

Desarrollo Utilizando Patrones Software

M.I. Capel

ETS Ingenierías Informática
y Telecommunicación
Universidad de Granada
Email: manuelcapel@ugr.es

Desarrollo de Software

- 1 Introducción
- 2 Patrones de diseño
 - Patrones de diseño estructurales
 - Patrones de diseño comportamentales
- 3 Patrones y Calidad del Software
- 4 Patrones Arquitectónicos

- 1 Introducción
- 2 Patrones de diseño
 - Patrones de diseño estructurales
 - Patrones de diseño comportamentales
- 3 Patrones y Calidad del Software
- 4 Patrones Arquitectónicos

- 1 Introducción
- 2 Patrones de diseño
 - Patrones de diseño estructurales
 - Patrones de diseño comportamentales
- 3 Patrones y Calidad del Software
- 4 Patrones Arquitectónicos

- 1 Introducción
- 2 Patrones de diseño
 - Patrones de diseño estructurales
 - Patrones de diseño comportamentales
- 3 Patrones y Calidad del Software
- 4 Patrones Arquitectónicos

Conceptos básicos y motivación

- Un patrón permite capturar el conocimiento más importante acerca del diseño de la solución a un problema, que permite su posterior reutilización
 - 1 Se aplica a un contexto
 - 2 Resuelve un problema
 - 3 Generan *indirectamente* una solución
- La idea de patrón proviene de la *arquitectura de edificios*

Christopher Alexander (Buildings Architect)

Each pattern describes a problem that occurs over and over again in our environment and then describes the core of the solution to that problem in such a way that you can use the solution a million times over without ever doing it the same way twice

Marcos de trabajo: definición

Definition (Framework)

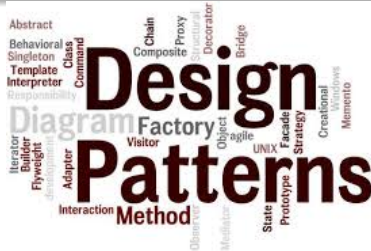
"Reusable minimum architecture that provides the generic structure and behaviour for a family of software abstractions, along with a context . . . that specifies their collaboration and their use within a given domain"

- Los marcos de trabajo se suelen aplicar sin cambios
- Proporcionan una estructura de clases para adaptar
- Arquitectónicamente más grandes que los patrones, pero más *concretos*

Patrones vs. marcos de trabajo

- Un patrón de diseño conceptualmente es mucho más que un diagrama de clases UML
- Son más abstractos que los marcos de trabajo
- Capturan las abstracciones del espacio del problema que son relevantes para encontrar una solución
- Se ha de utilizar una “metodología de diseño orientada a objetos” para poder obtener una descomposición del sistema

Clasificación de los patrones



- Patrones de Diseño Orientado a Objetos: descomposición, intercomunicación y ubicación de los componentes de un sistema dentro de una jerarquía de clases
- Patrones arquitectónicos: intentan reflejar la estructura de los problemas que resuelven
- Patrones de datos: relacionados con la organización de la información en aplicaciones

Clasificación de los patrones de diseño

- ❶ *Creacionales*: instanciación de objetos reforzando las restricciones en el tipo y número
- ❷ *Estructurales*: organización en la integración de clases de objetos
- ❸ *Comportamentales*: asignación de responsabilidades y comunicación entre los objetos

Ámbito de aplicación de los patrones

- Clases y subclasses \Rightarrow relaciones estáticas
- Objetos \Rightarrow relaciones dinámicas
- Diferencias conceptuales entre los tipos de patrones de diseño y su ámbito (sobre clases vs. objetos):
 - patrones de creación
 - patrones de comportamiento

Patrones aplicados a objetos

- Ayudan a determinar qué es un objeto y hasta donde llegamos en la descripción de su *granularidad*
- Determinación de la jerarquía y tipos de clases de objetos
- Determinación de la *interfaces*
- Diferencias entre el mecanismo de herencia de las clases y de las interfaces

Definition (Principio de diseño)

“Program to an interface not to an implementation”

Tipos de metodologías de diseño orientado a objetos

- Enfocadas hacia la descripción de jerarquías de clases
- Enfocadas hacia las colaboraciones y responsabilidades en el sistema objetivo
- Enfocada hacia el modelado
- Los enfoques anteriores son complementarios

Descripción de un patrón

Plantilla	
Contexto	describe el entorno en el que se ubica el problema incluyendo el dominio de aplicación
Problema	una o dos frases que explican lo que se pretende resolver
Fuerzas	lista el <i>sistema de fuerzas</i> que afectan a la manera en que ha de resolverse el problema incluye las limitaciones y restricciones que han de respetarse
Solución	proporciona una descripción detallada de la solución propuesta para el problema
Intención	describe el patrón y lo que hace
Anti-patrones	"soluciones" que no funcionan en el contexto o que son peores; suelen ser errores cometidos por principiantes
Patrones relacionados	referencias cruzadas relacionadas con los patrones de diseño
Referencias	reconocimientos a aquellos desarrolladores que desarrollaron o inspiraron el patrón que se propone

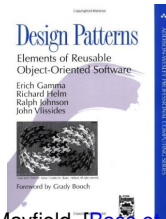
Lenguajes de patrones

Modelo de Requisitos
Describe el conjunto del problema. Establece el contexto e identifica los <i>intereses</i> que dominan en la búsqueda de un patrón.

- Se trata de una colección de patrones de diseño que colaboran para resolver los problemas dentro de un dominio de aplicaciones
- Se ha de definir una *arquitectura software* o un *marco de trabajo*
- Hasta llegar al nivel de *patrones de diseño* que implican clases y componentes relacionados

Catálogos de Patrones de Diseño

- Existen varios catálogos actualmente:
 - El denominado “The Gang of Four”(GoF):
Gamma–Helm–Johnson–Vlissides [[Gamma et al., 2009](#)]



- Pero no es el único: Coad-North-Mayfield, [[Bass et al., 2012](#)],
“Handbook of Software Architecture” [[Booch, 2008](#)],
Buschmann [[Buschmann and et al., 2004](#)],
Lethbridge [[Lethbridge and Laganieri, 2005](#)], etc.
- GoF y “Code Complete”(Steve McConnell, Microsoft) son
los catálogos de patrones de diseño de referencia

Diseño basado en patrones dentro de un contexto

- Principios generales de diseño
 - ❶ Descubrir el contexto del problema a partir del modelo de requerimientos
 - ❷ Extraer los patrones del nivel actual de abstracción
 - ❸ Escoger un esqueleto de implementación que refleje el contexto
 - ❹ Trabajar siempre *hacia dentro* del contexto
 - ❺ Refinar el prototipo adaptándolo a las características del software

Tareas de diseño [Bruegge and Dutoit, 2004]

- Desarrollar una *jerarquía* de entidades de análisis del problema
- Determinar existencia de un *lenguaje de patrones* aplicable
- Determinación de *patrones arquitectónicos* candidatos
- Utilizar las colaboraciones entre patrones de más bajo nivel
- Determinación de *patrones de diseño* candidatos
- Buscar los patrones de diseño de interfaces adecuados
- Independientemente del nivel de abstracción, compararlo con otras soluciones pre-existentes

Abstracción—caso

“Abstraction—occurrence”, se encuentra en diagramas de clases que forman parte de un modelo de dominio del sistema. Se aplica a conjuntos de objetos relacionados: “casos”

- *Contexto*
- *Problema*
- *Fuerzas*
- *Solución* item *Ejemplos*
- *Anti—patrones*
- *Patrones relacionados*
- *Referencias*: generalización del patrón Title—Item publicado inicialmente en [[Eriksson and Penker, 2000](#)].

Abstracción–caso–2

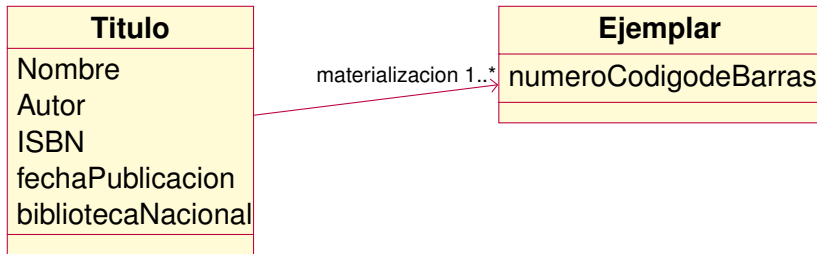
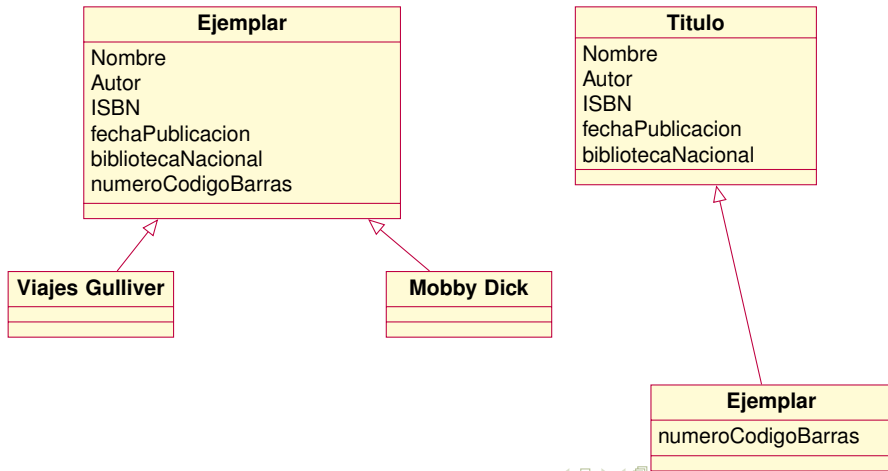


Figura: Plantilla–ejemplo del patrón *Abstracción–caso*

Abstracción—caso-3



Patrones relacionados

- El catálogo GoF propone los patrones: *Flyweight* y *Prototype*, que pueden considerarse relacionados con *Abstracción–caso*
- *Flyweight* utiliza la *compartición* del estado intrínseco de objetos que poseen una granularidad fina para evitar la creación masiva de estos objetos en una determinada aplicación.
- *Prototype*: especifica la tipología de los objetos a crear utilizando una clase–plantilla (*prototipo*) de estos, de tal forma que los nuevos objetos se crean copiando el prototipo.

Flyweight

- Un gran número de objetos de baja granularidad
- Almacenarlos todos sin compartir estado: mucha replicación de datos
- Posible encapsulación del *estado extrínseco*

- Contexto
- Problema
- Fuerzas
- Solución
- Ejemplos
- Anti-patrones
- Patrones relacionados
- Referencias: inicialmente propuesto en GoF [[Gamma et al., 2009](#)]

Flyweight-2

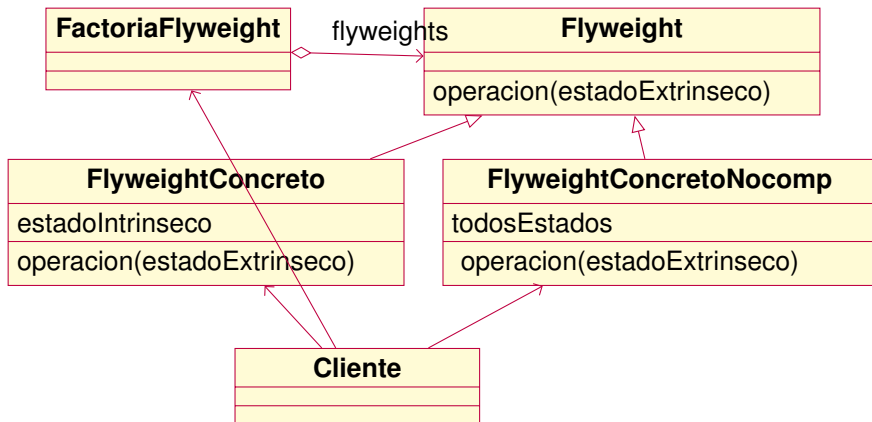


Figura: Diagrama de clases del patrón Flyweight

Prototype

Cuando resulta más práctico *clonar* los objetos que instanciar una clase determinada para crear objetos en determinadas aplicaciones.

- Contexto
- Problema
- Fuerzas
- Solución
- Ejemplos
- Anti-patrones
- Patrones relacionados
- Referencias: inicialmente propuesto en GoF [[Gamma et al., 2009](#)]

Prototype-2

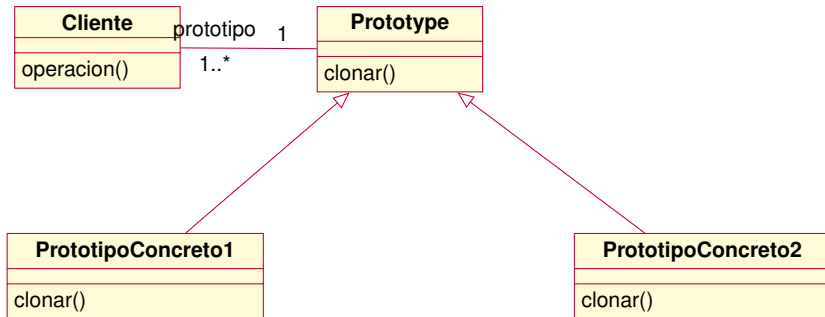


Figura: Diagrama de clases del patrón Prototype

Singleton

La idea de este patrón viene de la necesidad de los sistemas software de encontrar clases para las cuales sólo se necesita crear una instancia. A este tipo de clases se les denomina *singleton*.

- Contexto
- Problema
- Fuerzas
- Solución
- Ejemplos
- Anti-patrones
- Patrones relacionados
- Referencias: inicialmente propuesto en Lethbridge [[Lethbridge and Laganriere, 2005](#)]

Singleton-2

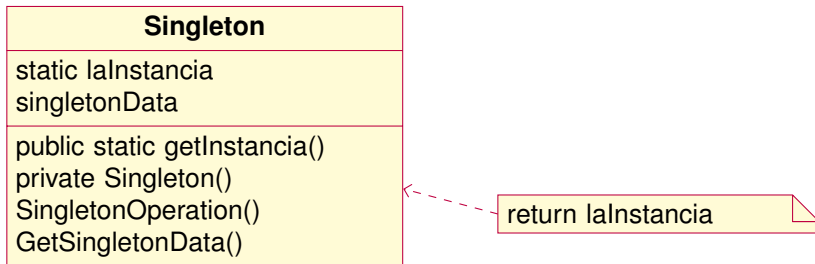


Figura: Diagrama de clases del patrón Singleton

Factoría

Necesitamos crear objetos dentro de un marco de trabajo pero desconocemos la clase de estos objetos ya que depende de la aplicación concreta. Proporcionar interfaces para crear familias de objetos sin especificar las clases a las que pertenecerán.

- Contexto
- Problema
- Fuerzas
- Solución
- Ejemplos
- Anti-patrones
- Patrones relacionados
- Referencias: inicialmente propuesto en GoF [[Gamma et al., 2009](#)]

factoria-2

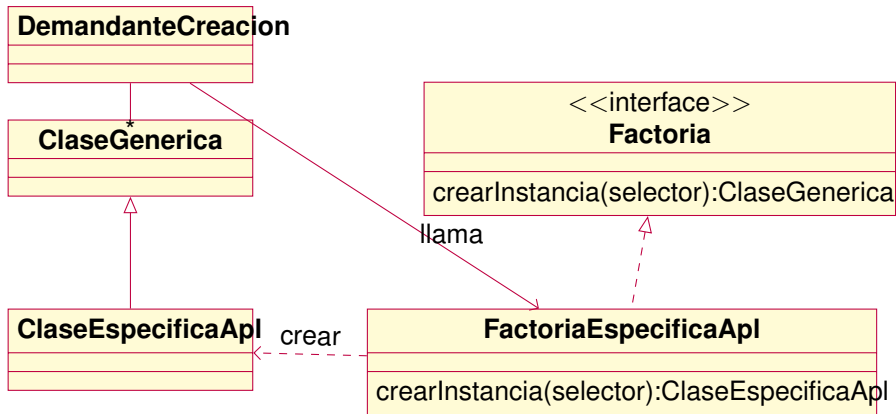


Figura: Diagrama de clases del patrón Factoria

Método Factoría

Cuando el método para crear los objetos está incluido en una clase abstracta pero el conocimiento para crearlos está fuera del marco actual de trabajo.

- Contexto
- Problema
- Fuerzas
- Solución
- Ejemplos
- Anti-patrones
- Patrones relacionados
- Referencias: inicialmente propuesto en GoF [[Gamma et al., 2009](#)]

Método Factoría-2

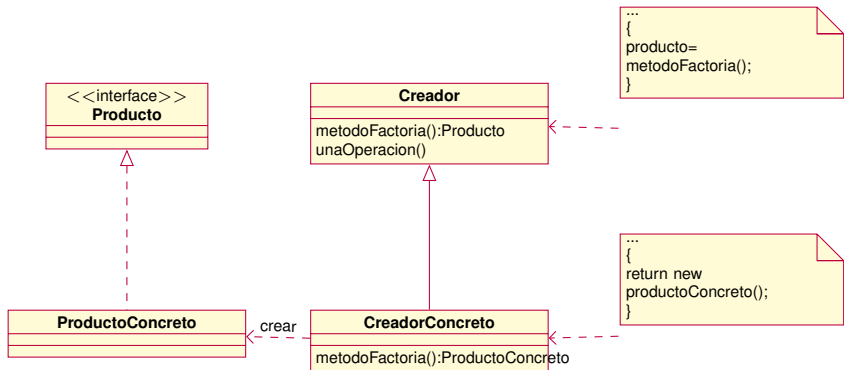


Figura: Diagrama de clases del patrón MetodoFactoria

Método Factoría vs Factoría Abstracta

El método factoría puede ser redefinido en una subclase.

```
Class A{
    public void hacerAlgo(){
        Foo f= hacerFoo();//metodo factoria
        f.loQueSea();
    }
    protected Foo hacerFoo(){
        return new FooNormal();
    }
}
Class extends A{
    protected Foo hacerFoo(){//Redefine el metodo-
        return new FooSuper();//factoria y devuelve algo
        diferente
    }
}
```

Método Factoría vs Factoría Abstracta-2

Una clase delega la creación del objeto a otra clase a través de *delegación*

```
Class A{
    private Factoria factoria;
    public A(Factoria factoria){
        this.factoria= factoria;
    }
    public void hacerAlgo(){
        Foo f= factoria.hacerFoo();//metodo factoria
        f.loQueSea();
    }
}

Interface Factoria{
    Foo hacerFoo();
    Barra hacerBarra();
}
```

Bridge

El objetivo de este patrón es conseguir desacoplar una abstracción de su implementación, de tal manera que ambas puedan variar independientemente.

- Contexto
- Problema
- Fuerzas
- Solución
- Ejemplos
- Anti-patrones
- Patrones relacionados
- Referencias: inicialmente propuesto en GoF [[Gamma et al., 2009](#)]

Bridge-2

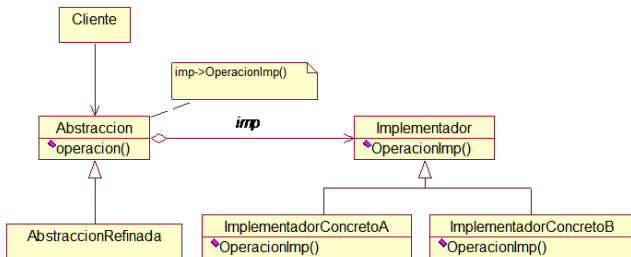


Figura: Diagrama de clases del patrón Bridge

Jerarquía General

Cuando se necesita representar una jerarquía de objetos en la que algunos de estos pueden tener subordinados y otros no.

- Contexto
- Problema
- Fuerzas
- Solución
- Ejemplos
- Anti-patrones
- Patrones relacionados
- Referencias: inicialmente propuesto en GoF [[Gamma et al., 2009](#)]

Jerarquía General-2

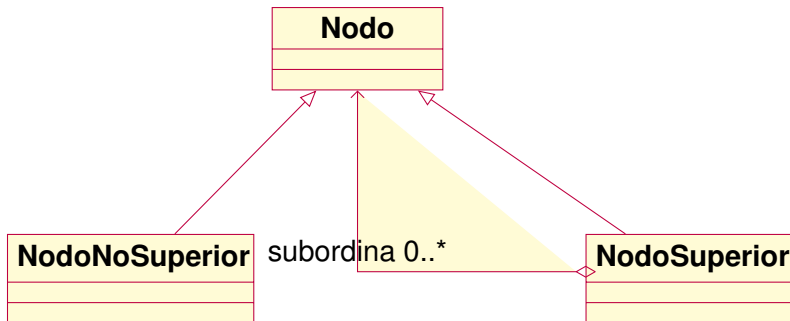


Figura: Diagrama de clases del patrón Jerarquía General

Composite

Mediante este patrón un grupo de objetos se puede manipular de la misma forma que un solo objeto. Se entiende como una *clase abstracta* que representa tanto a entidades como a sus contenedores.

- Contexto
- Problema
- Fuerzas
- Solución
- Ejemplos
- Anti-patrones
- Patrones relacionados
- Referencias: inicialmente propuesto en GoF [[Gamma et al., 2009](#)]

Composite–2

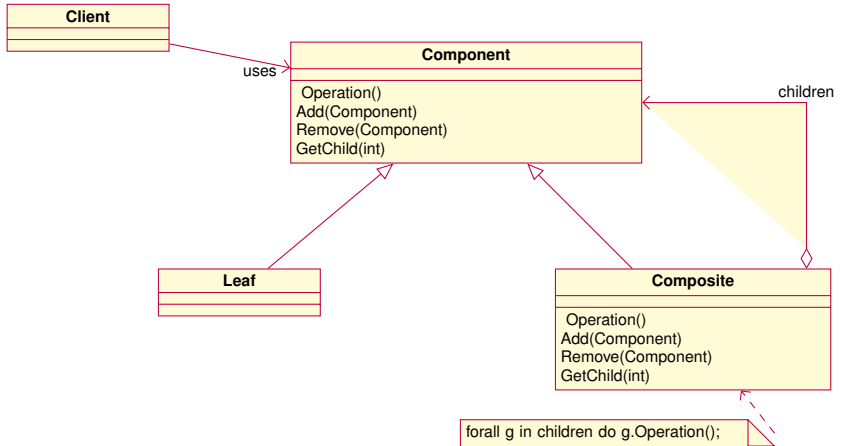


Figura: Diagrama de clases del patrón Composite

Proxy

Es de utilidad para reducir la carga en memoria de objetos pesados cuando la aplicación no los necesita inmediatamente.

- Contexto
- Problema
- Fuerzas
- Solución
- Ejemplos
- Anti-patrones
- Patrones relacionados
- Referencias: inicialmente propuesto en GoF [[Gamma et al., 2009](#)]

Proxy-2

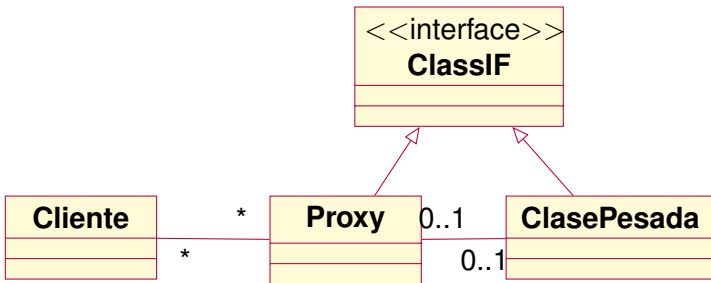


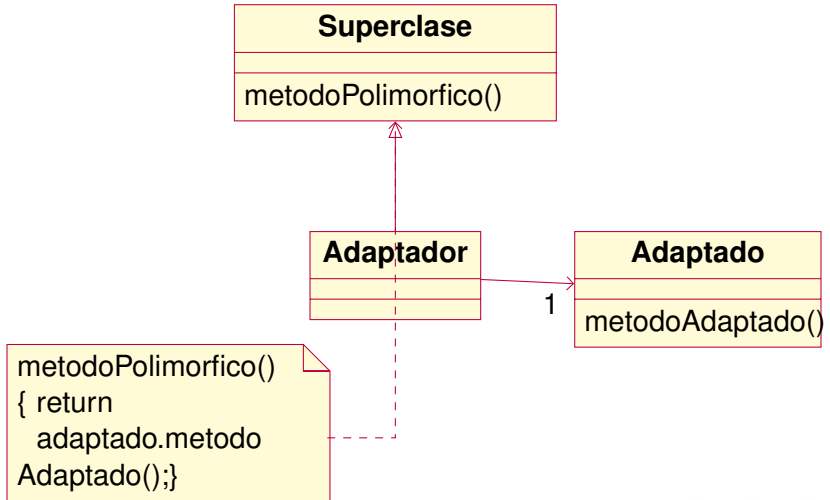
Figura: Diagrama de clases del patrón Proxy

Adaptador

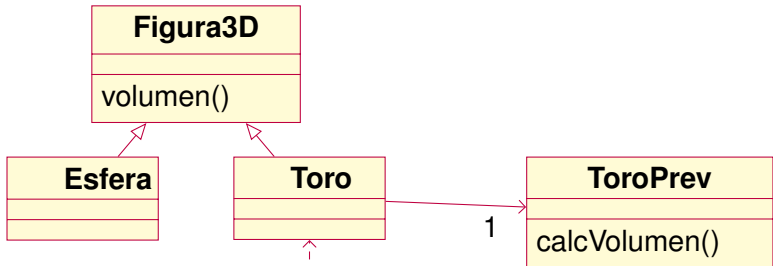
Para reutilizar una clase pre-existente en una jerarquía de clases, con la que no guarda ninguna relación, dentro de la aplicación que estamos desarrollando.

- Contexto
- Problema
- Fuerzas
- Solución
- Ejemplos
- Anti-patrones
- Patrones relacionados
- Referencias: inicialmente propuesto en GoF [[Gamma et al., 2009](#)]

Adaptador-2



Adaptador-3



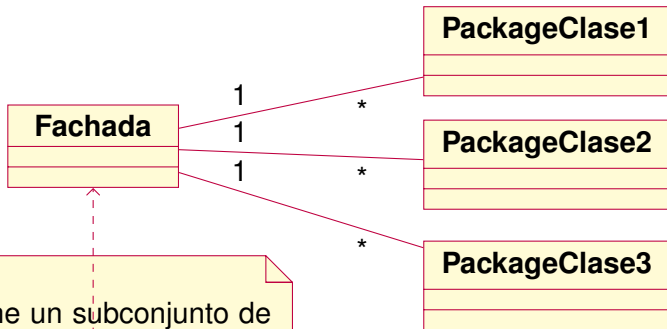
```
Toro
volumen(){ return
    toroPrev.calcVolumen();}
```

Fachada

Se utiliza para simplificar la utilización de un paquete de clases complejo *“fabricando una interfaz”* más fácil de utilizar.

- Contexto
- Problema
- Fuerzas
- Solución
- Ejemplos
- Anti-patrones
- Patrones relacionados
- Referencias: inicialmente propuesto en GoF [[Gamma et al., 2009](#)]

Fachada-2



Nota

Contiene un subconjunto de métodos públicos tal que los otros subsistemas no tendrán que acceder a las clases dentro de los paquetes.

Extension Interface

Para definir componentes evolutivos, que sufren modificaciones tanto en su implementación como en su interfaz.

- Contexto
- Problema
- Fuerzas
- Solución
- Ejemplos
- Anti-patrones
- Patrones relacionados
- Referencias: inicialmente propuesto en GoF [[Gamma et al., 2009](#)]

Extension Interface—2

Los clientes se diseñan para que puedan a los componentes software a través de interfaces separadas, una por cada rol que pueda jugar el componente.

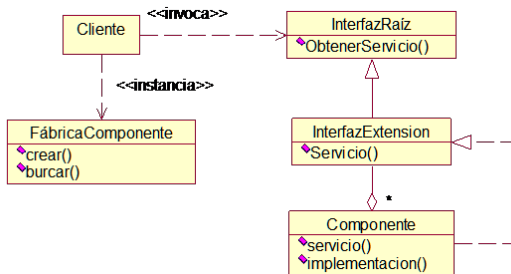


Figura: Diagrama de clases del patrón Extension Interface

Inmutable

Cuando se necesitan objetos cuyo estado no puede cambiar después de ser creados.

```
public final class User {  
    private final String nombreusuario;  
    private final String clave;  
    public User(String nombreusuario, String clave) {  
        this.nombreusuario = nombreusuario;  
        this.clave = clave;  
    }  
    public String getnombreusuario() {  
        return nombreusuario;  
    }  
    public String getclave() {  
        return clave;  
    }  
}
```

Immutable–2

El patrón `Builder` (GoF [[Gamma et al., 2009](#)]) puede utilizarse para construir el patrón *Immutable*.

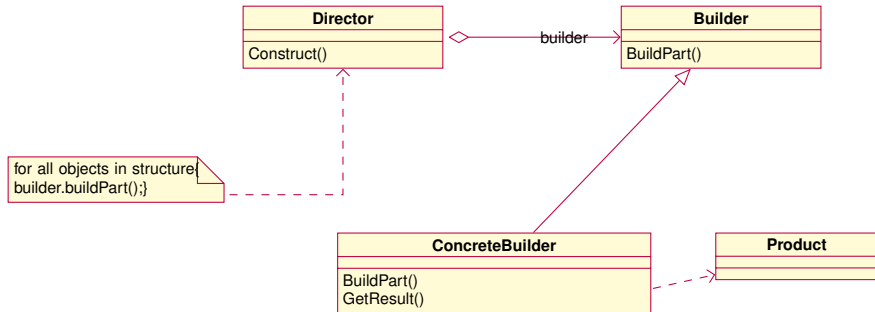


Figura: Diagrama de clases del patrón `Builder`

Immutable—3

Programación que utiliza una clase *Immutable* para garantizar que no se cambian los datos de un usuario, una vez creado.

```
public class UsuarioImmutable {  
    private final String nombreusuario;  
    private final String clave;  
    private final String nombre;  
    private final String apellidos;  
    private final String email;  
    private final Date fechaCreacion;  
    private UsuarioImmutable(BuilderUsuario builder) {  
        this.nombreusuario = builder.nombreusuario;  
        this.clave = builder.clave;  
        this.fechaCreacion = builder.fechaCreacion;  
        this.nombre = builder.nombre;  
        this.apellidos = builder.apellidos;  
        this.email = builder.email;  
    }  
    ...  
}
```

Inmutable-4

```
//... continua la clase Inmutable
public static class BuilderUsuario {
    private final String nombreusuario;
    private final String clave;
    private final Date fechaCreacion;
    private String nombre;
    private String apellidos;
    private String email;
    public BuilderUsuario(String nombreusuario, String
        clave) {
        this.nombreusuario = nombreusuario;
        this.clave = clave;
        this.fechaCreacion = new Date();
    }
    public BuilderUsuario nombre(String firstname) {
        this.nombre = firstname;
        return this;
    }
}
```

Immutable-5

```
//... continua la clase Immutable
    public BuilderUsuario apellidos(String apellidos) {
        ...
    }
    public BuilderUsuario email(String email) {
        ...
    }
    public UsuarioImmutable build() {
        return new UsuarioImmutable(this);
    }
} //acaba la clase interna BuilderUsuario
public String getnombreusuario() {
    return nombreusuario;
}
//getclave(), getnombre(), getapellidos(), getemail(),
//getfechaCreacion()
} //acaba la clase UsuarioImmutable
```

Inmutable—6

Programación del cliente de la clase `UsuarioImmutable`

```
public static void main(String[] args) {  
    UsuarioImmutable user = new UsuarioImmutable.  
        BuilderUsuario("manuel", "DS2016In").  
        nombre("manuel").apellidos("capel").email("  
            manuel@gmail.com").build();  
}
```

- Patrones relacionados: `SoloLectura`
- Referencias: aparece inicialmente en el libro de [\[Grand, 1999\]](#)

Interfaz SoloLectura

Para conseguir que clases *sin privilegio* puedan modificar el estado de los objetos de una clase. La clase `SoloLectura` sería inmutable para las clases sin privilegio.

- Contexto
- Problema
- Fuerzas
- Solución
- Ejemplos
- Anti-patrones
- Patrones relacionados
- Referencias: inicialmente propuesto en el libro [[Grand, 1999](#)]

SoloLectura-2

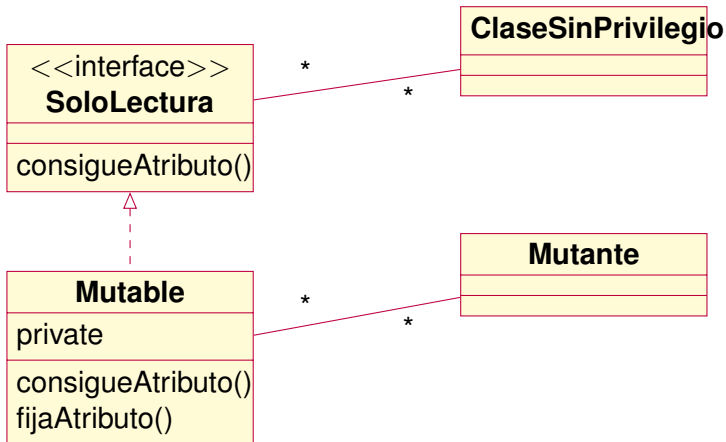


Figura: Diagrama de clases del patrón SoloLectura

Strategy

Permite intercambiar algoritmos después de hacerlos independientes (mediante “encapsulación”) de las aplicaciones que los utilizarán. Los algoritmos encapsulados puede modificarse independientemente de sus clientes. Cada algoritmo encapsulado en una clase se le llama *estrategia*

- Contexto
- Problema
- Fuerzas
- Solución
- Ejemplos
- Anti-patrones
- Patrones relacionados
- Referencias: inicialmente propuesto en GoF [[Gamma et al., 2009](#)]

Strategy-2

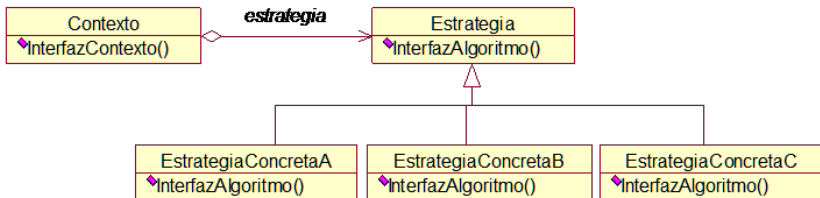


Figura: Diagrama de clases del patrón Strategy

Actor–papel

Cuando determinados objetos pueden adoptar comportamientos/funcionalidad diferentes durante la ejecución de una aplicación y no queremos utilizar herencia múltiple.

- Contexto
- Problema
- Fuerzas
- Solución
- Ejemplos
- Anti–patrones
- Patrones relacionados
- Referencias: inicialmente propuesto en el libro [[J. Rumbaugh, 2004](#)]

Actor-papel-2

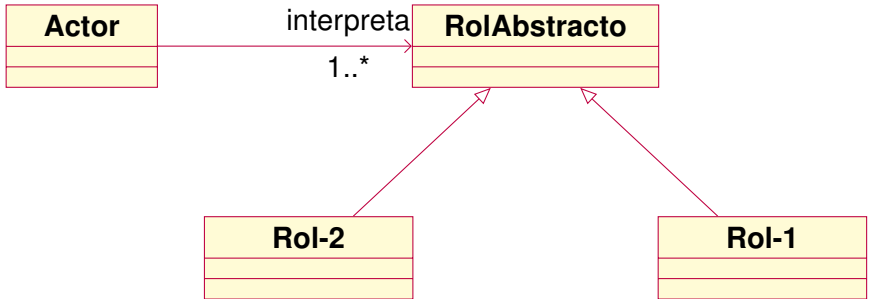


Figura: Diagrama de clases del patrón Actor-papel

Delegacion

Necesitamos utilizar una operación programada como un método de una clase externa pero no queremos heredar de esa clase.

- Contexto
- Problema
- Fuerzas
- Solución
- Ejemplos
- Anti-patrones
- Patrones relacionados
- Referencias: inicialmente propuesto en [[Lethbridge and Laganier, 2005](#)]

Delegacion-2

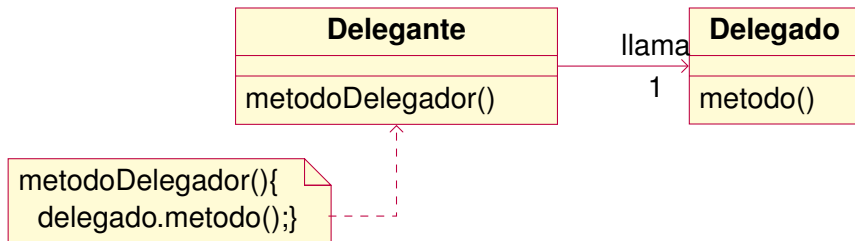


Figura: Diagrama de clases del patrón Delegacion

Delegacion-3

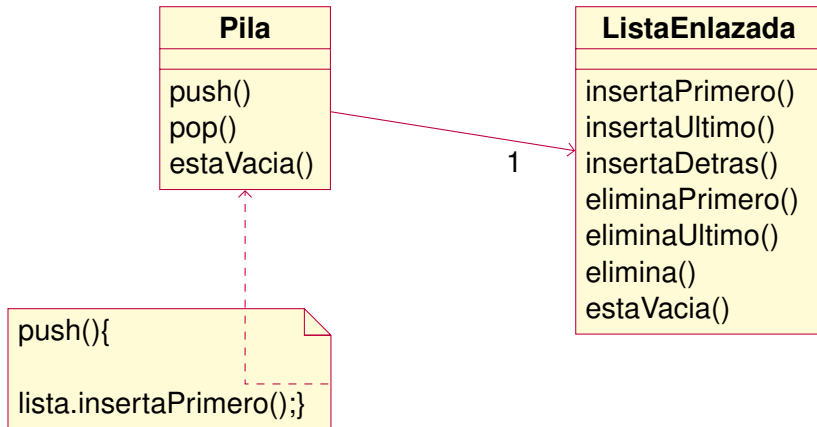


Figura: Ejemplo de uso del patrón de diseño Delegación

Command

Las solicitudes de los clientes se convierten en objetos, permitiendo generalizar a los clientes dependiendo sólo de su tipo de solicitud, encolar o registrar solicitudes y convertir en reversibles las operaciones.

- Contexto
- Problema
- Fuerzas
- Solución
- Ejemplos
- Anti-patrones
- Patrones relacionados
- Referencias: inicialmente propuesto en GoF [[Gamma et al., 2009](#)]

Command-2

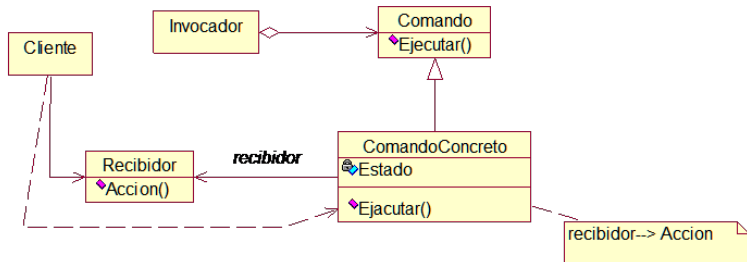


Figura: Diagrama de clases del patrón Command

Observable—Observador

Reducir la interdependencia entre clases cuando una de ellas modifica información y las otras se suscriben a dicha actualización.

- Contexto
- Problema
- Fuerzas
- Solución
- Ejemplos
- Anti–patrones
- Patrones relacionados
- Referencias: en GoF [[Gamma et al., 2009](#)] se le denomina: *Publicar y Suscribir*

Observable–Observador–2

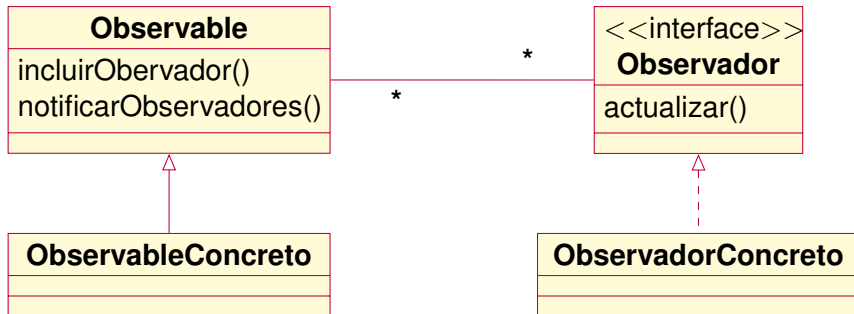


Figura: Diagrama de clases del patrón Observable–observador

Observable–Observador–3

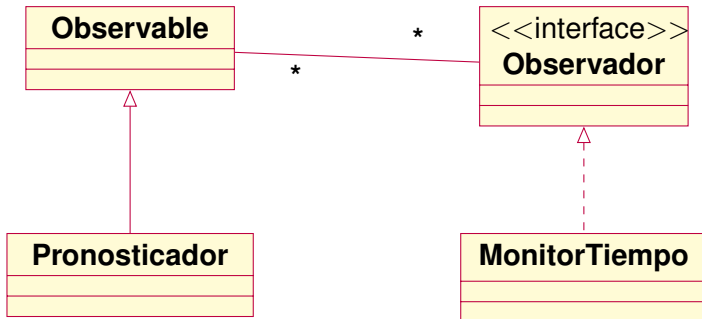


Figura: Ejemplo del patrón Observable–observador

Visitante

Para permitir a las aplicaciones realizar operaciones cuya ejecución necesitará recorrer una estructura jerárquica de datos sin modificar dichos datos o alterando la estructura o las relaciones de herencia de sus clases.

- Contexto
- Problema
- Fuerzas
- Solución
- Ejemplos
- Anti-patrones
- Patrones relacionados
- Referencias: inicialmente propuesto en GoF [[Gamma et al., 2009](#)]

Visitante-2

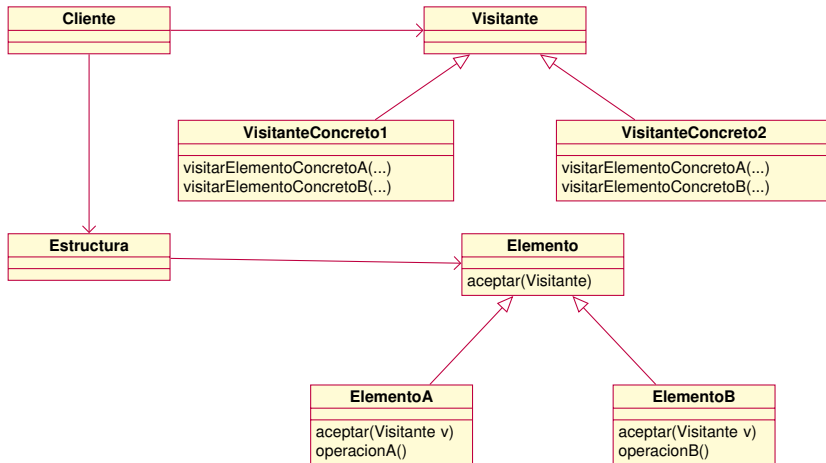


Figura: Diagrama de clases del patrón de diseño Visitante

Visitante—3

```
Cliente::{  
  
    VisitantePrecio vep= new VisitantePrecio();  
    Equipo t= new Tarjeta(); //elementos concretos...  
    Equipo b= new Bus();  
    Equipo d= new Disco();  
    ...  
    t.aceptar(vep); //visita al objeto tarjeta  
    b.aceptar(vep); //visita al objeto bus  
    d.aceptar(vep); //visita al objeto disco  
}
```

Componentización y calidad del software

- Objetivo fundamental: conseguir componentes y servicios software “evolutivos”.

Definition (Software evolutivo)

en la medida que incorpore, de manera estable, los cambios en los requerimientos que puedan aparecer durante el proceso de desarrollo y de la vida del sistema software.

- Para producir “software evolutivo” una arquitectura ha de propiciar la facilidad de modificación, flexibilidad y extensibilidad del software.
- Los *atributos de calidad* anteriores dependerán de la calidad de la arquitectura software utilizada para desarrollar el sistema.
- Los patrones producen arquitecturas software de calidad.

Patrones y mantenibilidad del software

- Se estudian los patrones de diseño que propician conseguir el atributo de calidad denominado “*mantenibilidad*”(facilidad de mantenimiento) del software.
- Referencia ISO 9126 (ISO/IEC 2002)

Definition (Mantenibilidad del software)

Facilidad de cambio, facilidad de pruebas, facilidad de análisis y estabilidad del software.

- Esta característica de calidad no se limita sólo a la fase de *Mantenimiento*: está relacionado con el concepto de *evolución del software*.

Fachada

Definition (Contexto)

- Mejorar la portabilidad de una aplicación al encapsular las APIs de bajo nivel del sistema operativo.
- Encapsular mecanismos o servicios proporcionados por APIs existentes, no orientadas a objeto.
- Aumentar la cohesión de componentes.

Fachada-2

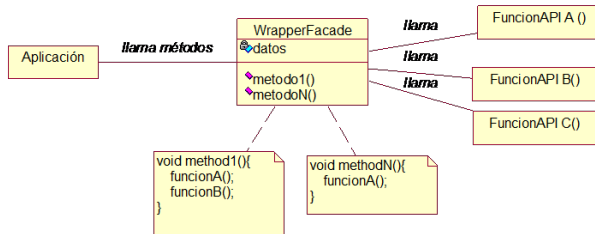


Figura: Diagrama de clases de Wrapper-Facade

Implicaciones en la calidad

Beneficios	Atributo de calidad	Características ISO 9126
Encapsular APIs de bajo nivel	Encapsulamiento, cohesión	Mantenibilidad, cambios
Decrementar probabilidad errores	Integridad	Facilidad, madurez
Acceso uniforme a APIs	Accesibilidad	Funcionalidad, adecuación
Aislar de efectos no portables	adaptabilidad	Portabilidad, adaptabilidad
Mejora de la configurabilidad	Independencia de la plataforma	Adaptabilidad
Mejora de la comprensión	Modificabilidad	Cambios y pruebas
Clases más cohesivas	Reusabilidad	Cambios, análisis
Manejo excepciones	Integridad	Fiabilidad, tolerancia fallos

Implicaciones en la calidad–2

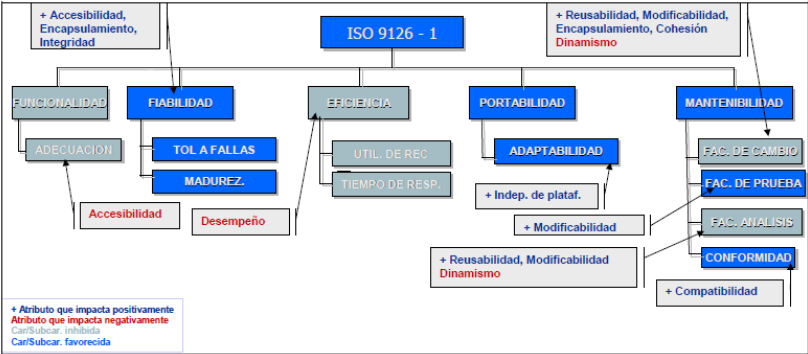
Perjuicios	Atributo de calidad	Características ISO 9126
Perder funcionalidad	Accesibilidad	Funcionalidad, adecuación
Degradar el rendimiento	Desempeño	Eficiencia, tiempo respuesta
Limitaciones de integración	Compatibilidad	Mantenibilidad
Estaticidad del patrón	Dinamismo	Facilidades

- Implementar una abstracción sobre otra pre-existente puede ocasionar que se pierda funcionalidad
- Incrementar el coste en recursos (salvar contextos) si los métodos de la nueva API incluyen mucha funcionalidad
- Problemas de integración de *interfaces nativas* en los lenguajes de programación que implementen la nueva API
- Limitación de la reusabilidad de las clases del patrón, una vez implementado

Resumen características de calidad

Característica	Sub-característica	Impacto	Atributo
Mantenibilidad	Facilidad de cambio	+	Reusabilidad Modificabilidad Encapsulamiento Cohesión
		-	Dinamismo
	Facilidad de análisis	+	Reusabilidad Modificabilidad
		-	Dinamismo
	Facilidad de prueba	+	Modificabilidad
	Conformidad con estándares	-	Compatibilidad
Funcionalidad	Adecuación	-	Accesibilidad
Fiabilidad	Madurez	+	Accesibilidad Encapsulamiento Integridad
	Tolerancia a Fallas		
Eficiencia	Tiempo de respuesta	-	Desempeño
	Uso de recursos		
Portabilidad	Adaptabilidad	+	Independencia de plataforma

Resumen características de calidad-2

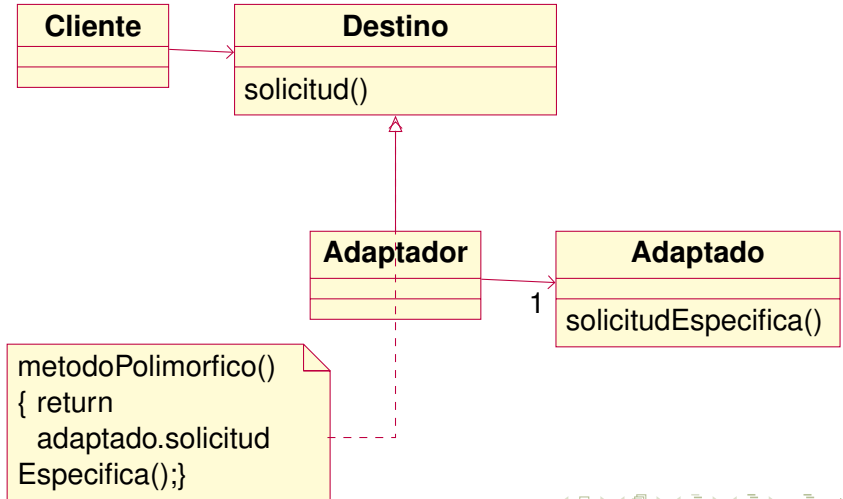


Adaptador

Definition (Contexto)

- Reutilizar clases existentes cuya interfaz no es adecuada para lo que se necesita en la aplicación cliente
- Crear clases reutilizables que cooperen con otras clases
- Utilizar varias subclases de manera conjunta

Adaptador-2



Implicaciones en la calidad

Beneficios	Atributo de calidad	Características ISO 9126
Coordinación mínima con adaptado	Desempeño	Eficiencia, tiempo respuesta
Funcionalidad inclusiva	Extensibilidad	Mantenibilidad, facilita cambios
Múltiples adaptadores	Reusabilidad	Mantenibilidad, cambios, análisis

Implicaciones en la calidad–2

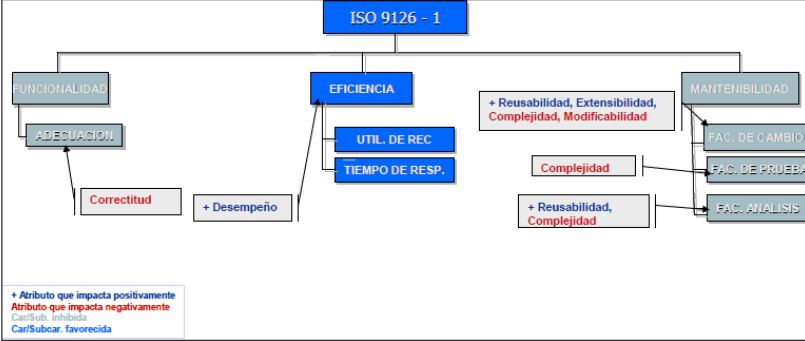
Perjuicios	Atributo de calidad	Características ISO 9126
No transparencia del adaptador	Complejidad	Manten., cambios, pruebas, análisis
Mala adaptación de subclases	Modificabilidad	Mantenibilidad, cambios
Invaldar comportamiento adaptado	Correctitud	Funcionalidad, adecuación
Trabajo adaptación variable	Complejidad	Mantenibilidad, cambios

- Los adaptadores no son transparentes a todos los clientes
- Una clase adaptadora se limita a la adaptación de una clase–origen en clases–destino
- La cantidad de trabajo a realizar dependerá del grado de similitud con el adaptado

Resumen características de calidad

Característica	Sub-característica	Impacto	Atributo
Mantenibilidad	Facilidad de cambio	+	Extensibilidad Reusabilidad
		-	Complejidad Modificabilidad
	Facilidad de prueba	-	Complejidad
	Facilidad de análisis	+	Reusabilidad
		-	Complejidad
Eficiencia	Tiempo de respuesta	+	Desempeño
Funcionalidad	Adecuación	-	Correctitud

Resumen características de calidad-2

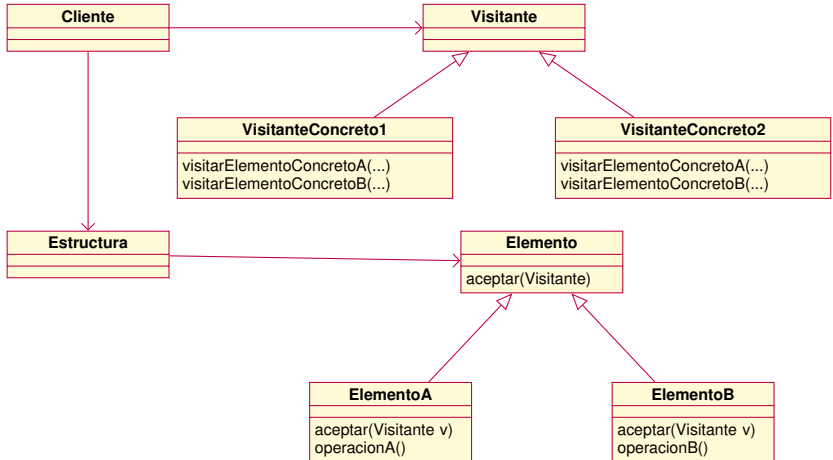


Visitante

Definition (Contexto)

- Estructuras de objetos de muchas clases y con interfaces diferentes
- Ejecutar operaciones distintas y no relacionadas sobre los objetos de la estructura
- Definir nuevas operaciones sobre los objetos

Visitante-2



Implicaciones en la calidad

Beneficios	Atributo de calidad	Características ISO 9126
Fácil adición de operaciones	Modif.,extens.	Mantenibilidad,cámbios,análisis
Comportameinto centralizado	Cohesión	Mantenibilidad,cambios,pruebas
Visitar objetos independientes	Flexibilidad	Mantenibilidad,cambios
Extensibilidad interfaces visitantes	Flexibilidad	Mantenibilidad,cambios

Implicaciones en la calidad–2

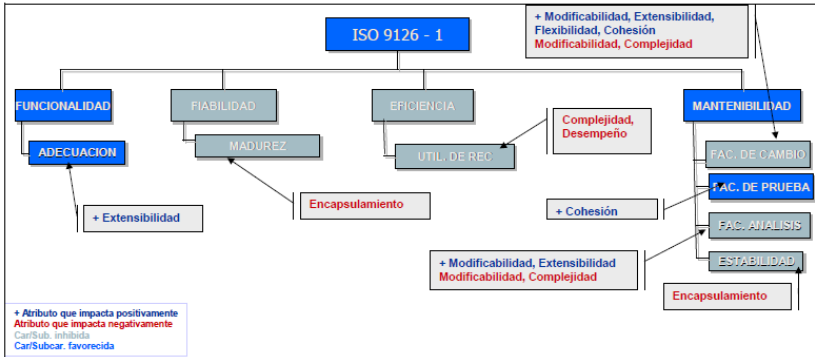
Perjuicios	Atributo de calidad	Características ISO 9126
Deterioro jerarquía visitantes	Cohesión	Manten.,cambios,pruebas
Acumulación estados visitados	Modif.,complj.	Manten.,cambios,pruebas
Compromiso encapsulamiento objetos	Complej.,desempeño	Eficiencia,recursos

- Cuando se añaden con frecuencia nuevas clases de elementos concretos, se puede afectar al Visitante
- Acumulación de distintos estados en los visitantes como consecuencia de visitar a los objetos por diferentes motivos
- Acceder al estado interno de los objetos visitados puede romper su encapsulamiento

Resumen características de calidad

Característica	Sub-característica	Impacto	Atributo
Mantenibilidad	Facilidad de cambio	+	Modificabilidad Extensibilidad Flexibilidad Cohesión
		-	Modificabilidad Complejidad
	Facilidad de análisis	+	Modificabilidad Extensibilidad
		-	Modificabilidad Complejidad
	Estabilidad	-	Encapsulamiento
	Facilidad de prueba	+	Cohesión
Fiabilidad	Madurez	-	Encapsulamiento
Eficiencia	Uso de recursos	-	Complejidad Desempeño
Funcionalidad	Adecuación	+	Extensibilidad

Resumen características de calidad-2

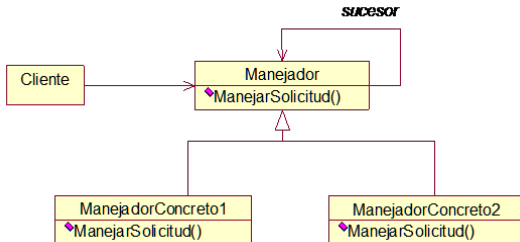


Chain of responsibility

Definition (Contexto)

- Objetos “manejadores” son asignados dinámicamente para que atiendan peticiones de aplicaciones–cliente
- Delegar el atender las peticiones a uno o varios objetos sin que estos estén determinados de antemano
- Inclusión dinámica de objetos manejadores
- Relacionado con el patrón `Composite`

Chain of responsibility



Implicaciones en la calidad

Beneficios	Atributo de calidad	Características ISO 9126
Simplificar interconexiones de objetos	Acoplamiento	Manten.,cambios,análisis
Cambio dinámico responsabilidad	Modif.,flexibilidad	Manten.,cambios
Detección errores fácil	Toler.fallas	Fiabilidad
Mejor control peticiones	Manejo error,precisión	Fiabilidad
Escalabilidad de la cadena	Escalab.,modif.	Manten.,cambios

Implicaciones en la calidad–2

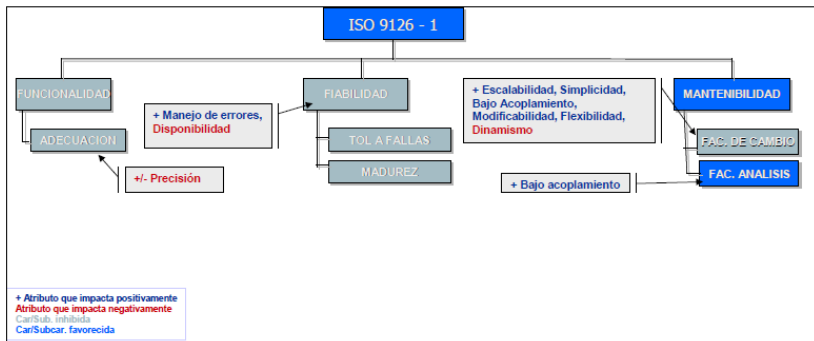
Perjuicios	Atributo de calidad	Características ISO 9126
Pérdida de peticiones	Adecuación,disponib.	Funcion.,fiab.,madurez
Funcionalidad por herencia	Dinamismo	Matenib.,cambios

- Una petición podría llegar al final de la cadena sin ser atendida
- Para que funcione bien la cadena ha de estar bien configurada
- El funcionamiento o el comportamiento necesitan de la herencia para poder ser manejados

Resumen características de calidad

Característica	Sub-característica	Impacto	Atributo
Mantenibilidad	Facilidad de cambio	+	Escalabilidad Simplicidad Bajo Acoplamiento Modificabilidad Flexibilidad
		-	Dinamismo
	Facilidad de análisis	+	Bajo acoplamiento
Fiabilidad	Madurez	-	Disponibilidad
	Tolerancia a fallas	+	Manejo de errores
Funcionalidad	Adecuación	+/-	Precisión

Resumen características de calidad-2



Patrón Arquitectónico

Clasificación de patrones (2008) [Booch, 2008]

Establece una clasificación atendiendo a características para cumplir con los requisitos de *seguridad*, *rendimiento*, *despliegue* y *almacenamiento*

- 1 *Control de accesos*: el acceso a determinadas partes de la arquitectura software ha de ser controlado rigurosamente.
- 2 *Concurrencia*: diferentes formas de permitir que los componentes de la aplicación sean concurrentes.
- 3 *Distribución*: la comunicación entre entidades software es muy diversa y afecta al diseño, la ubicación de los componentes ha de ser optimizable (*configurabilidad*)
- 4 *Persistencia*: la *supervivencia* de los objetos (atributos, estado) entre distintas ejecuciones es algo diseñable. Una mala solución puede dañar la eficiencia gravemente.

Patrones II

Para Buschmann son plantillas para arquitecturas de software concretas, que especifican las propiedades estructurales de una aplicación y tienen un impacto en la arquitectura de subsistemas.

El uso de ciertos mecanismos, como los estilos y patrones arquitectónicos, permite mejorar las características de calidad en el software [[Jansen and Bosch, 1999](#)], bien sean éstas observables o no en tiempo de ejecución [[Bass et al., 2012](#)].

- El ámbito del patrón es menos general que el del estilo arquitectónico
- Un patrón impone una regla a la arquitectura, describiendo cómo el software gestionará algún aspecto de su funcionalidad (p.e.: la concurrencia).
- Los patrones arquitectónicos tienden a abordar problemas de *comportamiento* del software específicos dentro del contexto de una arquitectura
- Los patrones y los estilos arquitectónicos se pueden utilizar de forma conjunta para dar forma a la estructura completa de un sistema.

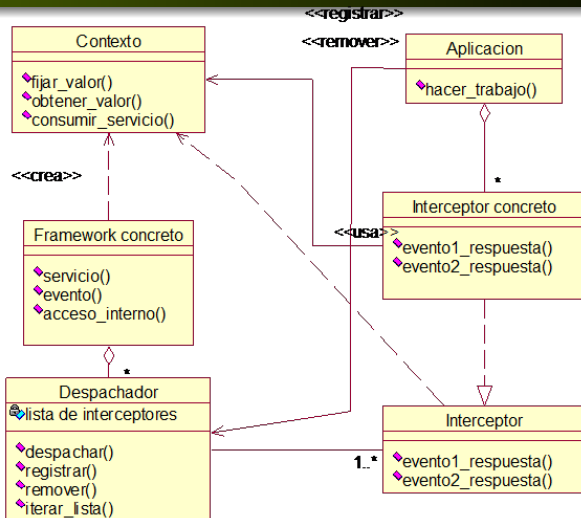
Diferencias según el nivel de abstracción



Patrón Interceptor

- Permite a determinados servicios ser añadidos de manera transparente a un marco de trabajo y ser disparados automáticamente cuando ocurren ciertos eventos
- Contexto: Desarrollo de marcos de trabajo que puedan ser extendidos de manera transparente
- Problema: Los marcos de trabajo, arquitecturas software, etc. ha de poder anticiparse a las demandas de servicios concretos que deben ofrecer a sus usuarios
- Integración dinámica de nuevos componentes sin afectar a la AS o a otros componentes
- Solución: Registro offline de servicios a través de una interfaz predefinida del marco de trabajo, posteriormente se disparan estos servicios cuando ocurran determinados eventos

Interceptor



Estructura de clases de “Interceptor”

- Un *FrameworkConcreto* que instancia una arquitectura genérica y extensible
- Los *Interceptores* que son asociados con un evento particular
- *InterceptoresConcretos* que especializan las interfaces del interceptor e implementan sus métodos de enlace
- *Despachadores* para configurar y disparar interceptores concretos
- Una *Aplicación* que se ejecuta por encima de un *framework concreto* y utiliza los servicios que éste le proporciona

Implicaciones en la calidad del patrón Interceptor

Beneficios	Atributo de calidad	Características ISO 9126
Cambiar/incluir servicios de un framework sin que sea preciso cambiarlo	Extensibilidad, Flex.,Dinamismo	Mantenibilidad Facilita cambios
Añadir interceptores sin afectar al código de la aplicación	Acoplamiento	Mantenibilidad Facilita cambios
Obtención dinámica inform. del framework con intercept. y objetos–contexto	Monitorización Control	Tolerancia a fallas Uso de recursos
Infraestructura de servicios estratificada con interceptores correspondientes simétricos	Encapsulamiento	Mantenibilidad Fac.cambios,análisis
Reutilización de interceptores en diferentes aplicaciones	Reusabilidad	Mantenibilidad Facilita cambios

Implicaciones en la calidad 2

Inconvenientes	Atributo de calidad	Características ISO 9126
Difícil ajuste del número de despachadores e interceptores	Complejidad Flexib., extensibilidad	Facilita cambios Facilita análisis
Bloqueo aplicación por fallo del interceptor	Disponibilidad Modificabilidad	Mantenibilidad Madurez, Tolerancia Fallas
Degradación rendimiento por cascadas de interceptores	Rendimiento Bloqueo	Eficiencia, madurez Tolerancia fallas

- Insuficientes interceptores y despachadores reduce la flexibilidad y extensibilidad del framework concreto.
- Sistema demasiado grande e ineficiente, complejo de aprender, implementar, usar, y optimizar, utilizando demasiados interceptores
- Para evitar el bloqueo completo de la aplicación pueden utilizarse estrategias de *time-out*, pero esto puede complicar el diseño del framework concreto
- Las cascadas de intercepción pueden conllevar una degradación del rendimiento o un bloqueo de la aplicación

Resumen características de calidad

Característica	Sub-característica	Impacto	Atributo
Mantenibilidad	Facilidad de cambio Facilidad de análisis	+	Reusabilidad Modificabilidad Encapsulamiento Extensibilidad Flexibilidad Acoplamiento Dinamismo
	Facilidad de cambio Facilidad de análisis	-	Extensibilidad Flexibilidad Complejidad Modificabilidad
Eficiencia	Tiempo de respuesta	-	Desempeño
	Uso de recursos	+	Monitoreo Control
Fiabilidad	Tolerancia a fallas	-	Disponibilidad Bloqueo
		+	Monitoreo Control
	Madurez	-	Disponibilidad Bloqueo

Resumen características de calidad

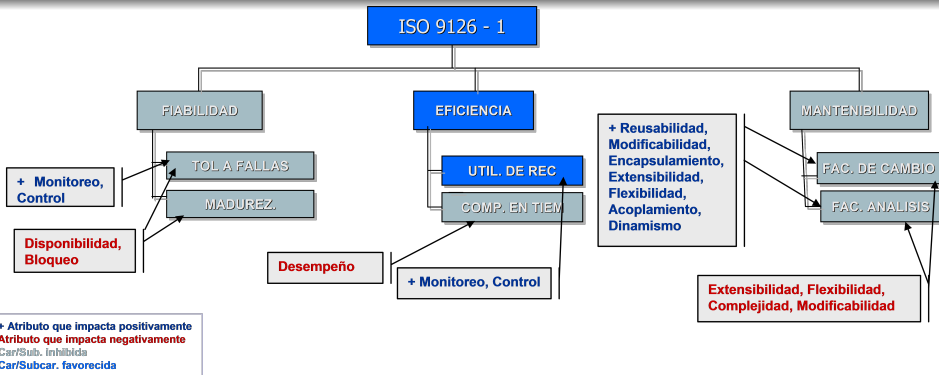
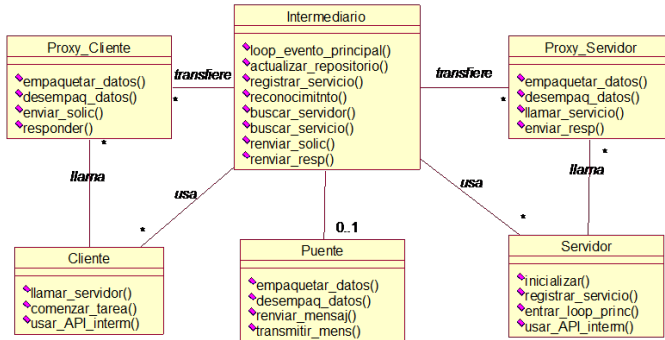


Figura: Características ISO del patrón "Interceptor"

Patrón Broker

- Para modelar sistemas distribuidos compuesto de componentes software totalmente desacoplados
- Contexto: cualquier sistema distribuido y, posiblemente, heterogéneo con componentes cooperando dentro de un sistema de información
- Problema: ¿Cómo estructurar componentes configurables dinámicamente e independientes de los mecanismos concretos de comunicación de un sistema distribuido?
- Solución: Introducir un componente *Broker* para mejor desacoplamiento entre clientes y servidores
- Las tareas de los Brokers incluyen la localización del servidor apropiado, envío de la petición al servidor y transmisión de los resultados

Broker



Estructura de clases del patrón Broker

- *Intermediario*: admite las solicitudes, asigna los servidores y responde a las peticiones de los clientes
- *Servidor*: se registra en el *Intermediario* e implementa el servicio
- *Cliente*: accede a los servicios remotos
- *Proxy Cliente* y *Proxy Servidor*, que proporcionan transparencia, ocultando los detalles de implementación del patrón
- *Puente*: le proporciona interoperabilidad al *Intermediario*

Ejemplo de uso del patrón Broker

Ejemplo demostrativo con un sistema E-Home que implementa una red de dispositivos domésticos colaborativos.

El `Broker` se encargará de facilitar la compatibilidad entre el software y los datos del cliente con los controladores de los distintos dispositivos incluidos en esta red.

Los servicios que están situados en un control central estarán conectados a través de una red con el software que se ejecuta en el controlador de los dispositivos.



Ejemplo de uso del patrón Broker-II

Estructura de clases

- `Cliente`: envía una petición de activación de un dispositivo concreto a `Broker`
- `Broker`: accede a `ControlCentral` de la red y envía al servidor los mensajes que ha recibido de `Cliente`
- `ControlCentral`: recibe mensajes de `Broker` y contacta con los distintos dispositivos para configurarlos y activarlos
- `ControladorDispositivo`: realiza acciones específicas sobre el dispositivo del que es responsable

Ejemplo de uso del patrón Broker-III

ControladorDispositivo, además de su constructor, sólo define un método tal como el siguiente:

```
synchronized void realizarAccion(String m){  
    System.out.println(m+ " realizara accion");  
}
```



Figura: Aspecto de la interfaz después de pulsar el botón *Lavadora*

Implicaciones en la calidad del patrón Broker

Beneficios	Atributos de calidad	Característica ISO 9126
Separación código de comunicaciones.	Acoplamiento Modificabilidad	Facilita cambios Fac.análisis,pruebas
Independencia de la plataforma de ejecución para la aplicación	Escalabilidad	Facilita cambios Uso de recursos
Mejor descomposición del espacio del problema	Interoperabilidad Simplicidad	Interoperabilidad Facilita análisis
Independencia de la implementación concreta de cada capa	Modificabilidad Flexibilidad	Mantenibilidad Fac.cambios
Interacciones basadas en el paradigma de objetos	Transparencia	Funcionalidad Interoperabilidad
Arquitectura software muy flexible	Dinamismo	Facilita cambios Facilita análisis
Reducción complejidad de programación distribuida	Modificabilidad Flexibilidad	Facilita cambios Facilita análisis
Integración de tecnologías	Reusabilidad	Facilita cambios Facilita análisis
Distribución del modelo de objetos	Interoperabilidad	Portabilidad Adaptabilidad

Implicaciones en la calidad 2

Inconvenientes	Atributos de calidad	Características ISO 9126
Empeora el rendimiento de la aplicación	Rendimiento	Eficiencia Tiempo respuesta
Impide la verticalidad en acceso a capas internas	Optimización Ocultamiento	Facilidad pruebas Uso recursos Tolerancia fallas

- Las capas de abstracción a las que conduce este patrón pueden perjudicar el desempeño
- Usar una capa separada del *Broker* puede ocultar los detalles sobre cómo la aplicación utiliza la capa más baja
- Eventualmente, optimizaciones específicas del rendimiento de algunas capas podrían resultar perjudicadas

Resumen características de calidad

Característica	Sub-característica	Impacto	Atributo
Mantenibilidad	Facilidad de cambio Facilidad de análisis	+	Flexibilidad Escalabilidad Reusabilidad Bajo acoplamiento Modificabilidad Dinamismo
	Facilidad de análisis	+	Simplicidad
	Facilidad de prueba	-	Ocultamiento
Eficiencia	Uso de recursos	+	Escalabilidad
		-	Optimización
	Tiempo de respuesta	-	Desempeño
Funcionalidad	Interoperabilidad	+	Transparencia
Fiabilidad	Tolerancia a fallas	-	Ocultamiento
Portabilidad	Adaptabilidad	+	Multiplataforma

Resumen características de calidad

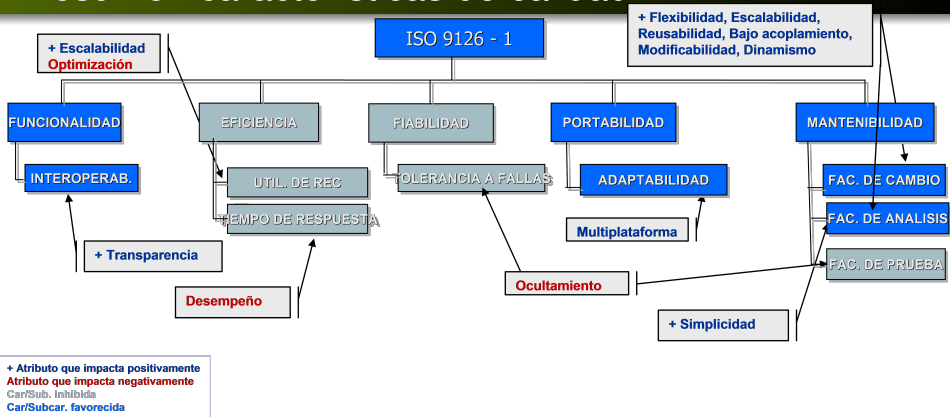
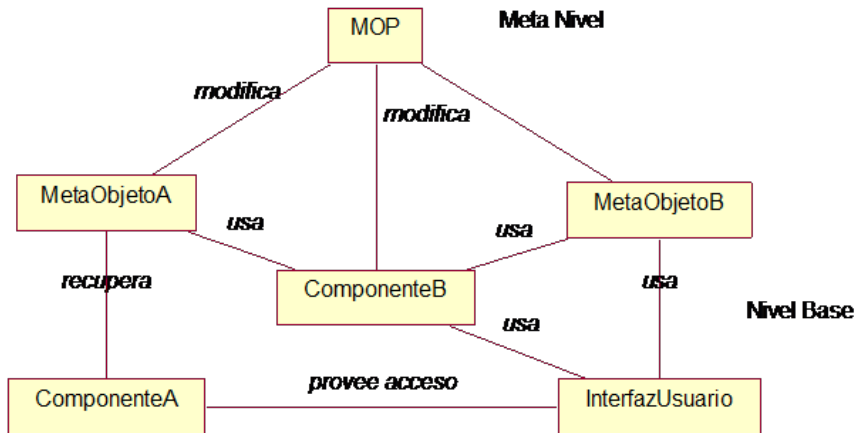


Figura: Características ISO del patrón "Broker"

Patrón Reflection

- Proporciona un mecanismo para cambiar la estructura y comportamiento de sistemas software dinámicamente
- Soporta la modificación de aspectos fundamentales, tales como estructuras y mecanismos de llamadas a métodos.
- Contexto: Cualquier sistema que necesite soporte para realizar cambios propios y para conseguir persistencia de sus entidades
- Problema: ¿Cómo se puede modificar el comportamiento de los objetos de una jerarquía dinámicamente sin afectar a los propios objetos en su configuración actual?
- Solución: Hacer que el software sea “auto-consciente” de su función y comportamiento, haciendo que los aspectos seleccionados sean accesibles para su adaptación y cambio dinámico

Reflection



Estructura de clases del patrón Reflection

- *Meta Nivel*: autoconsciencia de la estructura y funcionamiento del software
- La implementación del *Meta Nivel* utiliza *Meta Objetos*
- *Meta Objetos*: encapsulan y representan información acerca del software
- *Nivel Base*: objetivo y relación con el metanivel
 - Los cambios realizados en el *Meta Nivel* afectan consecuentemente al comportamiento del *Nivel Base*
- Cambios en los metaobjetos y su efecto en los componentes y código del nivel base

Ejemplo de uso del patrón *Reflection*: *juego de la metamorfosis*

Ejemplo demostrativo del patrón *Reflection* que genera clases y métodos utilizando este patrón de diseño y las facilidades reflectivas de Java.

También incluye una interfaz de 4 botones programada con Swing/Java.

Estructura de la aplicación:

- Clase `Bicho`: métodos para fijar el tipo de bicho, obtener su descripción (aparecerá como una etiqueta en el panel inferior) y métodos para que un bicho se “*mueva*” (si puede hacerlo).

Ejemplo de uso del patrón Reflection: *juego de la metamorfosis*–II

La metamorfosis de un bicho puede estar en tres fases distintas: *Gusano* (“worm”), *Capullo* (“cocoon”) y *Mariposa* (“butterfly”).

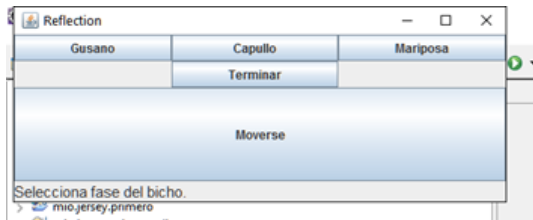


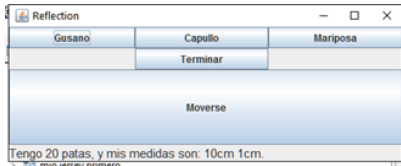
Figura: Aspecto de la interfaz al comenzar la aplicación

Ejemplo de uso del patrón Reflection: *juego de la metamorfosis*—III

```
public class Bicho {
    public String tipo;
    public void setTipo(String t){
        tipo= t;
    }
    public String getTipo(){
        return tipo;
    }
    public String descripcionGusano(int patas, int longitud, int ancho){
        return "Tengo: "+patas+" patas, y mis medidas son: "+longitud+"cm, "+ancho+"
            cm.";
    }
    public String descripcionCapullo(int radio){
        return "Mido: "+radio+"cm de radio.";
    }
    public String descripcionMariposa(int longitud, int ancho){
        return "Mido: "+longitud+"cm de longitud y "+ancho+"cm de ancho.";
    }
    ...
}
```

Ejemplo de uso del patrón Reflection: *juego de la vida*–IV

```
....  
public String mover(int velocidad){  
    if (tipo=="Gusano"){  
        return "Voy caminando:"+velocidad+"Km/h.";}  
    else if (tipo=="Capullo"){  
        return "Soy un capullo, no me puedo mover.";}  
    else{return "Voy volando:"+velocidad+"Km/h.";}  
    }  
}
```



Ejemplo de uso del patrón Reflection: *juego de la vida*-V

```
\\Clase Interfaz --parte Reflectiva  
Class cls = Class.forName("com.ds_reflection.Bicho");  
Object obj = cls.newInstance();  
Method metodo;
```

Instalación de botones

```
public void actionPerformed(ActionEvent evt){//Utilizar para: new JButton("Gusano")  
Class[] paramString = new Class[1];  
Class[] paramGusano = new Class[3];  
...  
paramString[0]= String.class;  
String tipo= "Gusano", salidal;  
try {  
    metodo = cls.getDeclaredMethod("setTipo", paramString);  
    metodo.invoke(obj, tipo);  
    metodo = cls.getDeclaredMethod("descripcionGusano", paramGusano);  
    Integer[] pam = {20,10,1};  
    salidal = (String) metodo.invoke(obj, pam);  
    textField.setText(salidal);  
}  
catch (NoSuchMethodException | SecurityException | IllegalAccessException |  
        IllegalArgumentException | InvocationTargetException e) {  
    e.printStackTrace(); }  
}; //actionPerformed()
```


Ejemplo de uso del patrón Reflection: *juego de la vida*–VI

```
\\Clase Interfaz --parte Reflectiva  
Class cls = Class.forName("com.ds_reflection.Bicho");  
Object obj = cls.newInstance();  
Method metodo;
```

Instalación del botón “Moverse”

```
public void actionPerformed(ActionEvent e){  
    Class[] paramVelocidad = new Class[1];  
    paramVelocidad[0] = Integer.TYPE;  
  
    ...  
    try {  
        metodo = cls.getDeclaredMethod("getTipo", null);  
        String tipo = (String)metodo.invoke(obj, null);  
        metodo = cls.getDeclaredMethod("mover", paramVelocidad);  
        int velocidad = 1;  
        salida2 = (String) metodo.invoke(obj, velocidad);  
        textField.setText(salida2);  
    }  
    catch (NoSuchMethodException | SecurityException | IllegalAccessException |  
            IllegalArgumentException | InvocationTargetException e) {  
        e.printStackTrace();  
    }  
}; //actionPerformed()
```

Implicaciones en la calidad del patrón Reflection

Beneficios	Atributos de calidad	Características ISO 9126
Comprobación correctitud desde el nivel de <i>MetaObjetos</i>	Correctitud	Funcionalidad Precisión
Ejecución de los cambios desde el nivel de <i>MetaObjetos</i>	Dinamismo	Facilidad cambios Facilidad análisis
Modificación y extensión de componentes software facilitada	Modificabilidad Extensibilidad	Mantenibilidad Facilidad cambios
Cambios en el software del <i>Nivel/Base</i> facilitados	Modificabilidad Extensibilidad	Mantenibilidad Facilidad cambios
Asume modificaciones no explícitas del código fuente	Extensibilidad	Mantenibilidad Facilidad cambios
Soporte para muchos tipos de cambios	Modificabilidad	Mantenibilidad Facilidad cambios

Implicaciones en la calidad 2

Inconvenientes	Atributos de calidad	Características ISO 9126
Complejidad de diseño e implementación por los niveles y protocolo meta–objetos	Complejidad	Facilidad análisis Facilidad pruebas
Demasiados meta–objetos en la aplicación final	Rendimiento	Uso recursos Eficiencia
Modificaciones en meta–nivel pueden causar daños comportamiento del sistema	Fallas	Madurez Tolerancia a fallas
Rendimiento sistema puede ser afectado por complejidad diseño del patrón	Complejidad	Eficiencia Tiempo respuesta
Difícil adaptación a la evolución de los productos generados con el patrón	Adaptabilidad	Tiempo respuesta Eficiencia

Resumen características de calidad





Característica	Sub-característica	Impacto	Atributo
Mantenibilidad	Facilidad de cambio Facilidad de análisis	+	Modificabilidad Extensibilidad Dinamismo
	Facilidad de análisis	-	Complejidad
	Facilidad de prueba		
Funcionalidad	Precisión	+	Correctitud
Eficiencia	Uso de recursos	-	Desempeño Complejidad
Fiabilidad	Madurez	-	Propensión a fallas
	Tolerancia a fallas		

Resumen características de calidad



Figura: Características ISO del patrón "Reflection"

Bibliografía Fundamental

-  Bass, L., Clements, P., and Kazman, R. (2012).
Software Architecture In Practice.
Third edition. Addison–Wesley, Boston, Massachussets.
-  Booch, G. (2008).
Handbook of Software Architecture.
<http://www.booch.com/systems.jsp>.
-  Bruegge, B. and Dutoit, A. (2004).
Object–Oriented Software Engineering.
Prentice-Hall, Upper Saddle River, NJ.
-  Buschmann, F. and et al. (2004).
Pattern-oriented software architecture, volume I.
Wiley, Chichester, England.