante sutiles e importantes, que no se suelen prever hasta el momento de la codificación.

En primer lugar se puede apreciar que aunque **Cuadrado** no necesita las dos variables miembro (*alto*, *ancho*), las hereda y obliga a que las funciones **EstableceAlto** y **EstableceAncho** deban ser redefinidas en la clase **Cuadrado** debido a que en un cuadrado el alto y el ancho tienen el mismo valor.

Teniendo en cuenta esta redefinición (*ver **Listado 4***) el programa principal que aparece en el **Listado 5** funcionará sin problemas, debido a que al cambiar su altura cambia automáticamente su anchura y viceversa, permaneciendo las invariantes de la clase **Cuadrado** intactas.

```
void main (void) {
      Cuadrado c1(2);
      cout << c1.Alto() << " " << c1.Ancho() << endl; //Salida: 2 2

      c1.EstableceAlto(5);
      cout << c1.Alto() << " " << c1.Ancho() << endl; //Salida 5 5

      c1.EstableceAncho(10);
      cout << c1.Alto() << " " << c1.Ancho() << endl; //Salida 10 10
}
```

Listado 5. Utilización de la clase Cuadrado sin problemas.

Pero, ¿qué sucedería con alguna función que recibiese una referencia o un puntero a un objeto **Rectangulo** y modificase su altura o su anchura? Pues simplemente que la función fallaría si a dicha función se le pasa un objeto **Cuadrado**.

Por ejemplo, considérese una función que modifica la relación de aspecto entre la anchura y la altura, de forma que ajusta la anchura a la mitad de su altura (*ver **Listado 6***).

```
void CambiaAspecto(Rectangulo &r) {
    r.EstableceAncho(r.Alto()*.5);
}
```

Listado 6. Función CambiaAspecto.

Si se tiene el programa principal del, al llamar a la función **CambiaAspecto** con un objeto **Cuadrado** ésta no mantendrá las invariantes del cuadrado porque llama al método **EstableceAncho** de la clase **Rectangulo** y no cambiará su altura. Esta es una clara violación del principio de Liskov. El motivo del fallo es que los métodos **EstableceAncho** y **EstableceAlto** no han sido declarados como virtuales en la clase **Rectangulo**.

```
void main (void) {
    Rectangulo r1(8, 7);
    cout << r1.Alto() << " " << r1.Ancho() << endl; //Salida: 8 7

    CambiaAspecto(r1);
    cout << r1.Alto() << " " << r1.Ancho() << endl; //Salida: 8 4

    Cuadrado c1(2);
    cout << c1.Alto() << " " << c1.Ancho() << endl; //Salida: 2 2

    CambiaAspecto(c1);
    cout << c1.Alto() << " " << c1.Ancho() << endl; //Salida: 2 1
}
```

Listado 7. Pérdida de las invariantes del Cuadrado con la función CambiaAspecto.

Esto es muy fácil de solucionar, pero el problema de fondo es que, cuando la creación de una clase derivada provoca cambios en la clase base, siempre implica un fallo de diseño. En el **Listado 8** aparecen las definiciones de la clase **Rectangulo** y de la clase **Cuadrado**.

Si con la definición de las clases **Rectangulo** y **Cuadrado** del **Listado 8** se probase la función **CambiaAspecto** del **Listado 6**, con el programa principal del **Listado 9**, la función **CambiaAspecto** no cumpliría su cometido al recibir un objeto de la clase **Cuadrado**, ya que llamaría al método **EstableceAncho** redefinido en la clase

Cuadrado, cambiando así su altura y su anchura, pero por el contra se habrá conseguido mantener las invariantes del cuadrado inalteradas.

```
class Rectangulo {
    double alto, ancho;
public:
    Rectangulo(double _alto, double _ancho) {alto=_alto; ancho=_ancho;}
    virtual void EstableceAlto (double tmp) {alto=tmp;}
    virtual void EstableceAncho(double tmp) {ancho=tmp;}
    double Alto()  const {return alto;}
    double Ancho() const {return ancho;}
};

class Cuadrado: public Rectangulo {
public:
    Cuadrado(double lado): Rectangulo(lado, lado){}
    virtual void EstableceAlto(double lado) {
      Rectangulo::EstableceAlto(lado);
      Rectangulo::EstableceAncho(lado);
    }
    virtual void EstableceAncho(double lado) {
      Rectangulo::EstableceAlto(lado);
      Rectangulo::EstableceAncho(lado);
    }
};
```

Listado 8. Nueva definición de las clases Rectangulo y Cuadrado.

```
void main (void) {
    Cuadrado c1(2);
    cout << c1.Alto() << " " << c1.Ancho() << endl; //Salida: 2 2

    CambiaAspecto(c1);
    cout << c1.Alto() << " " << c1.Ancho() << endl; //Salida: 1 1
}
```

Listado 9. Las invariantes de la clase Cuadrado se mantienen.

Dado que un cliente de la clase **Rectangulo** puede funcionar de forma inadecuada cuando se le pasa un objeto de la clase **Cuadrado**, los objetos de la clase **Cuadrado** no pueden ser sustituidos por los objetos de la clase **Rectangulo**.

Se puede realizar codificar la función **CambiaAspecto** de forma que si detecta que recibe un objeto de clase **Cuadrado** levante una excepción, como se puede apreciar en el **Listado 10**. Pero esta solución crea una dependencia de la función **CambiaAspecto** con la clase **Cuadrado,** además cada vez que se vaya a crear una nueva clase derivada de la clase **Rectangulo**, debe añadirse una comprobación a todas las funciones que estén en conflicto, con lo que se estará violando el principio abierto/cerrado.

```
#include <iostream.h>
#include <typeinfo.h>

class Rectangulo {
        double alto, ancho;
public:
        Rectangulo(double _alto, double _ancho) {alto=_alto; ancho=_ancho;}
        virtual void EstableceAlto(double tmp) {alto=tmp;}
        virtual void EstableceAncho(double tmp) {ancho=tmp;}
        double Alto() const {return alto;}
        double Ancho() const {return ancho;}
};

class Cuadrado: public Rectangulo {
public:
        Cuadrado(double lado):Rectangulo(lado, lado){}
        virtual void EstableceAlto(double lado) {
                Rectangulo::EstableceAlto(lado);
                Rectangulo::EstableceAncho(lado);
        }
        virtual void EstableceAncho(double lado) {
                Rectangulo::EstableceAlto(lado);
                Rectangulo::EstableceAncho(lado);
        }
};

void CambiaAspecto(Rectangulo &);

void CambiaAspecto(Rectangulo &r) {
        if (typeid(r) == typeid(Cuadrado))
                throw "\nNo se puede ajustar el aspecto de un cuadrado.";
        r.EstableceAncho(r.Alto()*.5);
}

void main (void) {
        try {
                Rectangulo r1(8, 7);
                cout << r1.Alto() << " " << r1.Ancho() << endl; //Salida: 8 7

                CambiaAspecto(r1);
                cout << r1.Alto() << " " << r1.Ancho() << endl; //Salida: 8 4

                Cuadrado c1(2);
                cout << c1.Alto() << " " << c1.Ancho() << endl; //Salida: 2 2

                CambiaAspecto(c1);
                cout << c1.Alto() << " " << c1.Ancho() << endl; //Levanta una excepción
        }
        catch(char *msg) {
                cout << msg << endl;
        }
}
```

Listado 10. Solución que viola el principio abierto/cerrado en la función CambiaAspecto.

De lo anterior se puede obtenerse una importante conclusión. Un modelo, visto por separado, no puede ser validado. La validez de un modelo sólo puede expresarse en términos de sus clientes. Por ejemplo, si se examina el modelo de las clases **Rectangulo** y **Cuadrado** por separado (***Listado 8***) se llega a la conclusión de que es consistente y válido, aunque como se ha demostrado que cuando se han tenido en cuenta una serie de circunstancias el diseño *se ha roto*. Por tanto, para considerar si un diseño es adecuado, no basta con estudiar el diseño aislado, sino que se deben considerar las circunstancias que hagan los usuarios del diseño.

Volviendo al ejemplo del rectángulo y el cuadrado, se concluye que un cuadrado puede ser un rectángulo, pero un objeto **Cuadrado** no es un objeto **Rectangulo**, porque en términos de comportamiento, el comportamiento de un objeto **Cuadrado** no es consistente con el comportamiento de un objeto **Rectangulo**.

El principio de Liskov deja claro que en DOO la relación ISA se refiere al comportamiento. No al comportamiento privado intrínseco, sino al comportamiento externo público, que es el comportamiento del que dependen los clientes.

Así, para cumplir el principio de Liskov, y por tanto el principio abierto/cerrado, todas las clases derivadas deben cumplir el comportamiento que esperan los clientes de sus clases bases.

Existe una fuerte relación entre el principio de Liskov y el concepto de diseño por contrato expuesto por Bertrand Meyer [Meyer88]. Según esta forma de diseñar, los métodos de las clases tienen precondiciones y postcondiciones. Las precondiciones deben ser ciertas para que se ejecute el método, y después de su ejecución, el método de garantizar que se cumple las postcondiciones.

Meyer enuncia la siguiente regla para las precondiciones y postcondiciones para las clases derivadas:

> *... cuando se redefine un método (en una clase derivada), sólo se puede reemplazar su precondición por una más suave, y su postcondición por una más fuerte.*
>
> *Bertrand Meyer [Meyer88]*

Lo que quiere decir esta famosa regla es que cuando se usa un objeto a través de la interfaz de su clase base, los usuarios conocen sólo las precondiciones y las postcondiciones de la clase base. Por tanto, los objetos derivados no pueden ser tales que sus precondiciones sean más estrictas que las de su clase base, esto es, deben aceptar cualquier cosa que la clase base pueda aceptar. Además, las clases derivadas deben cumplir, al menos, todas las postcondiciones que cumpla la clase base.

Ciertos lenguajes, como Eiffel, soportan directamente las precondiciones y las postcondiciones, es decir, se declaran y mediante sistemas en tiempo de ejecución se verifican. C++ no dispone de esta característica, por lo que el soporte del diseño por contrato debe hacerse manualmente considerando las precondiciones y postcondiciones de cada método, y asegurar que no se viola la regla de Meyer.

9. El principio de inversión de dependencia

Un software que cumple sus requisitos pero que presenta alguna, o todas, de las características siguientes tiene un mal diseño:

1. ***Rigidez:*** Es difícil de cambiar porque cada cambio tiene demasiados efectos en otras partes del sistema.
2. ***Fragilidad:*** Cuando se realiza un cambio, partes inesperadas del sistema dejan de funcionar.
3. ***Inmovilidad:*** Es difícil de reutilizar en otras aplicaciones porque no puede separarse de la aplicación actual.

La interdependencia de los módulos dentro de un diseño es lo que provoca que un diseño sea rígido, frágil e inmóvil.

El principio de inversión de dependencia indica la dirección que tienen que tomar todas las dependencias en un diseño orientado al objeto. Así, los detalles deben depender de

las abstracciones, pero las abstracciones no deben depender de los detalles. Esto es, todas las funciones y las estructuras de datos de alto nivel deben ser completamente independientes de las funciones y estructuras de datos de bajo nivel.

El principio de inversión de dependencia puede enunciarse como sigue:

> A. Los módulos de alto nivel no deben depender de los módulos de bajo nivel. Ambos deben depender de las abstracciones.
>
> B. Las abstracciones no deben depender de los detalles. Los detalles deben depender de las abstracciones.
>
> *Robert C. Martin [Martin96c]*

La palabra inversión se debe a que en los métodos más tradicionales de desarrollo de software, como pueden ser el Análisis y el Diseño Estructurado, se tiende a crear estructuras software en las que los módulos de alto nivel dependen de los módulos de bajo nivel, y en los que las abstracciones dependen de los detalles. De hecho, en los métodos estructurados uno de los objetivos es definir una jerarquía de subprogramas que describen como los módulos de alto nivel realizan llamadas a los módulos de bajo nivel. Sin embargo, la estructura de dependencia de un programa orientado al objeto bien diseñado está invertida con respecto a la estructura de dependencia de los métodos estructurales tradicionales.

Los módulos de alto nivel contienen las políticas de decisiones y los modelos de negocio de las aplicaciones. Estos modelos contienen la identidad de la aplicación. Cuando estos módulos dependen de los módulos de bajo nivel, los cambios de los módulos de bajo nivel tendrían efectos directos en los módulos de alto nivel, y podrían forzar cambios en ellos.

Este planteamiento no tiene sentido. Los módulos de alto nivel son los que deben forzar los cambios en los módulos de bajo nivel, pero nunca al revés. Por tanto, los módulos de alto nivel nunca deben depender de los módulos de bajo nivel de ninguna forma.

Cuando los módulos de alto nivel dependen de los módulos de bajo nivel, es muy difícil reutilizarlos en contextos diferentes, convirtiéndose en un software inmóvil. Sin embargo, cuando los módulos de alto nivel son independientes de los módulos de bajo nivel, los módulos de alto nivel pueden ser reutilizados de forma bastante simple.

9.1 Ejemplo del principio de inversión de dependencias

Como ejemplo se va a tomar un programa sumamente simple, el cual va a tener la misión de mandar los caracteres que se introduzcan por el teclado a un fichero en disco. El diagrama de estructura que se correspondería con dicho programa se muestra en la **Figura 8**.

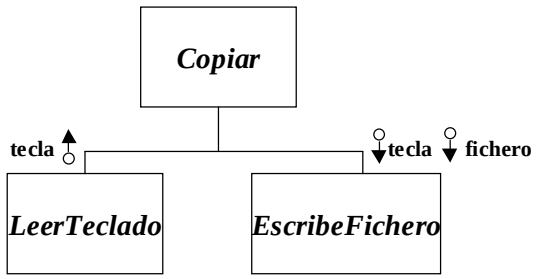

Figura 8. Diagrama de estructura del programa Copiar.

En el diagrama de estructura de la **Figura 8** se puede apreciar que se tienen tres módulos, de forma que el módulo **Copiar** llama a los otros dos, como se puede corroborar en el **Listado 11**.

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>

void Copiar(char *);
char LeerTeclado(void);
void EscribeFichero(char *, char);


void Copiar(char *cad) {
	char tecla;

	while ((tecla = LeerTeclado()) != EOF)
		EscribeFichero(cad,tecla);
}

char LeerTeclado(void) {
	char tecla;

	tecla=getch();
	tecla != 13 ? printf("%c", tecla) : printf("\n");

	return tecla;
}

void EscribeFichero(char *fichero, char tecla) {
	FILE *out;

	if ((out = fopen(fichero, "a+")) == NULL) {
		fprintf(stderr, "Error al crear el fichero.\n");
		exit(-1);
	}
	tecla != 13 ? fprintf(out, "%c", tecla) : fprintf(out, "\n");
	fclose(out);
}

void main(void) {
	Copiar("pp.txt");
}
```

Listado 11. Programa Copiar.

Los dos módulos de más bajo nivel son reutilizables, se pueden utilizar en otros programas para acceder al teclado y para guardar caracteres en un fichero de texto. Esta es la misma reutilización que se obtiene de las rutinas de una biblioteca.

Sin embargo, el módulo **Copiar**, que es el que encierra la política del proceso y sería deseable reutilizar, no se puede reutilizar en ningún otro proceso que no haga referencia al teclado y a un fichero. Por lo tanto, se tiene que el módulo **Copiar** es dependiente del disco y no puede reutilizarse en otro contexto diferente.

Así, si se quisiese añadir una nueva funcionalidad al programa **Copiar**, por ejemplo que pudiese mandar los datos a un fichero de texto o a la impresora, habría que modificar el módulo **Copiar** añadiendo una condición que seleccione el dispositivo en función de una *bandera*, ver **Listado 12**. Esto añade cada vez más interdependencias en el sistema, de forma que cuantos más dispositivos se introduzcan mayor será el grado de dependencia del módulo **Copiar** con varios módulos de bajo nivel. Como consecuencia se habrá obtenido un código rígido y frágil.

```
/* Tipos de dispositivos */
enum Dispositivo {disco, impresora};

/* ..............................*/


void Copiar(enum Dispositivo dev, char *cad) {
        char tecla;

        while ((tecla = LeerTeclado()) != EOF)
                if (dev==disco)
                        EscribeFichero(cad,tecla);
                else
                        EscribeImpresora(tecla);
}
```

Listado 12. Módulo Copiar modificado y más dependiente de los módulos de más bajo nivel.

Para solventar estas dependencias del módulo de alto nivel (**Copiar**) de los módulos de bajo nivel (**EscribeFichero, EscribeImpresora**) se debe buscar la forma de independizar el módulo **Copiar** de los detalles que él controla, para que de esta forma pueda ser reutilizado sin problemas, esto es, se debe buscar un módulo que copie caracteres de cualquier dispositivo de entrada a cualquier dispositivo de salida, para lo cual se debe tener presente el principio de inversión de dependencias.

El diagrama de clases de la **Figura 9** muestra una clase **Copiar** que contiene dos clases abstractas, una clase abstracta **Lector** y otra clase abstracta **Escritor**. De esta forma se tiene un ciclo trivial en el que la clase **Copiar** obtiene un carácter del **Lector** y se la manda al **Escritor**, pero de forma totalmente independiente de los módulos de bajo nivel. De esta forma se han invertido las dependencias, la clase **Copiar** depende de abstracciones, y los lectores y escritores especializados dependen de las mismas abstracciones.

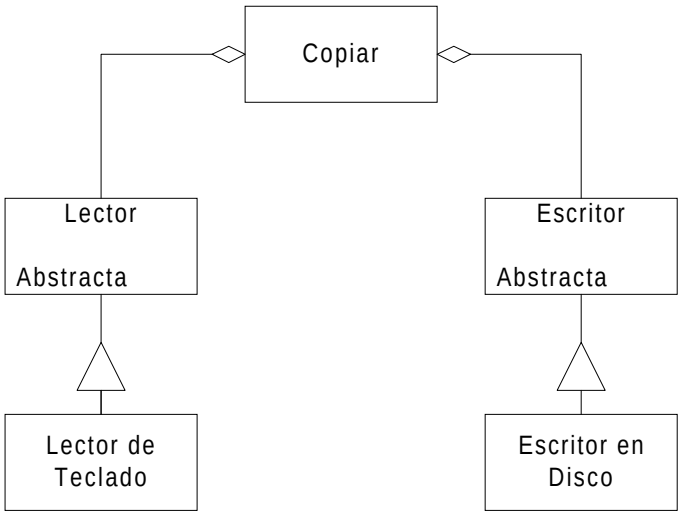

Figura 9. El programa Copiar con un diseño orientado al objeto.

De esta forma ahora se puede reutilizar la clase **Copiar** de forma independiente de los dispositivos físicos. Se pueden añadir nuevos tipos de *lectores* y de *escritores* sin que la clase **Copiar** dependa en absoluto de ellos.

La implementación que se corresponde con el diseño de la Figura 9, se tiene en el **Listado 13**.

```
#include <iostream.h>
#include <fstream.h>
#include <string.h>

class Lector {
public:
	virtual char leer() = 0;
};

class Escritor {
public:
	virtual void escribir(char) = 0;
};

class LectorTeclado : public Lector {
public:
	virtual char leer() {
		char tecla;
		cin.get(tecla);
		return tecla;
	}
};

class EscritorFichero :public Escritor {
	char nombre[25];
public:
	EscritorFichero(char *cad) {strcpy(nombre,cad);}
	virtual void escribir(char tecla) {
		fstream fichero;

		fichero.open(nombre, ios::app);

		fichero << tecla;

		fichero.close();
	}
};

void Copiar(Lector& l, Escritor& e) {
	char tecla;
	while ((tecla=l.leer()) != EOF)
		e.escribir(tecla);
};


void main (void) {
	EscritorFichero E1("pp.txt");
	LectorTeclado L1;

	Copiar (L1, E1);
}
```

Listado 13. Solución al problema del programa de copia.

9.2 Niveles

Según expresa Grady Booch [Booch96b] "*... todas las arquitecturas orientadas al objeto bien estructuradas tienen niveles claramente definidos, donde cada nivel ofrece algún conjunto de servicios coherentes a través de una interfaz bien definida y controlada*".

Sin embargo, una interpretación errónea de esta afirmación puede llevar al diseñador a crear una estructura similar a la que se presenta en la **Figura 10**. En este diagrama, la clase que contiene la política de alto nivel utiliza un mecanismo de bajo nivel, el cual a su vea utiliza una clase de utilidad con un alto nivel de detalle. Aunque esto pueda parecer apropiado, tiene una desventaja intrínseca, la capa de política es sensible a los cambios que se produzcan en la capa de utilidad.

La dependencia es transitiva: la capa que contiene la política depende de algo que depende de una capa de utilidad, por lo tanto la capa de más alto nivel depende de la capa de más bajo nivel.

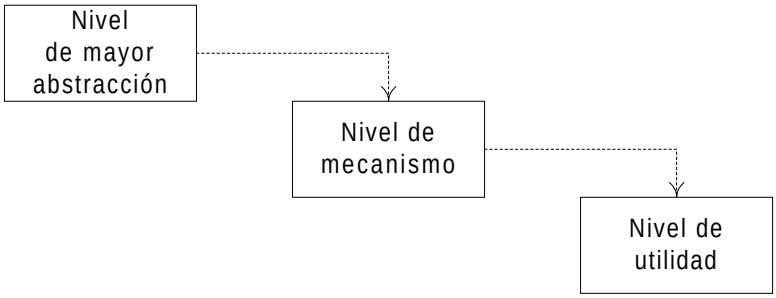

Figura 10. Estructura de niveles incorrecta.

En la **Figura 11** se presenta un modelo más apropiado. Cada uno de los niveles de más bajo nivel se representa por una clase abstracta, de forma que los propios niveles se representan por clases derivadas de estas clases abstractas. Las clases de alto nivel utilizan el nivel siguiente a través de la interfaz abstracta. De esta manera, ninguno de los niveles depende del resto de niveles. En su lugar, los niveles dependen de sus clases abstractas. No sólo se consigue eliminar la dependencia transitiva entre el nivel de mayor abstracción y el nivel de utilidad, sino que además desaparece la dependencia directa entre la capa de mayor nivel de abstracción y la capa de mecanismo.

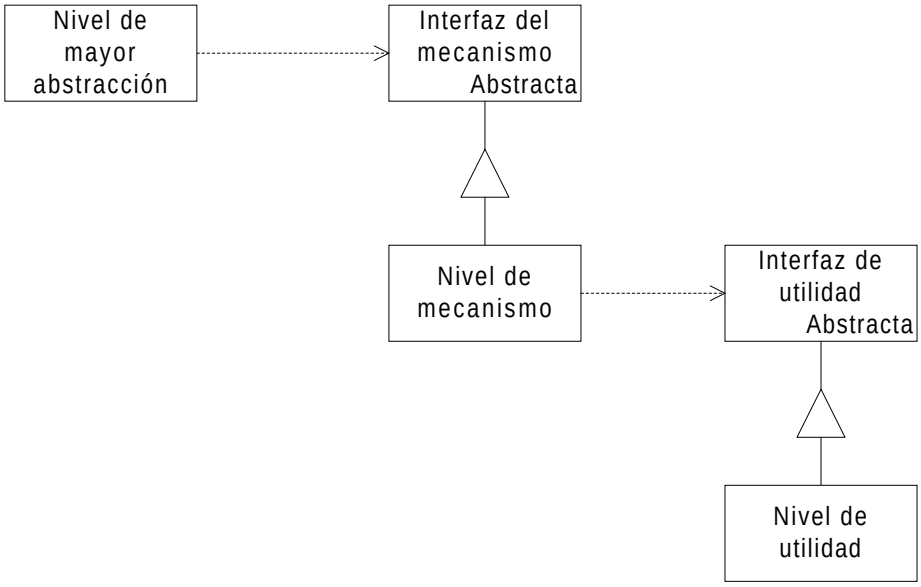

Figura 11. Solución con niveles abstractos.

Utilizando este modelo, el nivel que retiene la política del mismo no se ve afectado por los cambios en el nivel de mecanismo o en el nivel de utilidad, de modo que el nivel de mayor abstracción puede ser reutilizado en cualquier contexto en el que se definan los

módulos de bajo nivel que cumplan la interfaz de la capa de mecanismo. Así, mediante la inversión de las dependencias, se ha creado una estructura que es más flexible, resistente y móvil.

9.3 Ejemplo de aplicación del principio de inversión de dependencias y de nivelado

La inversión de dependencia puede aplicarse cada vez que una clase manda mensajes a otra. Como ejemplo, se va a considerar el caso de un botón que controla una bombilla.

Se tiene un objeto botón que controla un evento externo, el que un usuario lo haya pulsado. La bombilla se ve afectada por el comportamiento externo, si recibe el mensaje de encendido, ilumina algo, si recibe el mensaje de apagado, deja de emitir luz.

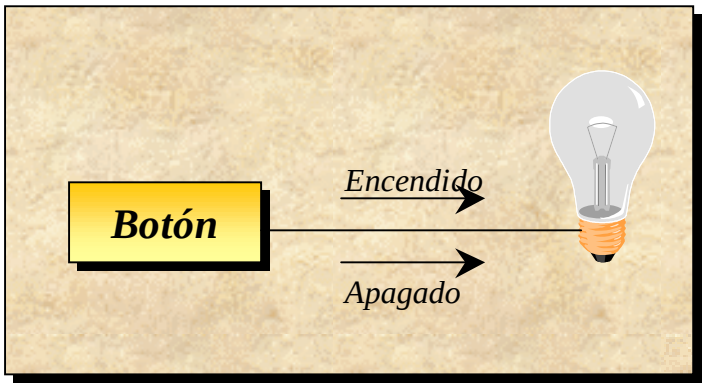

Figura 12. Ejemplo de la bombilla.

Un primer diseño, no muy acertado, puede ser el que se muestra en la **Figura 13**, donde la clase botón depende directamente de la clase bombilla.

Figura 13. Diseño incorrecto para el problema del botón y la bombilla.

Una implementación para el diseño de la **Figura 13** se muestra en el **Listado 14**. Claramente se viola el principio de inversión de dependencias, debido a que la política de alto nivel de la aplicación no se encuentra separada de los módulos de bajo nivel. De esta forma la abstracción de alto nivel depende de forma automática de los módulos de bajo nivel. En el código se puede apreciar que el módulo **boton.cpp** incluye el fichero **bombilla.h**, lo cual implica que la clase **Boton** debe modificarse, o al menos recompilarse, cada vez que se modifica la clase **Bombilla**, impidiéndose que se reutilice la clase **Boton** para controlar cualquier otro objeto.

Para encontrar la política de alto nivel de la aplicación deben fijarse las abstracciones que subyacen en la aplicación, los elementos que permanecen invariantes cuando los detalles cambian. En el ejemplo que se está desarrollando, el botón y la bombilla, la abstracción subyacente es la detección de una acción sobre el botón, y la transmisión de dicha acción al elemento destino. Sin embargo, los mecanismos utilizados para detectar la acción del usuario, o cuál es el objeto destino son detalles irrelevantes.

Para cumplir el principio de inversión de dependencias, se debe aislar la abstracción de los detalles del problema, y hacer que los detalles dependan de las abstracciones.

```
---------------------------- bombilla.h --------------------------------
class Bombilla {
public:
       void Encendido();
       void Apagado();
};
---------------------------- boton.h -----------------------------------
enum Estado {OFF, ON};

class Bombilla;
class Boton {
       Bombilla *b1;
       Estado estado;
public:
       Boton(Bombilla& b, Estado e):b1(&b){estado = e;}
       void Deteccion();
       void CambiaEstado();
};
---------------------------- bombilla.cpp ------------------------------
#include <iostream.h>
#include "bombilla.h"

void Bombilla::Encendido() {
       cout << "Toy encendida" << endl;
}
void Bombilla::Apagado() {
       cout << "Toy apagada" << endl;
}
---------------------------- boton.cpp ---------------------------------
#include "boton.h"
#include "bombilla.h"
#include <iostream.h>

void Boton::Deteccion() {
       (estado==ON) ? b1->Encendido() : b1->Apagado();
}
void Boton::CambiaEstado() {
       estado = (estado==ON) ? OFF : ON;
}
---------------------------- prueba.cpp --------------------------------
#include "boton.h"
#include "bombilla.h"


void pulsar(Boton& b) {
       b.CambiaEstado();
}

void main(void) {
       Bombilla bombi;
       Boton boton(bombi, OFF);

       boton.Deteccion();
       pulsar(boton);
       boton.Deteccion();
}
```

Listado 14. Primera implementación del problema botón/bombilla.

En la **Figura 14** se muestra un diseño más correcto para el problema del botón y la bombilla, habiéndose separado la abstracción de la clase **Boton** de sus detalles de implementación. Una implementación en C++ que se corresponde con este diseño se tiene en el **Listado 15**.

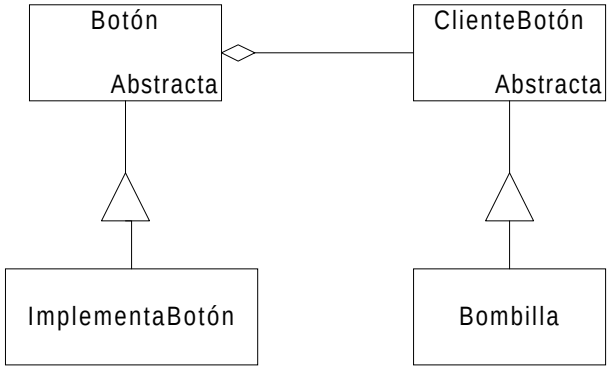

Figura 14. Diseño correcto para el problema del botón y la bombilla.

```
        ---- ClienteBoton.h ----
class ClienteBoton {
public:
  virtual void Encendido() = 0;
  virtual void Apagado() = 0;
};
             ---- boton.h ----
class ClienteBoton;

class Boton {
protected:
  ClienteBoton* cliente;
public:
  Boton(ClienteBoton& b1):cliente(&b1){};
  virtual void Deteccion() = 0;
  virtual void CambiaEstado() = 0;
};
           ----- bombilla.h ----
#include "ClienteBoton.h"

class Bombilla : public ClienteBoton {
public:
  virtual void Encendido();
  virtual void Apagado();
};
        ---- ImplementaBoton.h ----
#include "boton.h"

enum Estado {OFF, ON};

class ImplementaBoton : public Boton {
  Estado estado;
public:
  ImplementaBoton(ClienteBoton&, Estado);
  virtual Estado DevuelveEstado();
  virtual void CambiaEstado();
  virtual void Deteccion();
};
          ---- bombilla.cpp -----
#include <iostream.h>
#include "bombilla.h"

void Bombilla::Encendido() {
  cout << "Toy encendida" << endl;
}

void Bombilla::Apagado() {
  cout << "Toy apagada" << endl;
}
                  ---- ImplementaBoton.cpp -----
#include "ImplementaBoton.h"
#include "ClienteBoton.h"

ImplementaBoton::ImplementaBoton(ClienteBoton& b, Estado e):
  Boton(b) {estado = e;}

void ImplementaBoton::CambiaEstado() {
  estado = (estado==ON) ? OFF : ON;
}

Estado ImplementaBoton::DevuelveEstado() {
  return estado;
}

void ImplementaBoton::Deteccion() {
  (estado == ON) ? cliente->Encendido() : cliente->Apagado();
}
                      ---- prueba.cpp ----
#include "ImplementaBoton.h"
#include "bombilla.h"


void pulsar(ImplementaBoton& b) {
  b.CambiaEstado();
}

void main(void) {
  Bombilla bombi;
  ImplementaBoton boton(bombi, OFF);

  boton.Deteccion();
  pulsar(boton);
  boton.Deteccion();
}
```

Listado 15. Implementación para el problema del botón y la bombilla.

10. El principio de separación de la interfaz

El principio de separación de la interfaz es otro principio estructural, que combate las desventajas de las clases con interfaces grandes. Las clases con interfaces grandes son clases cuyas interfaces no están cohesionadas. Esto es, las interfaces de las clases pueden romperse en grupos de funciones miembros, donde cada grupo sirve a un conjunto de clientes diferentes.

El principio de separación de la interfaz se enuncia como sigue:

> Los clientes no deben ser forzados a depender de interfaces que no utilizan.
>
> *Robert C. Martín [Martin96d]*

Cuando los clientes se ven forzados a depender de interfaces que no utilizan, éstos se ven afectados por los cambios de dichas interfaces. Esto da como resultado un acoplamiento entre todos los clientes. De esta forma, cuando un cliente depende de una clase que contiene en su interfaz partes que el cliente no utiliza, pero que otros clientes sí lo hacen, entonces el cliente se verá afectado por los cambios en la interfaz que fuercen los otros clientes que dependen de la clase y que utilizan esa parte de la interfaz.

Se debe intentar evitar este tipo de acoplamientos, separando las interfaces donde sea posible.

10.1 Ejemplo del principio de separación de la interfaz

Supóngase que se está desarrollando un sistema de seguridad, en el que existe una clase que es **Puerta**, que puede estar *candada* o *descandada*, y que es capaz de saber si está abierta o cerrada (*Ver **Listado 1***).

```
class Puerta {
public:
    virtual void Candada()     = 0;
    virtual void Descandada()  = 0;
    virtual int  EstaAbierta() = 0;
};
```

Listado 16. Clase Puerta.

La clase **Puerta** es una clase abstracta, por lo tanto los clientes de la misma podrán utilizar objetos que cumplan la interfaz de la clase **Puerta,** con total independencia de las implementaciones de las *puertas específicas*.

Considérese ahora una puerta que cuando lleve abierta un determinado tiempo haga sonar una alarma, y cuya clase va a denominarse **PuertaConAlarma**. Para conseguir este objetivo, los objetos de la clase **PuertaConAlarma** deben comunicarse con los objetos de la clase **Temporizador**, la cual se muestra en el **Listado 17**.

```
class Temporizador {
public:
    void Registro(int tiempo, ClienteTemporizador* cl);
};
class ClienteTemporizador {
public:
    virtual void TimeOut() = 0;
};
```

Listado 17. Clase Temporizador.

Cuando un objeto **Puerta** quiera ser informado de que ha excedido el tiempo que le está permitido estar abierto, llama a la función **Registro** del **Temporizador**. Los argumentos de esta función son el límite de tiempo y un puntero a un objeto **ClienteTemporizador,** cuya función **TimeOut** será llamada cuando el tiempo expire.

En la **Figura 15** se muestra una posible solución al problema que se está tratando.

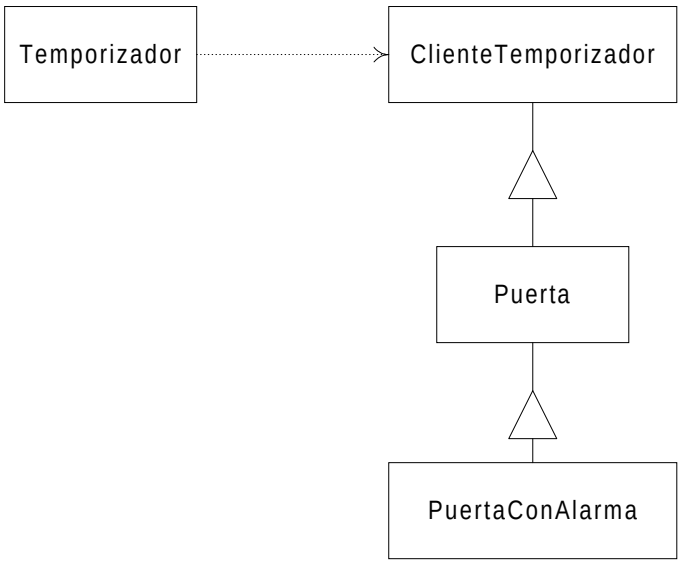

Figura 15. Diseño de la puerta con alarma.

Se ha forzado a la clase **Puerta**, y por tanto a la clase **PuertaConAlarma**, a heredar de **ClienteTemporizador**. Esta solución es problemática, debido a que ahora **Puerta** depende de **ClienteTemporizador**, y no todas las puertas necesitan de este control de tiempo, de hecho la abstracción original de **Puerta** no contemplaba en absoluto el tiempo. Según este diseño, todas las puertas que se deriven de **Puerta** y que no tengan que contemplar el temporizador, deberán implementar una función nula de **TimeOut,** además las aplicaciones que usen las especializaciones de **Puerta** tendrán que incluir la definición de la clase **ClienteTemporizador**, aunque no la use. Esto es lo que se conoce como contaminación de la interfaz. La interfaz de la clase **Puerta** ha sido contaminada con una interfaz que no requiere, sólo por el beneficio de una de sus subclases.

La solución a este problema viene por aplicar el principio de separación de la interfaz en el diseño, como se puede apreciar en la. Aquí se ha hecho uso de la herencia múltiple, de forma que la clase **PuertaConAlarma** herede de la clase **Puerta** y de la clase **ClienteTemporizador**, de esta forma los clientes de las dos clases bases podrán recibir objetos de **PuertaConAlarma**, utilizando el mismo objeto a través de interfaces separadas. Además, las clases que se deriven de la clase **Puerta,** y que no necesiten el temporizador, no tendrán necesidad de implementar funciones nulas de **TimeOut,** ni las aplicaciones que las usen tendrán que incluir la especificación de la clase **ClienteTemporizador.**

En este caso el uso de la herencia múltiple se adapta perfectamente a la solución del problema, aunque muchos autores recomiendan que no se haga uso de ella.

La separación de las interfaces podía haberse llevado de otras formas, por ejemplo mediante el uso de delegación, siguiendo patrón de adaptación[7] descrito en el GOF [Gamma95].

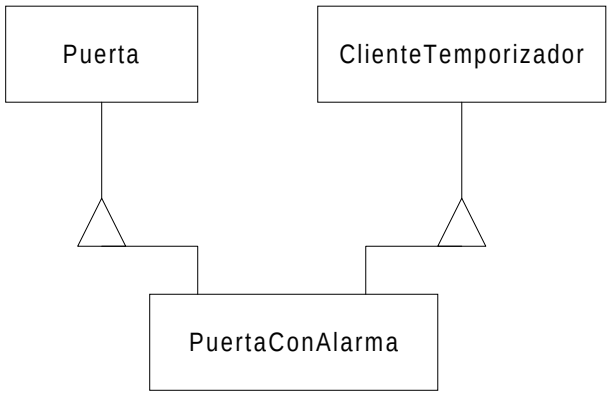

Figura 16. Solución con herencia múltiple.

11. El principio de equivalencia reutilización/revisión

Las aplicaciones crecen en tamaño y en complejidad y se requiere algún tipo de organización de alto nivel. La clase es un elemento adecuado para organizar pequeñas aplicaciones, pero es un grano demasiado fino para utilizarlo como unidad organizativa en aplicaciones grandes. Se necesita algo más mayor que la clase para organizar las aplicaciones grandes.

De esta forma los mayores expertos en metodología han identificado organizaciones de granularidad mayor a la clase. Así se pueden citar categorías de clases de Booch [Booch96], los clusters de Meyer, o más recientemente los *tres amigos* en UML hablan de paquetes[8] (*packages*) [Rational97c].

La reutilización es uno de los más aclamados principios del DOO. Sin embargo, la reutilización no viene automáticamente, no es tan simple como escribir una clase y decir que es reutilizable. En primer lugar, una clase probablemente tendrá una serie de clases que colaboran con ella. Por tanto, la clase no es reutilizable por sí sola, debe reutilizarse en conjunto con las clases con las que colabora. Por otra parte, los desarrolladores con reutilización no desean tener que mantener los cambios de las clases que ellos reutilizan. Por lo tanto se requiere un cierto mecanismo de distribución de las nuevas revisiones.

En este sentido se enuncia el principio de equivalencia reutilización/revisión, que dice:

> La granularidad de la reutilización es la granularidad de la revisión. Sólo los componentes que son revisados a través de un sistema de distribución pueden ser reutilizados de forma efectiva. Este grano es el paquete.
>
> *Robert C. Martin [Martin96e]*

[7] El patrón de adaptación (*adapter pattern o wrapper pattern*) convierte la interfaz de una clase en otra interfaz que el cliente espera. Permite que clases, que de otro modo no podrían debido a interfaces incompatibles, trabajar juntas.

[8] Un paquete es un mecanismo de propósito general para organizar elementos en grupos. Es posible incluir paquetes dentro de paquetes. Un sistema puede pensarse como un paquete de nivel superior, con todo el resto del sistema contenido en él [Rational97c].

12. El principio de cierre común

Las clases dentro de un componente deben tener un cierre común. Esto es, si una necesita ser cambiada, todas ellas pueden requerir cambios. Lo que afecta a una, afecta a todas. Según esto se puede enunciar lo que se conoce como el principio de cierre común:

> Las clases en un paquete deben estar cerradas juntas frente a los mismos tipos de cambios. Un cambio que afecta a un paquete afecta a todas las clases del paquete.
>
> *Robert C. Martin [Martin96e]*

El mantenimiento de del software tiene tanta o más importancia que la propia reutilización. Si el código de una aplicación debe cambiar, lo más interesante es que los cambios se centren en un único paquete en lugar de dispersarse por el conjunto completo de paquetes que forman el sistema.

El principio de cierre común se basa en la idea de juntar en el mismo lugar todas las clases que pueden cambiar por las mismas razones.

13. El principio de reutilización común

Este principio indica qué clases deben colocarse dentro de un paquete. Implica que las clases que tiendan a reutilizarse juntas deben permanecer juntas en el mismo paquete.

Las clases rara vez se reutilizan por separado. Generalmente se reutilizan clases que colaboran con otras clases que son parte de la abstracción reutilizable. Según el principio de reutilización común estas clases deben estar juntas en el mismo paquete.

El principio de reutilización común se enuncia como sigue:

> Las clases que pertenecen a un paquete se reutilizan juntas. Si se reutiliza una de las clases del paquete, se reutilizan todas.
>
> *Robert C. Martin [Martin96e]*

Es común que los paquetes tengan una representación física del tipo de bibliotecas compartidas o DLLs. Si se produce una nueva revisión de una DLL porque se ha producido algún tipo de cambio, se redistribuye la DLL y todas las aplicaciones que trabajan con esta DLL trabajarán con la nueva revisión.

Por tanto, cuando se depende de un paquete, se depende de cada una de las clases del paquete. De otro modo, se tendría que revalidar y distribuir más de lo necesario, con el consiguiente aumento de esfuerzo.

14. El principio de dependencia acíclica

Este principio se enuncia de la siguiente manera:

> La estructura de dependencia entre los paquetes debe ser un grafo dirigido acíclico (DAG). Esto es, no debe haber ciclos en la estructura de dependencia.
>
> *Robert C. Martin [Martin96e]*

Cuando se tiene que afrontar un proyecto de grandes dimensiones se buscan soluciones del tipo de dividir el entorno de desarrollo en paquetes. Los paquetes se convierten así en unidades de trabajo. Cuando un equipo encargado de un paquete lo tiene funcional, le asigna un número de versión y lo pone a disposición del resto de equipos, a la vez que este equipo puede seguir modificándolo en su área privada de trabajo.

Cuando se realizan nuevas revisiones de un paquete, el resto de equipos deciden si adoptan la nueva revisión inmediatamente, o si por el contrario continúan utilizando la revisión antigua.

Este es un proceso de trabajo lógico y ampliamente difundido. Sin embargo, para que funcione se debe gestionar una estructura de dependencia de paquetes, en la cual no puede haber ciclos. Si hay ciclos en esta estructura de dependencia no se podrá evitar lo que se conoce como *síndrome de la mañana siguiente*.

Los paquetes dependen unos de los otros. De forma específica, una clase de un paquete puede incluir un fichero cabecera de una clase de otro paquete diferente. Esto se muestra en un diagrama de clases como una relación de dependencia entre paquetes, **Figura 17**.

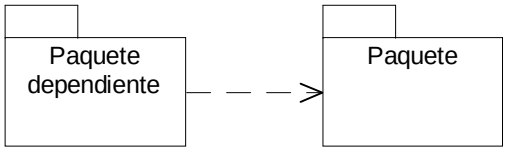

Figura 17. Dependencia entre paquetes.

En la **Figura 18**, se muestra un diagrama de paquetes. Se puede apreciar una estructura típica de paquetes ensamblados en una aplicación, esta estructura es un grafo. Los paquetes son los nodos, y las relaciones de dependencia los arcos. Como las relaciones de dependencia tienen dirección, se tiene una estructura de grado dirigido.

Con independencia del paquete que se elija para iniciar el recorrido de las relaciones dependencias es imposible volver al mismo paquete. Esta estructura no tiene ciclos, por tanto es un grafo dirigido acíclico.

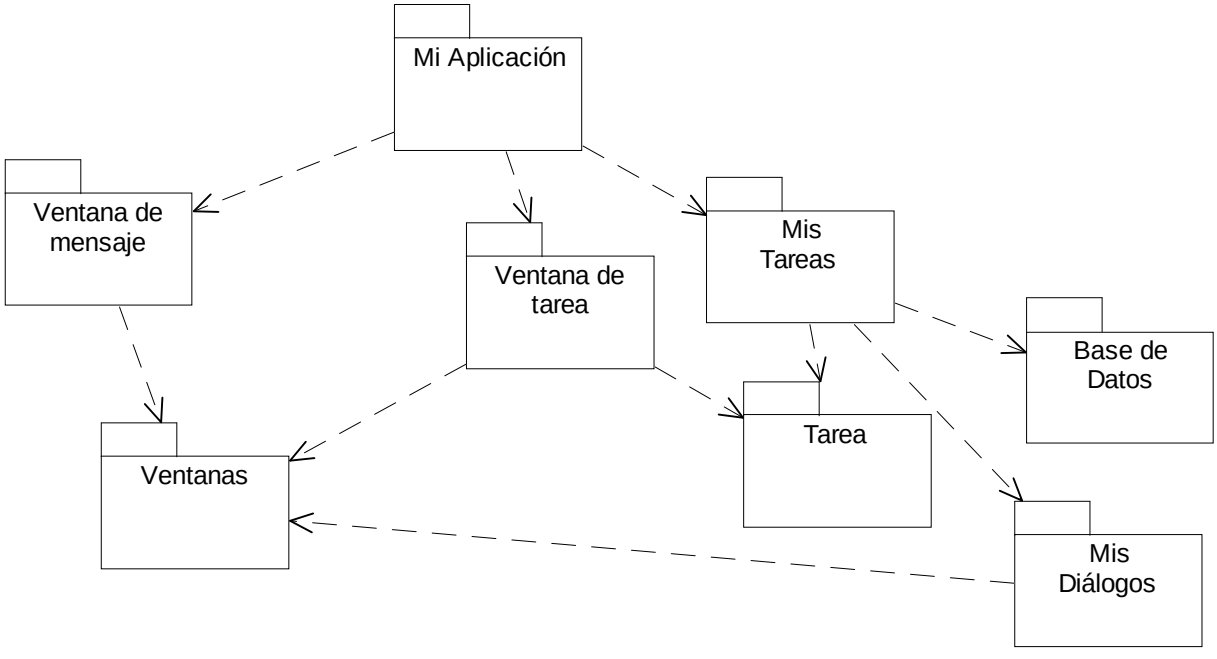

Figura 18. Diagrama de paquetes sin ciclos.

Supóngase que un cambio de requisitos fuerza a cambiar una clase en el paquete **MisDialogos** de forma que incluya una cabecera de clase de **MiAplicación**, entonces se crearía un ciclo como se muestra en la **Figura 19**.

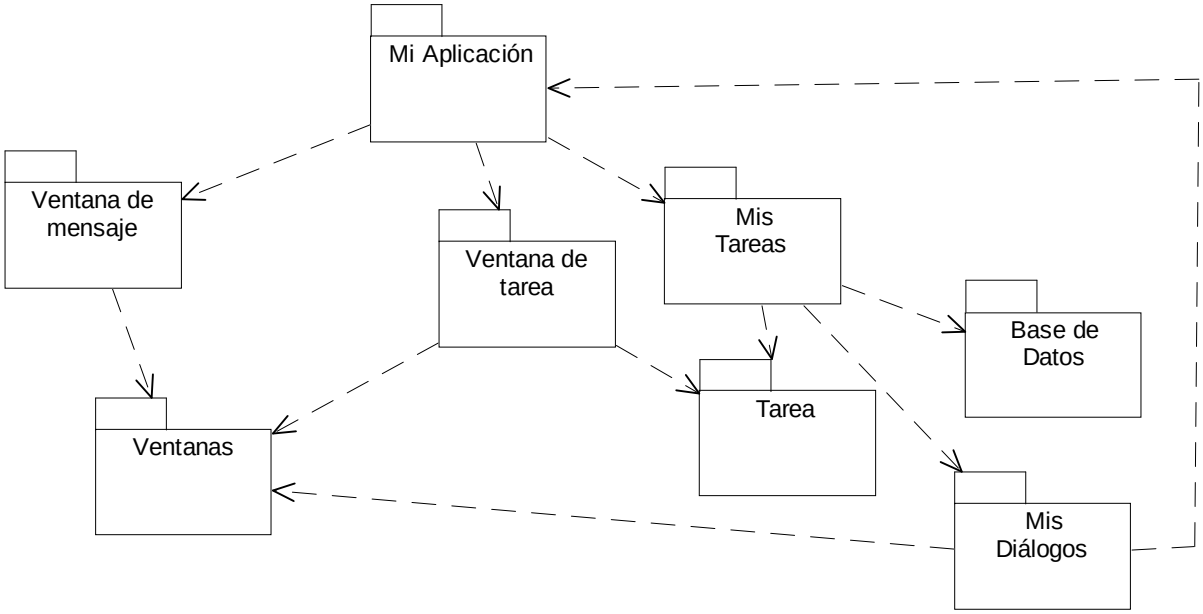

Figura 19. Diagrama de paquetes con ciclos.

Este ciclo crea algunos problemas inmediatos. El paquete **MisTareas** depende ahora de todos los paquetes del sistema, con lo que se hace muy difícil de desarrollar. El paquete **MisDiálogos** sufre del mismo mal, así **MiAplicación, MisTareas** y **MisDiálogos** deben desarrollarse al mismo tiempo, volviéndose a producir las interferencias.

Es posible romper el ciclo, y volver a tener el grafo de dependencia como un **DAG**. La forma más adecuada de hacerlo es aplicar el principio de inversión de dependencia. En el caso de la **Figura 19** se puede crear una clase base que tenga la interfaz que el paquete **MisDiálogos** necesita. Se coloca la clase base en el paquete **MisDiálogos** y se hereda dentro del paquete **MiAplicación**. Esto invierte la dependencia entre **MisDiálogos** y **MiAplicación**, rompiendo el ciclo, **Figura 20**.

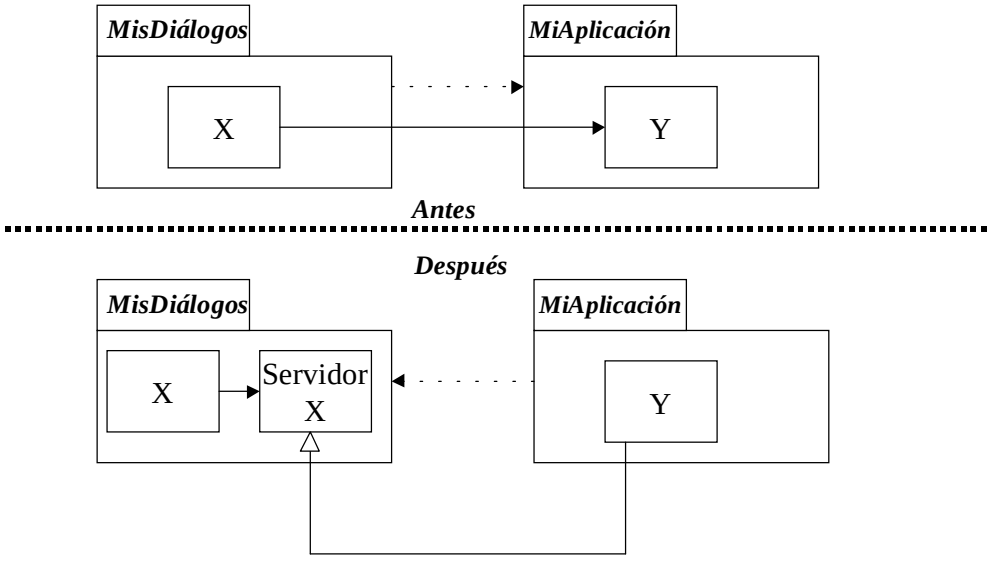

Figura 20. Eliminación del ciclo con el principio de inversión de la dependencia.

15. El principio de las dependencias estables

El principio de las dependencias estables introduce el concepto de ***buena dependencia***, que es una dependencia de algo que tiene baja volatilidad. Cuanto menor sea la volatilidad del destino de la dependencia mejor será la dependencia. De la misma manera se puede decir que una ***mala dependencia*** es una dependencia de algo que es volátil. Cuanto más volátil es el destino de la dependencia peor será la dependencia.

La volatilidad de un módulo es algo difícil de entender. La volatilidad depende de todo tipo de factores. Sin embargo, hay un factor que influye en la volatilidad que es más fácil de medir. Este factor es la estabilidad.

La estabilidad se puede definir como algo que no es fácilmente cambiable. Claramente los módulos que son más difíciles de cambiar, serán los menos volátiles, por lo tanto, cuanto más difícil de cambiar sea un módulo, más estable es, y menos volátil será.

Las clases que son muy dependientes se denominan responsables. Estas clases tienden a ser estables porque cualquier cambio tiene grandes repercusiones.

Las clases más estables de todas, son las clases que son independientes y responsables. Estas clases no tienen motivos para cambiar, y sí muchos para no hacerlo.

De acuerdo con esto, el principio de las dependencias estables se enunciaría como sigue:

> Las dependencias entre paquetes en un diseño debe hacerse buscando la dirección de la estabilidad de los paquetes. Un paquete debe depender sólo de los paquetes que son más estables que él.
>
> *Robert C. Martin [Martin97b]*

Los diseño no pueden ser completamente estáticos. Algún tipo de volatilidad será necesaria si el diseño va ser mantenido. Esto se permite mediante el principio de cierre común. Utilizando este principio se crean paquetes que son sensitivos a ciertos tipos de cambios. Estos paquetes se diseñan para ser volátiles, se espera que cambien.

Cualquier paquete que se espere que sea volátil no puede depender de otro paquete que sea difícil de cambiar, de otra forma, el paquete volátil será difícil de cambiar.

Para cumplir el principio de las dependencias estables, se debe asegurar que los módulos que se diseñan para ser inestables (*fáciles de cambiar*) no dependen de los módulos que son más estables que ellos (*difíciles de cambiar*).

15.1 Métricas de estabilidad

Robert C. Martin [Martin95] propone una serie de métricas que permiten calcular la estabilidad posicional de un paquete. Estas métricas son:

- ***Acoplamientos Aferentes (Ca):*** Se define como el número de clases fuera del paquete que dependen de las clases contenidas en el paquete.
- ***Acoplamientos Eferentes (Ce):*** Se define como el número de clases dentro del paquete que dependen de clases externas al paquete.
- ***Inestabilidad (I):*** Esta métrica está comprendida en el intervalo **[0, 1]**, de forma que **I=0** indica un paquete completamente estable, y por el contrario **I=1** indica un paquete completamente inestable. La inestabilidad se define mediante la siguiente relación:

$$I = \frac{Ce}{Ca + Ce}$$

Los términos **Ca** y **Ce** pueden calcularse contando el número de clases exteriores al paquete que tienen dependencias con las clases internas al mismo.

En el ejemplo que se presenta en la **Figura 21**, se va a proceder a calcular la estabilidad del paquete que se encuentra en el centro de la figura. Se tienen **4** clases externas al paquete que tienen dependencia con las clases internas (*la clase **A** se deriva de la clase **1**, la clase **B** contiene objetos de la clase **2**, la clase **C** contiene objetos de la clase **2**, y la clase **E** se deriva de la clase **1***), por lo tanto se tiene que **Ca=4**. Por otra parte se tiene que existen **3** relaciones de las clases interiores del paquete que tienen como destino clases exteriores al paquete, es decir existen **3** dependencias de las clases interiores del paquete con clases exteriores (*la clase **1** contiene objetos de la clase **G**, la clase **1** se deriva de la clase **H**, y la clase **2** tiene una relación de asociación con la clase **I***), por lo tanto **Ce=3**. Así, **I =3/7.**

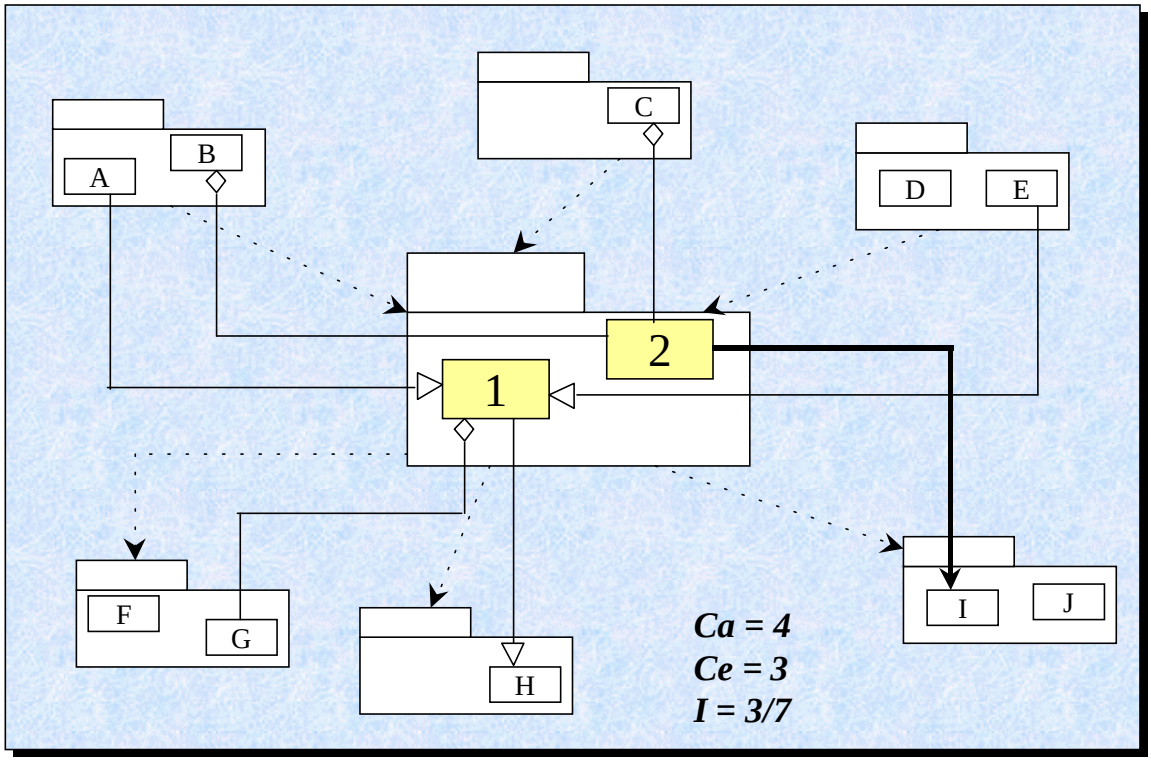

Figura 21. Ejemplo para el cálculo de la estabilidad.

Cuando la inestabilidad, **I**, es 1 significa que el paquete ningún paquete depende del paquete que se está midiendo, y por el contrario este paquete si depende de otros paquetes, siendo un paquete inestable: es no responsable y dependiente. La falta de paquetes que dependan de él no le ofrece razones para no cambiar, y los paquetes de los que depende pueden darle amplias razones para cambiar.

Por el contrario, cuando la inestabilidad es nula, **I=0**, significa que uno o varios paquetes dependen de él, pero él no depende de nadie. Es responsable e independiente, convirtiéndose en un paquete completamente estable. Los módulos que dependen de él hacen fuerza para que sea difícil de cambiar, y como él no depende de otros paquetes no se ve forzado a cambiar.

El principio de las dependencias estables dice que la métrica **I** de un paquete debe ser mayor que las métricas **I** de los paquetes de los que él depende, esto es la métrica **I** debe decrecer en la dirección de la dependencia.

Se debe tener presente que no es deseable que todos los paquetes sean completamente estables, porque el sistema no podría modificarse.

16. El principio de las abstracciones estables

Este principio establece la relación entre estabilidad y abstracción. Expresa que un paquete estable debe ser también abstracto, ya que su estabilidad no le impide ser extendido. Por otro lado, un paquete inestable debe ser concreto, debido a que la inestabilidad permite que el código concreto que encierra sea fácil de cambiar.

De acuerdo a lo expresado en el párrafo anterior, el principio de las abstracciones estables se enuncia como sigue:

> Los paquetes que son estables al máximo deben ser abstractos al máximo. Los paquetes inestables deben ser concretos. La abstracción de un paquete debe ser proporcional a su estabilidad.
>
> *Robert C. Martin [Martin97b]*

16.1 Métricas de abstracción

Se puede establecer una métrica de la abstracción de un paquete, que se va a representar por **A**. Este valor mide la proporción de clases abstractas dentro de un paquete con relación al número total de clases dentro del paquete.

$$A = \frac{Clases\ Abstractas}{Clases\ Totales}$$

La métrica **A** está definida en el intervalo **[0, 1]**. El valor **A=0** implica que el paquete no tiene clases abstractas. El valor **A=1** indica que el paquete sólo contiene clases abstractas.

Una vez definidas las métricas **I** y **A,** se puede establecer una relación entre la estabilidad y la abstracción. Se puede representar en unos ejes coordenados esta relación, colocando en el eje de ordenadas la medida de la abstracción y en el eje de abscisas la medida de la estabilidad. Si se unen los dos puntos que representan los paquetes más interesantes se obtendrá que los paquetes que son completamente estables y abstractos se concentrarán en el punto **(0, 1),** mientras que los paquetes que son completamente inestables y concretos se concentrarán en el punto **(1, 0),** como se refleja en la **Figura 22**.

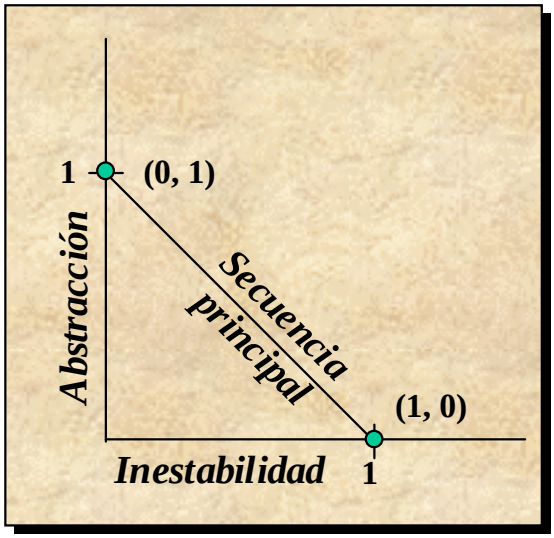

Figura 22. Secuencia principal.

Se debe tener presente que todos los paquetes no se encuentran localizados en una de estas dos posiciones. Los paquetes pueden tener diferentes grados de abstracción y estabilidad. Por ejemplo, es muy frecuente que una clase abstracta se derive de otra clase abstracta, siendo la derivación una abstracción que conlleva una dependencia, decreciendo así su estabilidad.

Dado que no se puede forzar a que todos los paquetes se encuentren localizados en los puntos **(0, 1)** o **(1, 0)**, se debe admitir la existencia de puntos dentro del gráfico ***Abstracción/Estabilidad*** que definen posiciones razonables para los paquetes. Se puede establecer donde se encuentran localizados estos puntos definiendo las áreas donde los paquetes no pueden encontrarse, es decir, encontrando las áreas de exclusión.

Así, un paquete que se encuentre en el área cercana al punto **A=0** y **I=0** será un paquete altamente estable y concreto. Esto no es deseable porque es un paquete rígido. No puede ser extendido porque no es abstracto, y además será difícil de cambiar debido a su estabilidad. De esto se deduce que un paquete bien diseñado no debe encontrarse en el área cercana al punto **(0, 0)**, así el área cercana a este punto es una zona de exclusión.

Un paquete con **A=1** y **I=1** no es deseable, quizás imposible, porque es completamente abstracto y nadie depende de él. Es rígido porque no es extensible. La zona **(1, 1)** constituye otra zona de exclusión.

La línea que une los puntos **(0, 1)** y **(1, 0)** representa los paquetes cuya abstracción se encuentra equilibrada con su estabilidad, denominándose a esta línea **secuencia principal**.

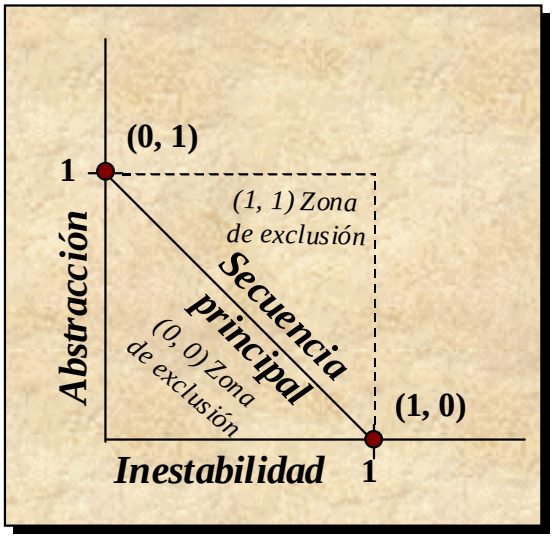

Figura 23. Zonas de exclusión.

Un paquete que se encuentre en la secuencia principal no es demasiado abstracto para su estabilidad, ni demasiado inestable para su abstracción. Tiene un número adecuado de clases concretas y abstractas en proporción a su acoplamiento aferente y a su acoplamiento eferente. Aunque claramente lo más deseable es que un paquete se encuentre en uno de los puntos extremos de la secuencia principal.

De acuerdo con lo expresado hasta ahora se puede enunciar una nueva métrica que mida la distancia de un paquete a la secuencia principal. Esta métrica se representará por **D** y será la distancia perpendicular de un paquete a la secuencia principal, y viene definida por la siguiente relación:

$$D = \left| \frac{A + I - 1}{\sqrt{2}} \right|$$

Esta métrica se encuentra definida en el intervalo **[0, ~0,707]**. Se puede normalizar esta métrica al intervalo **[0, 1]** utilizando una forma simplificada que se denominará **D'** y que vendrá representada por la relación siguiente:

$$D' = |A + I - 1|$$

17. Bibliografía

[Booch96] Booch, Grady. "*Análisis y Diseño Orientado a Objetos con Aplicaciones*". 2ª Ed. Addison-Wesley/Díaz de Santos. 1996.

[Booch96b] Booch, Grady. "*Object Solutions*". Addison-Wesley. 1996.

[Business92a] "*Definition of the Development Process*". Esprit project 5311 (Business Class), Report BC.R.TS.W3.T31. Release 2. 1992.

[Devis97] Devis Botella, Ricardo. "*C++. STL – Plantillas – Excepciones Roles y Objetos*". Paraninfo. 1997.

[Freeman87b] Freeman, P. "*A Perspective on Reusability*". IEEE Tutorial: Software Reusability (ed. P. Freeman), IEEE Computer Society Press, pp. 2-8. 1987.

[Gamma95] Gamma, Erich, Helm, Richard, Johnson, Ralph, Vlissides, John. "*Design Patterns. Elements of Reusable Object-Oriented Software*". Addison-Wesley. 1995.

[García95] García Peñalvo, Francisco José. "*Curso de C++*". Revisión 2. ALI CyL, Valladolid. 1995.

[Graham96] Graham, Ian. "*Métodos Orientados a Objetos*". 2ª Edición. Addison-Wesley/Díaz de Santos. 1996.

[Jacobson92] Jacobson, Ivar, Christerson, M., Jonsson, P., Overgaard, G. "*Object-Oriented Software Engineering: A Use Case Driven Approach*". Addison-Wesley. 1992.

[Joyanes94] Joyanes Aguilar, Luis. "*C++ a su Alcance: Un Enfoque Orientado a Objetos*". McGraw Hill. 1994.

[Joyanes96] Joyanes Aguilar, Luis. "*Programación Orientada a Objetos. Conceptos, Modelado, Diseño y Codificación en C++*". McGraw-Hill. 1996.

[Karlsson95] Karlsson, Even-André. "*Software Reuse. A Holistic Approach*". John Wiley & Sons Ltd. 1995.

[Kernighan91] Kernighan, Brian W., Ritchie, Dennis M. "*El Lenguaje de Programación C*". 2ª Edición. Prentice Hall. 1991.

[Krueger92] Krueger, Charles W. "*Software Reuse*". ACM Computing Surveys. Vol. 24, Nº 2, pp. 131-183. June 1992.

[Lea96] Lea, Doug. "*A Position Statement on Object Oriented Design*". ACM Computing Surveys, 28A(4), December 96. http://g.oswego.edu/dl/html/acmPos.html. 1996.

[Liskov88] Liskov, Barbara. "*Data Abstraction and Hierarchy*". SIGPLAN Notices, Vol. 23(5), May 1988.

[Marqués95] Marqués Corral, José Manuel. "*Jerarquías de Herencia en el Diseño de Software Orientado al Objeto*". Tesis Doctoral, Universidad de Valladolid, 1995.

[Marqués96] Marqués Corral, José Manuel. "*Reutilización y Diseño Orientado al Objeto: Informe de Resultados y Propuestas de Investigación*". En las actas de las I Jornadas de Trabajo en Ingeniería del Software, Sevilla, 14-15 de Noviembre de 1996.

[Martin92a] Martin, Robert C. "*Abstract Classes and Pure Virtual Functions*". C++ Report. June/July 1992.

[Martin93] Martin, Robert C. "*OO(A, D, P (C++))*". C++ Report. March 1993.

[Martin95] Martin, Robert C."*OO Design Quality Metrics. An Analysis of Dependencies*". ROAD. September - October, 1995.

[Martin96a] Martin, Robert C. "*The Open Closed Principle*". C++ Report, January 1996.

[Martin96b] Martin, Robert C. "*The Liskov Substitution Principle*". C++ Report, March 1996.

[Martin96c] Martin, Robert C. "*The Dependency Inversion Principle*". C++ Report. May 1996.
[Martin96d] Martin, Robert C. "*The Interface Segregation Principle*". C++ Report. August 1996.
[Martin96e] Martin, Robert C. "*Granularity*". C++ Report. November-December 1996.
[Martin97a] Martin, Robert C. "*Principles of OOD*". OMA. 1997.
[Martin97b] Martin, Robert C. "*Stability*". C++ Report. February 1997.
[McIlroy76] McIlroy, M. D. "*Mass-produced Software Components*". In J.M. Buxton, P. Naur, and B. Randell, editors, Software Engineering Concepts and Techniques; 1968 NATO Conference on Software Engineering, pp. 88-98. Van Nostrand Reinhold, 1976.
[Meyer88] Meyer, Bertrand. "*Object Oriented Software Construction*". Prentice Hall, 1988.
[Mili95] Mili, Hafedh, Mili, Fatma, Mili, Ali. "*Reusing Software: Issues and Research Directions*". IEEE Transactions on Software Engineering. Vol. 21. Nº 6, pp. 528-562. June 1995.
[Monarchi92] Monarchi, David E., Puhr, Gretchen I. "*A Research Typology for Object-Oriented Analysis and Design*". Communicatios of the ACM. Vol. 35, Nº 9, pp. 35-47. September 1992.
[Ohnjec97] Ohnjec, Viktor. "*Converging on OOAD Agreement*". Applications Development Trends, Vol. 4, Nº2, Feb. 1997, http://www.admag.com/feb97/fe203.htm. 1997.
[Piattini96] Piattini Velthuis, Mario Gerardo. "*Tecnología Orientada al Objeto*". En las notas del curso Tecnología Orientada al Objeto. ALI-CyL, Valladolid, Noviembre 1996.
[Prieto-Díaz91] Prieto-Díaz, Rubén, Arango, Guillermo. "*Domain Analysis and Software Systems Modeling*". IEEE Computer Society Press. 1991.
[Rational97a] Rational Software Corporation. "*Unified Modeling Language. UML Summary*". Rational Software Corporation. Version 1.0. 13 January 1997.
[Rational97b] Rational Software Corporation. "*Unified Modeling Language. UML Semantics*". Rational Software Corporation. Version 1.0. 13 January 1997.
[Rational97c] Rational Software Corporation. "*Unified Modeling Language. UML Semantics Appendix M1 – UML Glossary*". Rational Software Corporation. Version 1.0. 13 January 1997.
[Rational97d] Rational Software Corporation. "*Unified Modeling Language. Notation Guide*". Rational Software Corporation. Version 1.0. 13 January 1997.
[Rumbaugh96] Rumbaugh, James, Blaha, Michael, Premerlani, William, Eddy, Frederick, Lorensen, William. "*Modelado y Diseño Orientados a Objetos. Metodología OMT*". Prentice Hall. 1996.
[Sparks96] Sparks, Steve, Benner, Kevin, Faris, Chris. "*Managing Object Oriented Framework Reuse* ". IEEE Computer. September 1996, pp. 52-61. 1996