[75.10] Técnicas de Diseño

María Inés Parnisari

18 de julio de 2014

Índice

Parte I
Resumen

Técnicas de diseño procedimientos y métodos para diseñar una solución a un problema. Tiene *dependencias*:

1. **Requerimientos**: funcionales y no funcionales
2. **Paradigmas**: son los puntos de vista para atacar un problema
3. **Metodologías de desarrollo**: dependen del tipo de problema (grande o chico)
4. **Tecnologías**: lenguajes de programación

1 Construcción de software

Etapas:

1. Definición del problema
2. Análisis de requisitos
3. Planeamiento
4. Diseño de alto nivel (arquitectura)
5. Diseño detallado
6. Implementación
7. Integración
8. Pruebas unitarias
9. Mantenimiento correctivo
10. Mejoras funcionales

1.1 Definición del problema

Es importante que sea correcto porque sino, perdemos tiempo resolviendo el problema equivocado.
Ejemplo: "Necesitamos soportar la cantidad de pedidos que nos realizan"

1.2 Análisis de requisitos

Los requisitos pueden cambiar en el tiempo, porque el usuario no siempre sabe lo que quiere.

- Siempre verificar que se está trabajando para cumplir los requisitos.
- Hacerle saber al cliente lo que implica un cambio de requisitos (costo y dinero).
- Establecer un procedimiento para lidiar con los cambios de requisitos.
- Usar ciclos cortos de desarrollo, que agregan una nueva funcionalidad en cada iteración.

1.3 Diseño de alto nivel (arquitectura)

Distribuir el sistema en módulos, cuya función esté bien definida. Cada módulo debe saber lo menos posible del resto de los módulos. Cada módulo debe tener una interfaz bien definida.
Componentes de la arquitectura:

- Organización del programa
- Estrategia de cambios de requisitos
- Decisiones de tipo "construir vs comprar"
- Estructuras de datos principales
- Algoritmos principales
- Objetos principales (en sistemas orientados a objetos)
- Interfaz de usuario
- Procesamiento de errores
- Procesamiento de parámetros inválidos
- Robusticidad del sistema
- Tolerancia a fallos
- *Performance*

2 Diseño de rutinas

Rutina función individual que tiene un solo objetivo.

Función rutina que devuelve un valor.

Procedimiento rutina que no devuelve un valor.

PDL (*Program Design Language*) es una metodología por la cual se escribe en lenguaje natural un pseudocódigo de la rutina, de tal forma que su conversión en código es rápida. No debe atarse a características del lenguaje en el que se va a programar.
Pasos para diseñar una rutina:

- Chequear los pre-requisitos
- Definir el problema que va a resolver la rutina
- Nombrar la rutina
- Decidir cómo testear la rutina
- Buscar los algoritmos apropiados
- Escribir el PDL
- Pensar sobre la eficiencia
- Pensar sobre los datos

Ventajas del uso de rutinas:

- Reducen la complejidad
- Evitan el código duplicado
- Limita los efectos de los cambios
- Ocultan la secuencialidad de las operaciones

2.1 Características de una buena rutina

1. El nombre nos dice qué hace la rutina
2. Está bien documentada
3. No utiliza variables globales, sino que se comunica con otras rutinas
4. **Fuerte cohesión**: tiene un objetivo claro
5. **Poco acoplamiento**: la rutina puede ser llamada fácilmente por otras rutinas
6. Es defensiva contra datos inválidos
7. No usa números mágicos
8. Utiliza todos los parámetros
9. Recibe hasta 7 parámetros

3 Diseño de clases

Pasos para un buen diseño:

1. Identificar los objetos y sus atributos (métodos y datos).
2. Determinar qué se puede hacer con cada objeto.
3. Determinar lo que un objeto puede hacerle a otros objetos.
4. Determinar las partes del objeto que serán visibles a otro objeto.
5. Determinar la interfaz pública del objeto.

(a) Usar abstracciones que nos ayudan a lidiar con la complejidad.

(b) Usar encapsulación: "tenés permitido usar una vista simple de un concepto complejo, pero no tenér permitido ver los detalles de ese concepto"

Figura 1: Principios

Heurísticas de buen diseño:

- Usar herencia cuando sea apropiado
- Usar abstracciones
- Usar encapsulación
- Ocultar detalles de implementación
- Identificar posibles áreas de cambio, y separarlas del resto
- Usar patrones de diseño
- Documentar las precondiciones y postcondiciones con *asserts*
- *The Principle of One Right Place—there should be One Right Place to look for any nontrivial piece of code, and One Right Place to make a likely maintenance change*

4 Diseño de software

Diseño de software concepción de un esquema que ayude transformar la especificación de requisitos en un programa operacional.

1. Es un problema "malvado" que solo se resuelve "resolviéndolo".
2. Es un proceso "descuidado" porque se dan muchos pasos en falso y se cometen muchos errores, y nunca se sabe si el diseño es lo suficientemente bueno.
3. Se trata de balancear prioridades con desventajas.
4. Es un proceso heurístico; no siempre las herramientas funcionan bien en todos los procesos.
5. Es un proceso "emergente" que se desarrolla y evoluciona

Hay dos tipos de problemas que hacen que el diseño de software sea complicado:

- Problemas esenciales: surgen de la necesidad de modelar el mundo real y las muchas interconexiones que se producen en él.
- Problemas accidentales: la mayoría fueron solucionados con la evolución de los lenguajes de programación.

El mayor problema es la **complejidad**. El objetivo es reducir la complejidad del sistema utilizando módulos y sub módulos independientes entre sí, para que en en cualquier momento el programador no tenga que lidiar, mentalmente, con muchos objetos a la vez.
Características deseables de un buen diseño:

- Mínima complejidad: preferir diseños simples y fáciles de entender por sobre diseños "astutos" que nadie entiende
- Fácil de mantener
- Mínimas interconexiones
- Extensibilidad
- Reusabilidad
- *High fan-in*: tener muchas clases que utilicen una clase.
- *Low-to-medium fan-out*: una clase no debe utilizar muchas otras clases.
- Portabilidad
- No debe tener funcionalidad extra
- Estratificación: si debemos utilizar un código viejo mal escrito, escribir una clase de servicio que interactúe con dicho código.

4.1 Proceso de diseño de software

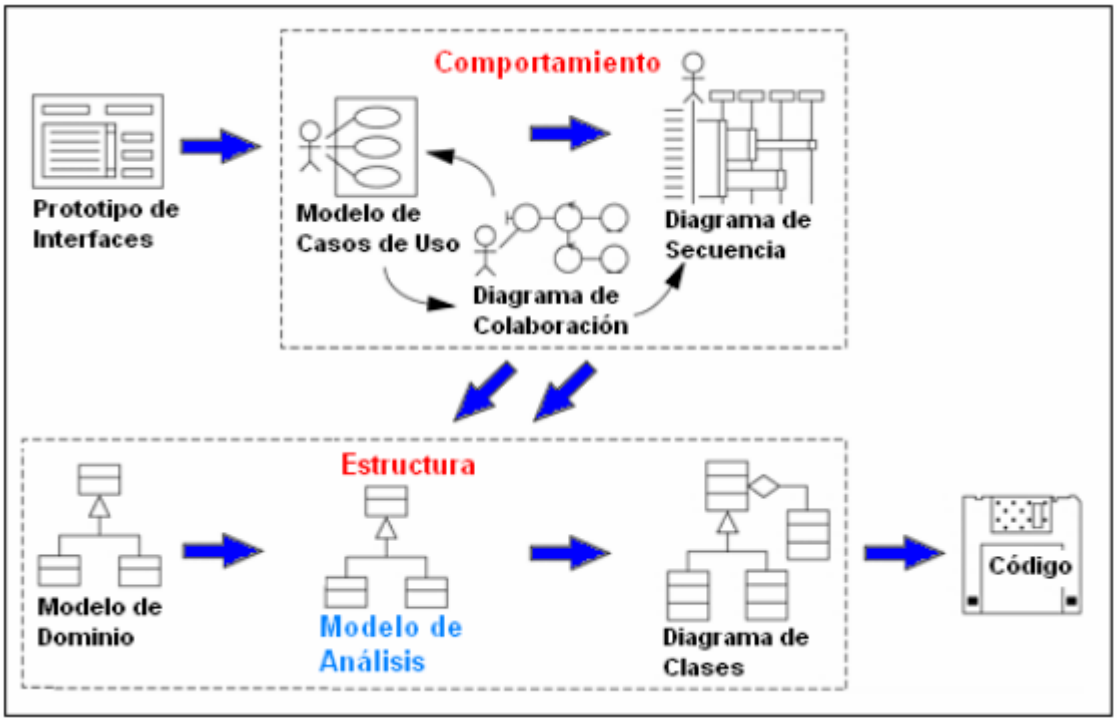


Figura 2: Proceso de desarrollo de software

Paso 1: relevar requerimientos a partir de las necesidades de los usuarios (prototipo de interfaces).

Paso 2: especificar los casos de uso para entender el comportamiento (diagramas de colaboración y/o secuencia).

Paso 3: pensar una estructura simple a partir de conceptos extraídos del dominio de problema (modelo de dominio).

Paso 4: refinar el modelo de dominio a un modelo de análisis.

4.1.1 Modelo de análisis

Su objetivo es entender el negocio y sus reglas.

Concepto	Instancia	Descripción	Patrones de colaboración[1]
Gente	Actor	No conoce el contexto. Puede conocer muchos roles, pero solo toma uno por vez.	Actor - Rol
	Rol	No puede existir sin un actor. Conoce al contexto.	
Lugares	Lugar		Gran Lugar - Lugar
	Gran lugar		
Cosas	Ítem		Ítem - Ítem específico
	Ítem específico	Se utiliza para modelar objetos que pueden existir en muchas variaciones. No pueden existir sin su ítem correspondiente.	
	Ensamble	Debe contener al menos una parte.	Ensamble - Parte
	Parte	Puede existir sin estar en un ensamble. No puede estar en más de un ensamble.	
	Contenedor	Puede no contener nada.	Contenedor - Contenido
	Contenido	No puede estar en más de un contenedor.	
	Grupo		Grupo - Miembro
	Miembro	Puede pertenecer a más de un grupo.	
Eventos	Transacción	Representa un evento.	▪ Rol - Transacción ▪ Lugar - Transacción ▪ Ítem específico - Transacción
	Transacción compuesta	Representa un evento que involucra más de un objeto.	Transacción compuesta - *Line item*
	Transacción cronológica	Sucede antes o después de otra transacción.	Transacción - Transacción cronológica
	Line item	Captura detalles de un ítem específico con relación a una transacción compuesta.	Ítem específico - *Line item*

Cuadro 1: Guía de selección de objetos en el modelo de análisis

Regla de negocio son las restricciones que gobiernan las acciones dentro de un dominio de negocio. En el modelo se traducen en **reglas de colaboración**. Se debe decidir qué colaborador (objeto) prueba cada regla, en función de lo que conoce o puede consultar.

Regla de colaboración chequeo de reglas de negocio entre objetos participantes de relaciones.

Tipos de reglas:

1. **Tipo**: *¿el potencial colaborador es del tipo correcto?*
2. **Multiplicidad**: *¿qué pasa si agrego o elimino colaboradores?*
3. **Propiedad**: verificar los valores de las propiedades mías o de un potencial colaborador contra un valor estándar
4. **Estado**: *¿estoy en el estado apropiado para formar o disolver una colaboración?*
5. **Conflicto**: *¿alguno de mis colaboradores actuales está en conflicto con el potencial colaborador?*

¿Qué colaborador chequea cada regla?

1. Cuando se modela gente, lugares o cosas, *siempre* asignar las reglas a los objetos más específicos.
2. Cuando se modelan objetos y sus partes, *siempre* asignas las reglas a los objetos que son "parte de".
3. Cuando la gente, lugares o cosas interactúan con un evento, asignarles las reglas a ellos.
4. Cuando existen transacciones en secuencia, la transacción precedente es la que chequea las reglas.

Algoritmo 1 Métodos para aplicar reglas de colaboración

```
add(aCollaborator);
remove(aCollaborator();

testAdd(aCollaborator);
testRemove(aCollaborator);

testAddConflict(directCollaborator, indirectCollaborator, ...);
testRemoveConflict(directCollaborator, indirectCollaborator, ...);

doAdd(aCollaborator);
doRemove(aCollaborator);
```

Propiedades de los objetos

1. Descriptivos
2. De tiempo
3. Estado de vida
4. Estado operativo
5. Rol
6. Tipo

Algoritmo 2 Métodos para aplicar reglas de propiedad

```
set(aValue);
setValue();

testSet(aValue);
testSetValue();

doSet(aValue);
doSetValue();
```

Modelo de diseño implementar una solución al problema planteado en el análisis más las restricciones impuestas por los requerimientos no funcionales.

5 Diseño de paquetes

5.1 Cohesión

- *The unit of reuse is the unit of release*
- *Classes that change together, belong together*

5.2 Acoplamiento

- *Packages must not be indirectly dependent on themselves*

5.2.1 Estabilidad y abstracción

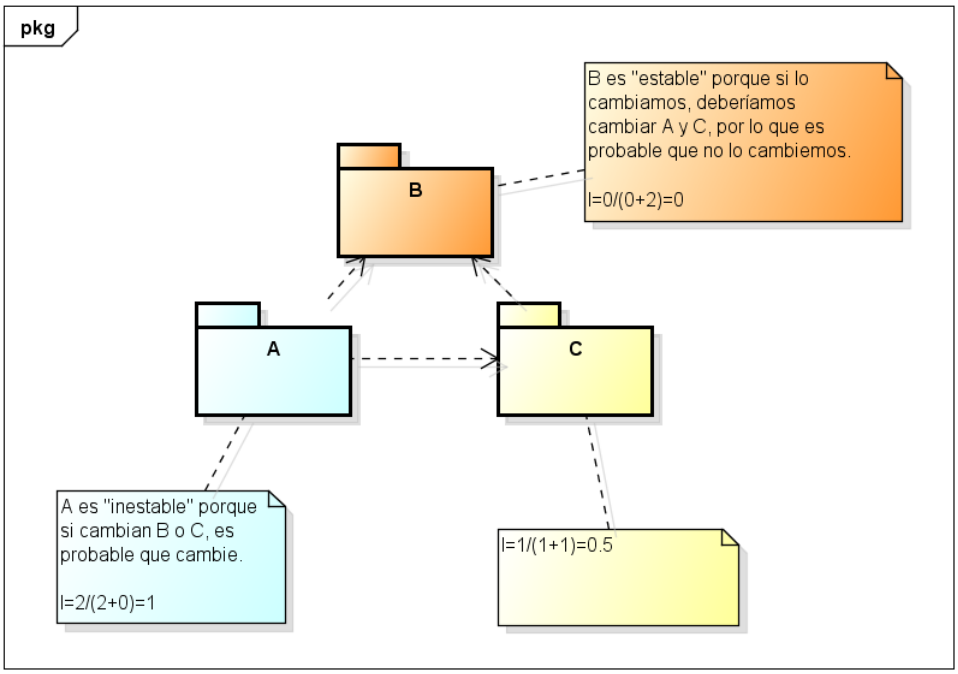


Figura 3: Un paquete debe depender de paquetes más estables

Métricas Sea el paquete P, que depende de X paquetes, y del cual Y paquetes dependen de él. La **inestablidad** de P se define como:

$$I = \frac{X}{X+Y}$$

$I \in [0,1]$. Si $I = 1$, el paquete es "inestable". Si $I = 0$, el paquete es "estable".

Sean las clases concretas C_C y las clases abstractas C_A dentro del paquete P. La **abstracción** de P se define como:

$$A = \frac{C_A}{C_A + C_C}$$

$A \in [0,1]$. Si $A = 1$, el paquete es "abstracto". Si $A = 0$, el paquete es "concreto".

Un paquete "estable" es fácil de extender. Un paquete "inestable" es fácil de modificar. Por ende, los paquetes "inestables" deben depender de paquetes más "estables".

Los **conceptos** no suelen cambiar mucho, por ende deben hacerse abstractos. Los **objetos** de la vida real son inestables, y por ende deben hacerse concretos. Si las abstracciones son buenas, no cambiarán.

6 Principios de diseño y métricas

1. Tomar lo que varía y separarlo de lo que no
2. Programar contra una interfaz, no contra una implementación
3. Preferir la composición por sobre la herencia
4. Preferir diseños desacoplados
5. Clases abiertas para extensión pero cerradas para modificación
6. Depender de abstracciones, no de clases concretas (tanto las clases de alto nivel como las de bajo nivel)

7. Principio de *least knowledge*: hablar sólo con los amigos más cercanos
8. Una clase sólo debe tener una razón para variar.

Principio	**Métrica**	**Automatización posible?**
Cohesión de clases	LCOM	Si
SRP	Por inspección de código, responsabilidades por clase	
OCP	Por cada *commit*, cantidad de clases modificadas y/o extendidas	Si
LSP	Cada clase debe pasar los tests unitarios de su superclase	
ISP	Promedio $\frac{n}{N}$ para todos los clientes de I	Si
DIP	$\frac{d_A}{D}$	Si
Ley de Demeter	Promedio de profundidad de navegación	Si
Reutilización de abstracciones	Cantidad de clases que extienden o implementan una clase o interfaz	Si

Cuadro 2: Principios de diseño

6.1 Alta cohesión

Lack of Cohesion of Methods (LCOM): sea la clase C con A atributos, M métodos y $R(A)$ métodos que utilizan el atributo A.

$$LCOM = \frac{\left(\frac{\sum R(A)}{A}\right) - M}{1 - M}$$

6.2 Ley de Demeter

Los objetos solo deben colaborar con sus vecinos más cercanos. Esto significa que hay que evitar "llamadas de llamadas".

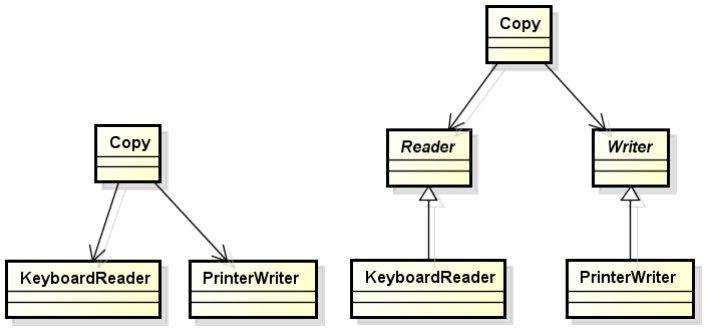
Sólo se permite invocar métodos que pertenecen a:

1. El objeto mismo
2. Un objeto pasado por parámetro
3. Objetos que el método crea
4. Componentes del objeto

6.3 Principios SOLID

6.3.1 Principio de inversión de dependencia

Consider the implications of high level modules that depend upon low level modules. It is the high level modules that contain the important policy decisions and business models of an application. It is these models that contain the identity of the application. Yet, when these modules depend upon the lower level modules, then changes to the lower level modules can have direct effects upon them; and can force them to change.



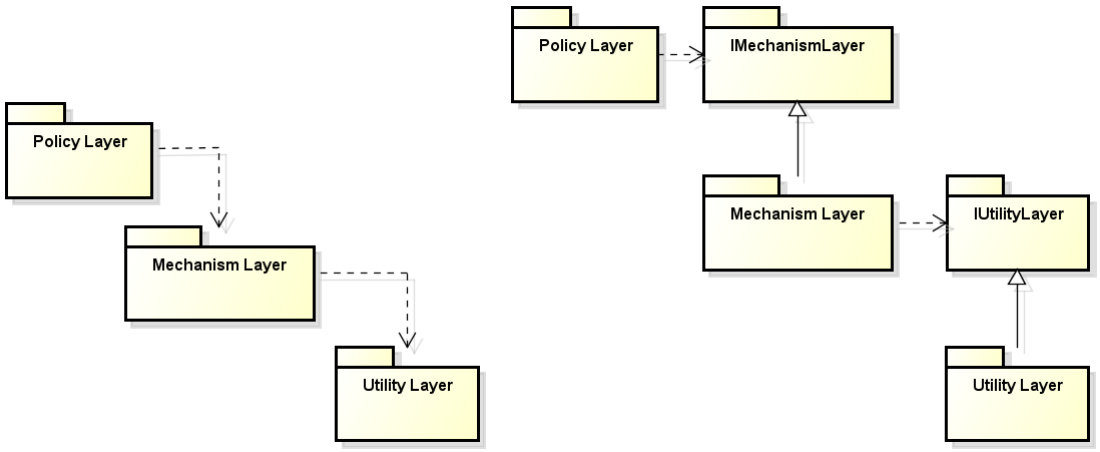
(a) La clase `Copy` depende de los detalles de implementación

(b) La clase `Copy` no depende de los detalles de implementación

Figura 4: Inversión de dependencia

1. Los módulos de alto nivel no deben depender de módulos de bajo nivel. Ambos deben depender de abstracciones.
2. Las abstracciones no deben depender de los detalles. Los detalles deben depender de las abstracciones.

All well structured object-oriented architectures have clearly-defined layers, with each layer providing some coherent set of services though a well-defined and controlled interface.



(a) Sin inversión

(b) Con inversión

Figura 5: Inversión de dependencia

Dependency Inversion can be applied wherever one class sends a message to another.

Ejemplo: se quiere implementar una clase Botón que detecta cuando el mismo fue presionado o soltado, y enviarle esta información a una Lámpara, que se prende o apaga. ¿Cuál es la política de alto nivel? Es la abstracción de que hay que detectar un gesto de on/off y forwardearselo a un objetivo. No importa cómo se detecta el gesto, no importa cuál es el objetivo.

Métrica si la clase C depende de d_A clases abstractas y D clases en total:

$$\frac{d_A}{D}$$

6.3.2 Principio de sustitución de Liskov

1. Las funciones que usan referencias a clases base, deben poder usar objetos de clases derivadas sin saberlo.

If there is a function which does not conform to the LSP, then that function uses a pointer or reference to a base class, but must know about all the derivatives of that base class. Such a function violates the Open-Closed principle because it must be modified whenever a new derivative of the base class is created.

When the creation of a derived class causes us to make changes to the base class, it often implies that the design is faulty

When considering whether a particular design is appropriate or not, one must not simply view the solution in isolation. One must view it in terms of the reasonable assumptions that will be made by the users of that design.

Ejemplo: tenemos una clase base Rectángulo que tiene dos atributos, alto y ancho. Queremos implementar la clase Cuadrado. Si hacemos que ésta herede de Rectángulo y modificamos los getters y setters de ancho y alto para que "matemáticamente hablando" el ancho y el alto sean correctos (iguales), el siguiente test fallará:

```
public void TestMultiplicarAnchoPorAlto(Rectangulo &rect)
{
  rect.SetAncho(5);
  rect.SetAlto(4);
  // Lo siguiente fallara si r es un Cuadrado
  Assert.IsTrue(rect.GetAncho() * rect.GetAlto() == 20);
}
```

Esto es porque el programador asumió, de manera razonable, que el método SetAncho() no cambia el Alto y viceversa, algo que no es cierto para la clase Cuadrado.

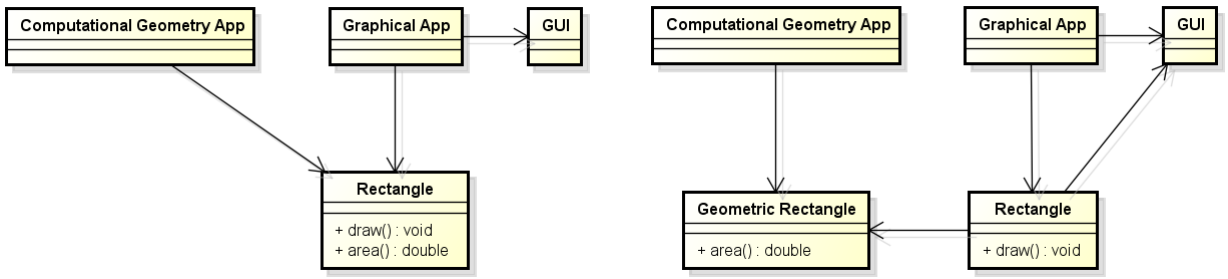
The LSP makes clear that in OOD the ISA relationship pertains to behavior, behavior that clients depend upon.

Now the rule for the preconditions and postconditions for derivatives is: ...when redefining a routine in a derivative, you may only replace its precondition by a weaker one, and its postcondition by a stronger one.

6.3.3 Principio de Responsabilidad Única

1. Nunca debería haber más de una razón para cambiar una clase.

When the requirements change, that change will be manifest through a change in responsibility amongst the classes. If a class assumes more than one responsibility, then there will be more than one reason for it to change. If a class has more then one responsibility, then the responsibilities become coupled. Changes to one responsibility may impair or inhibit the class' ability to meet the others.



(a) Sin SRP (b) Con SRP

Figura 6: Principio SRP

6.3.4 Principio abierto-cerrado

1. Las clases, módulos y funciones deberían estar abiertas para la extensión, y cerradas para la modificación.

When requirements change, you extend the behavior of such modules by adding new code, not by changing old code that already works.

Abstraction is the key. The abstractions are abstract base classes, and the unbounded group of possible behaviors is represented by all the possible derivative classes. It is possible for a module to manipulate an abstraction. Such a module can be closed for modification since it depends upon an abstraction that is fixed. Yet the behavior of that module can be extended by creating new derivatives of the abstraction.

It should be clear that no significant program can be 100 % closed. In general, no matter how "closed" a module is, there will always be some kind of change against which it is not closed. Since closure cannot be complete, it must be strategic. The experienced designer knows the users and the industry well enough to judge the probability of different kinds of changes. He then makes sure that the open-closed principle is invoked for the most probable changes.

Heurísticas y convenciones:

- Atributos deben ser privados.
  - Si un atributo depende de otros, y se permite cambiar el valor de uno de ellos, el estado del objeto puede quedar inconsistente.
- No usar variables globales.
  - Si dos módulos usan variables globales y uno de ellos altera su valor, el otro módulo puede romperse.
- No usar RTTI (RunTime Type Identification)

Algoritmo 3 Si agregamos un nuevo tipo de *Shape*, hay que modificar la función *DrawAllShapes*

```
void DrawAllShapes(Set<Shape*>& ss)
{
  for (Iterator<Shape*>i(ss); i; i++)
  {
    Circle* c = dynamic_cast<Circle*>(*i);
    Square* s = dynamic_cast<Square*>(*i);

    if (c) DrawCircle(c);
    else if (s) DrawSquare(s);
  }
}
```

Conformance to this principle is not achieved simply by using an object oriented programming language. It requires a dedication on the part of the designer to apply abstraction to those parts of the program that the designer feels are going to be subject to change.

6.3.5 Principio de segregación de interfaces

1. Los usuarios no deberían ser forzados a depender de interfaces que no utilizan. En vez de una interfaz "gorda" con muchas operaciones, se prefieren interfaces pequeñas basadas en grupos de operaciones.

Fat interfaces lead to inadvertent couplings beween clients that ought otherwise to be isolated. By making use of the ADAPTER pattern, either through delegation (object form) or multiple inheritance (class form), fat interfaces can be segregated into abstract base classes that break the unwanted coupling between clients.

Métrica: si la interfaz I expone N métodos, y el cliente C utiliza n métodos, entonces la interfaz es $\frac{n}{N}$-específica para C.

Se toma un promedio $\frac{n}{N}$ para todos los clientes de I.

7 Malos diseños

Las siguientes tres características son síntomas de un mal diseño en software:

1. **Rigidez**: es difícil de cambiar porque un cambio afecta muchas otras partes del sistema.

2. **Fragilidad**: al realizar un cambio, se rompen partes inesperadas del sistema.
3. **Inmobilidad**: es difícil de reusar en otra aplicación porque no se puede desenganchar fácilmente de la actual.

Lo que produce que un diseño sea rígido, frágil e inmóvil es la i*nterdependencia entre los módulos.*

8 Comentarios

- Escribir código que esté tan claro que no se necesiten comentarios

Comentarios útiles:

- Legales
- Informativos. *Ejemplo: al usar expresiones regulares.*
- Advertencias
- Amplía información. *Ejemplo: proporciona un link a un algoritmo.*

9 Herramientas de software

- *FxCop*: analiza el código objeto de C# para verificar que cumpla ciertas reglas de diseño de Microsoft.
- *StyleCop*: analiza el código fuente de C#.
- *CheckStyle*: analiza el código fuente de Java.

10 Metáforas de software

Algoritmo instrucciones bien definidas para hacer una tarea. Es predecible, determinístico, y no depende del azar.

Heurística técnica para buscar una solución a un problema.

Una buena metáfora de construcción de software es la construcción de edificios.

11 Refactoring[2]

Refactoring proceso de cambiar un sistema de tal forma que no se altera su comportamiento externo, pero sí cambia su estructura interna.

- *If you need to refactor, do it before adding a feature, then add the feature (7)*
- *"Before you start refactoring, check that you have a solid suite of tests. These tests must be self-checking." (8)*
- *"Never be afraid to change the names of things to improve clarity" (15)*
- *"In most cases a method should be on the object whose data it uses" (17)*
- *"[Temporary variables] are often a problem in that they cause a lot of parameters to be passed around when they don't have to be. You can easily lose track of what they are there for." Use Replace Temp with Query refactoring. (21)*
- *"Don't worry about [performance] while refactoring." After running a profiler you may need to optimize, but you'll be in a better position to do it. (32)*
- *Refactoring to eliminate switch statements with enums: Replace Type Code with State/Strategy (39)*

[2] http://www.rallydev.com/community/engineering/quick-summary-martin-fowler%E2%80%99s-%E2%80%9Crefactoring%E2%80%9D

- *"The rhythm of refactoring: test, small change, test, small change, test, small change" (52)*
- *Kent Beck's two hats: adding function and refactoring. "When you add function you shouldn't be changing existing code; you are just adding new capabilities. You can measure your progress by adding tests and getting the tests to work. When you refactor, you make a point of not adding function; you only restructure the code. You don't add any tests (unless you find a case you missed earlier); you only change tests when you absolutely need to in order to cope with a change in an interface ... always be aware of which hat you're wearing." (54)*
- *Don Roberts'* ***Rule of Three****: "the first time you do something, you just do it. Second time you do something similar, you wince at the duplication, but you do the duplicate thing anyway. The third time you do something similar, you refactor." (58)*
- *What makes programs hard to modify: hard to read programs, programs with duplicate logic, programs that require additional behavior that requires you to change running code, programs with complex conditional logic (60)*
- *When not to refactor: if it's easier to rewrite, current code has bugs, close to a deadline (66)*
- *Generate a plausible first shot at design, code, then refactor. Refactoring changes the role of upfront design. If you don't refactor, there's a lot of pressure in getting that upfront design right. (67)*
- *"Even if you know exactly what is going on in your system, measure performance, don't speculate. You'll learn something, and nine times out of ten, it won't be that you were right!" (69)*
- *"Whenever we feel the need to comment something, we write a method instead. Such a method contains the code that was commented but is named after the intention of the code rather than how it does it." (77)*
- *"Don't forget to test that exceptions are raised when things are expected to go wrong" (100)*
- *"There is a point of diminishing returns with testing, and there is the danger that by trying to write too many tests, you become discouraged and end up not writing any" (101)*
- *When to stop refactoring: when you lose confidence in your next step. If the code you've written makes the system better keep it, otherwise trash it. (410)*
- *Refactor in pairs (411)*
- *"Refactoring first is less dangerous than adding new code. Touching the code will remind you how it works" so that adding new code becomes easier/faster. (412)*

12 Métricas

Fuentes de información del código:

- El código mismo
- La documentación
- Los tests
- La gente que lo escribió
- => MÉTRICAS

Overview pyramid herramienta para visualizar un sistema de forma compacta. Se divide al sistema en tres categorías, y se los analiza con métricas:

- *Inheritance*
- *Coupling*
- *Size & Complexity*

Métricas para cada categoría:

- Size & Complexity
  - NOC (Number of Classes)
  - NOP (Number of Packages)
  - NOM (Number of Methods)
  - LOC (Lines of Code)
  - CYCLO (Cyclomatic Complexity): cuenta la cantidad de caminos independientes que hay en todo el código (*if, for, and, or, try, catch*)

Algoritmo 4 Cálculo de CYCLO

```
cyclo = 1
para cada keyword en (if, while, repeat, for, and, or):
  cyclo = cyclo + 1
para cada case en un case:
  cyclo = cyclo + 1
```

- Coupling (hay tres tipos: intensivo -hablo mucho con uno-, dispersivo -hablo poco con muchos-, combinado)
  - Calls: la cantidad de veces que se llama a una función.
  - Fanout: cuenta los tipos que se referencian por clases e interfaces.
  - ATFD (Access to Foreign Data)
- Inheritance
  - ANDC (Average Number of Derived Classes). Por ejemplo, en un sistema con 10 clases y un valor de ANDC de 0.5, significa que 5 clases heredan de alguna otra clase.
  - AHH (Average Hierarchy Height). Mide el promedio de profundidad en la jerarquía de herencia.

Métrica	Bajo	Promedio	Alto
CYCLO / LOC	0.16	0.20	0.24
LOC / NOM: líneas de código por función	7	10	13
NOM / NOC: funciones por clase	4	7	10
NOC / NOP: clases por paquete	6	17	26
CALLS / NOM	2.01	2.62	3.2
FANOUT / CALLS	0.56	0.62	0.68
ANDC	0.25	0.41	0.57
AHH	0.09	0.21	0.32

Cuadro 3: Valores de referencia

Polimétricas representación visual de las métricas.

12.1 Antipatterns

- Categorías de problemas:
  - ***Brain class***: una clase que sabe demasiado o hace demasiado.
  - ***Brain method***: un método que centraliza la funcionalidad de una clase.
  - ***Feature envy***: una clase que utiliza muchos métodos de otra clase.
  - ***Data class***: clases que solo tienen atributos, *getters* y *setters*.

    Data classes are like children. They are okay as a starting point, but to participate as a grownup object, they need to take some responsibility.

  - ***Significant duplication***

- Problemas de herencia
  - ***Refused parent bequest***: una clase hija que no necesita métodos de su clase padre.
    - Crear una clase hermana y utilizar Push Down Method y Push Down Field
  - ***Tradition breaker***: una clase hija que provee servicios que no se relacionan con los que provee su clase padre
  - Puede haber código repetido entre subclases
  - Un cambio en la clase madre puede inintencionalmente afectar a las subclases
  - En tiempo de ejecución, los cambios de comportamiento son difíciles
  - No se puede reemplazar el comportamiento por otro para testearlo
- Problemas de entidades
  - ***Intensive coupling***
  - ***Dispersed coupling***
  - ***Divergent change***: una clase que se la cambia muy frecuentemente por muchas razones.
  - ***Shotgun surgery***: cada vez que hacemos un cambio, tenemos que cambiar algo en muchas clases

Tight Class Cohesion (TCC) mide cuántos atributos usan los métodos. Si un método usa pocos atributos de una clase, la misma podría no ser cohesiva.

Parte II
Patrones de diseño

Patrón *solución* a un *problema* en un *contexto*. El problema debe ser *recurrente*. Un patrón proporciona *reuso de experiencia* y permite *compartir vocabulario* entre desarrolladores.

Categorías de patrones:

- **Arquitectura**: al nivel de paquete.
- **Diseño**: al nivel de clase.
- ***Idiom***: asociados con el código.

Otra categoría de patrones:

- **Creacionales**: lidian con la instanciación de objetos.
  - *Singleton, Factory Method, Abstract Factory, Prototype, Builder*
- **Estructurales**: lidian con la composición de objetos en estructuras más grandes
  - *Composite, Decorator, Proxy, Facade, Adapter, Bridge (Flyweight)*
- **De comportamiento**: lidian con cómo los objetos interactúan entre sí y se distribuyen responsabilidades
  - *Template Method, Command, Observer, Iterator, Strategy, State, Mediator, Visitor, Chain of Responsibility (Memento, Interpreter)*

13 *Double dispatch*

Mecanismo para llamar una función dependiendo del tipo del objeto.

Algoritmo 5 Ejemplo de *double dispatch* (en este caso triple)

```
void displayShape (Shape shape, Brush brush)
{
  shape.displayWith(brush);
}

class Oval extends Shape
{
  public void displayWith(Brush brush)
  {
    brush.DisplayOval(this);
  }
};

class PostscriptBrush extends Brush
{
  public void DisplayOval(Oval oval)
  {
    System.out.writeln("Displaying␣an␣oval");
  }
};
```

Ventajas	Desventajas
	Duplicación de código
	Falta de flexibilidad (si agrego un *Shape* tengo que agregar un método en *Brush*)

14 Creacionales

14.1 *Singleton*

Contexto: Necesitamos una funcionalidad que pueda accederse desde el resto del sistema. Esa funcionalidad debe ser provista por un solo objeto.

Problema: Garantizar que una clase tenga una sola instancia, y proporcionar un punto de acceso global a ella.

Solución: Una clase es responsable de controlar la cantidad de instancias que posee, y de proporcionar un único punto de acceso a ella. Su constructor se declara como privado.

Ventajas	Desventajas
Acceso controlado a una única instancia	No está garantizado el orden de construcción y destrucción de los objetos estáticos
Es una mejora con respecto a las variables globales (estas últimas solo proveen *early instantiation*)	Puede haber problemas en aplicaciones *multithreaded* (3 soluciones: *syncronized getInstance, create instance eagerly, double-checked locking*)

Algoritmo 6 Clase *Singleton*

```
public class Singleton {
  private static Singleton instance;
  private int data;

  private Singleton() {
    data = 0;
  }

  public static Singleton getInstance() {
    if (instance == null)
      instance = new Singleton();
    return instance;
  }
}
```

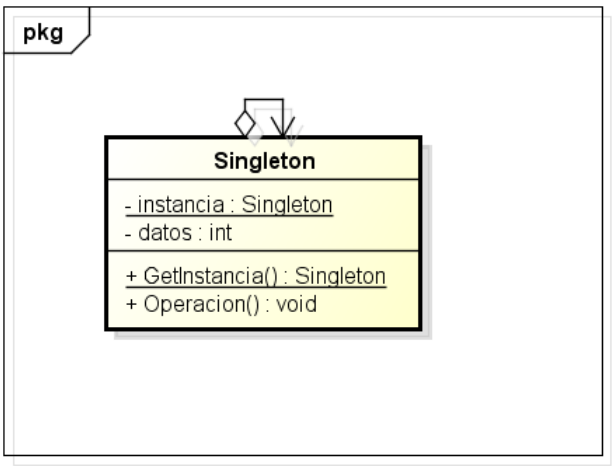


Figura 7: Patrón *Singleton*

14.2 *Factory Method*

When you have code that instantiates concrete classes, this is an area of frequent change.

The Factory Method Pattern defines an interface for creating an object, but lets subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses. The creator class is written without knowledge of the actual products that will be created, which is decided purely by the choice of the subclass that is used.

Contexto: Una clase no puede prever qué clase de objetos debe crear, quiere que sean sus subclases quienes especifiquen qué objetos se crean.

Solución: Utiliza la herencia. La creación de objetos se delega a subclases que implementan el *factory method* que crea objetos. El *factory method* tiene la siguiente estructura:

```
abstract Producto crearProducto(int tipo)
```

Factory Method es un caso especial de *Template Method* (la subclase implementa todo el algoritmo definido en la clase abstracta)

Ventajas	Desventajas
Conecta jerarquías de clases paralelas (productos y creadores)	Se necesitan agregar Creadores por cada nuevo Producto
Bajo acoplamiento entre los creadores y los productos concretos	

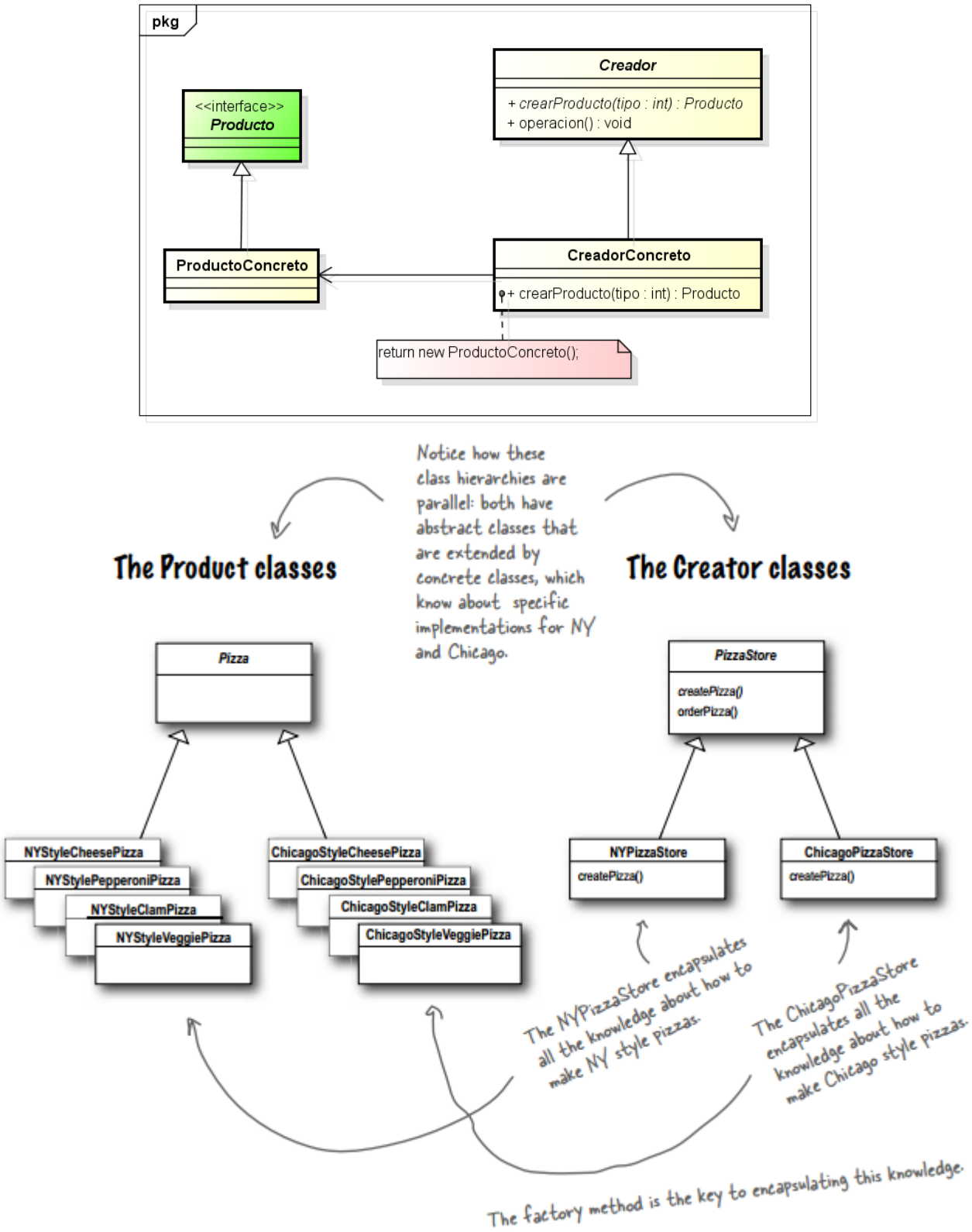


Figura 8: Patrón *Factory Method*

14.3 *Abstract Factory*

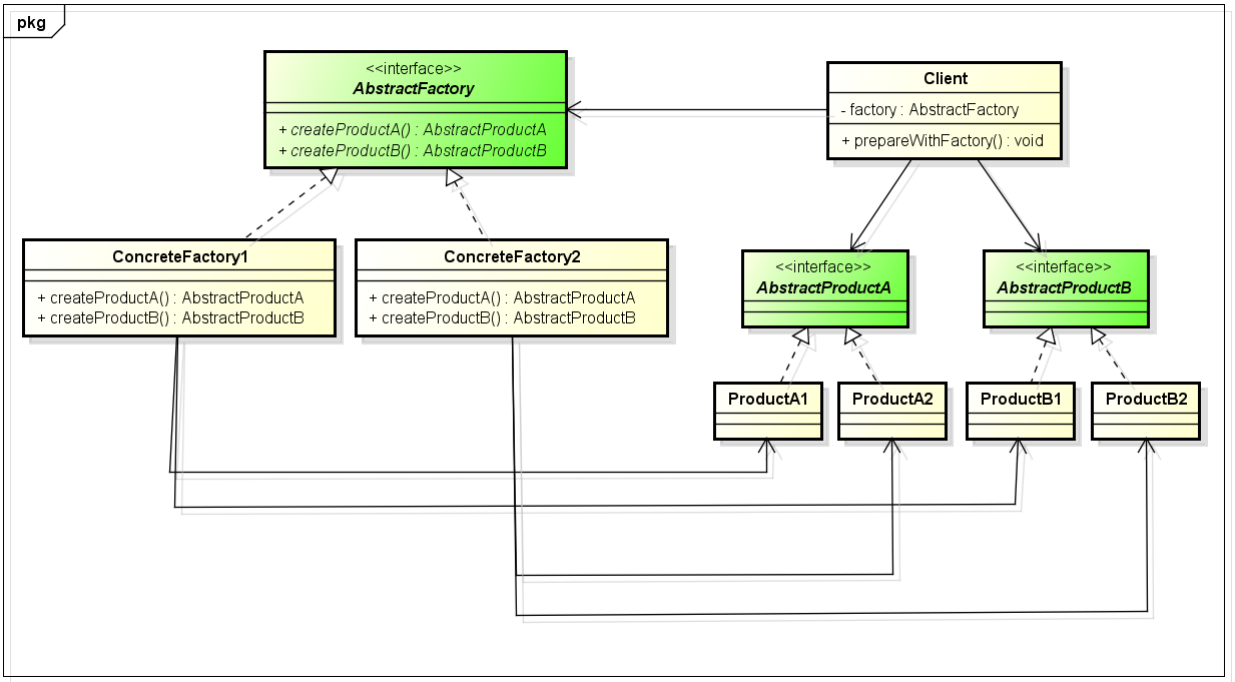
Provide an interface for creating families of related or dependent objects without specifying their concrete classes.

Contexto: Un sistema debe ser independiente de cómo se crean, componen y representan sus productos. La familia de productos puede variar. Los productos sólo pueden ser utilizados junto a su familia.

Problema: ¿cómo crear familias de objetos similares sin tener que depender de clases concretas?

Solución: Utilizar composición. La creación de objetos se implementa en métodos expuestos en la interfaz de la *factory*.

Ventajas	Desventajas
	Si se agrega una nueva familia de productos, hay que modificar la interfaz de la *factory*



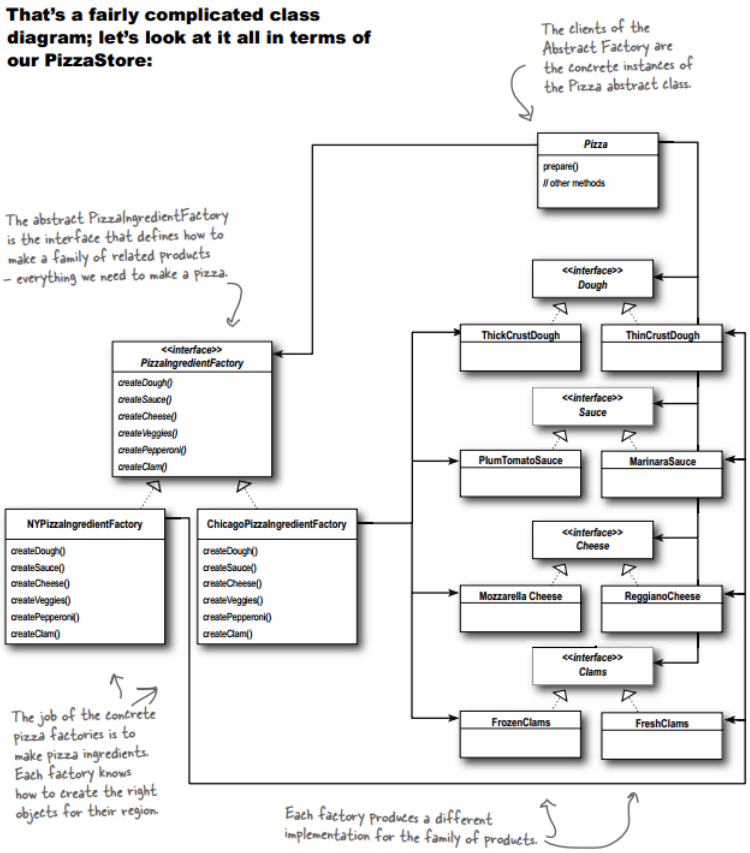


Figura 9: Patrón *Abstract Factory*

14.4 *Prototype*

Use when creating an instance of a class is either expensive or complicated. Allows to make new instances by copying existing ones.

Contexto: El sistema debe ser independiente de cómo se crean, componen y representan los objetos. Las clases a instanciar deben ser especificadas en tiempo de ejecución. Las instancias de una clase pueden tener uno de entre solo unos pocos estados diferentes.

Problema: Especificar los tipos de objetos a crear.

Solución: Un objeto prototipo es clonado para producir nuevos objetos. Estos nuevos objetos tienen los mismos atributos y los mismos métodos que la clase "madre".

Aplicabilidad: Cuando las clases a instanciar sean especificadas en tiempo de ejecución.

Ventajas	Desventajas
Permite añadir y eliminar clases en tiempo de ejecución	Cada subclase de Prototipo debe implementar la operación `clonar`
Reduce la herencia	
Reduce el costo de crear nuevos objetos	

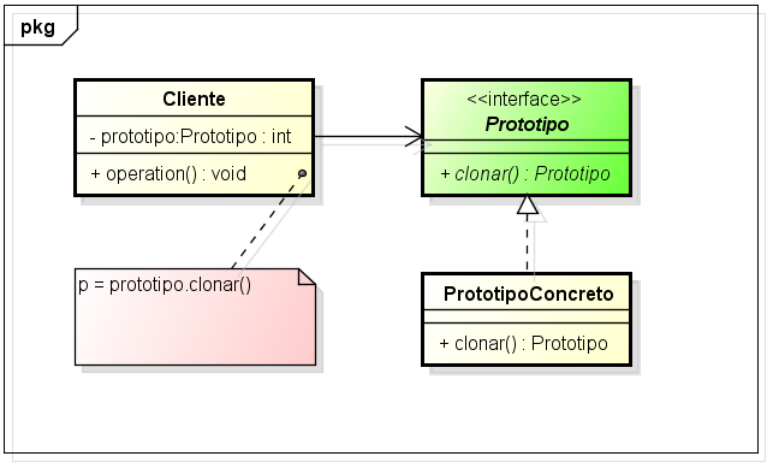


Figura 10: Patrón *Abstract Factory*

14.5 *Builder*

Use to encapsulate the construction of a complex object and allow it to be built in steps.

Contexto: hay una entrada común y muchas representaciones posibles.

Problema: ¿cómo encapsular la construcción de un objeto complejo, y permitir crearlo en pasos? ¿cómo permitir crear varias representaciones del objeto?

Solución: un objeto **director** interpreta el pedido e invoca al ***builder***. El ***builder*** crea las partes del objeto complejo. El **cliente** recibe el resultado final del ***builder***.

Ventajas	Desventajas
Permite variar la representación interna de un objeto	Se requiere más conocimiento del dominio para construir los objetos
Aísla el código de representación del cliente	
Da más control sobre el proceso de construcción	

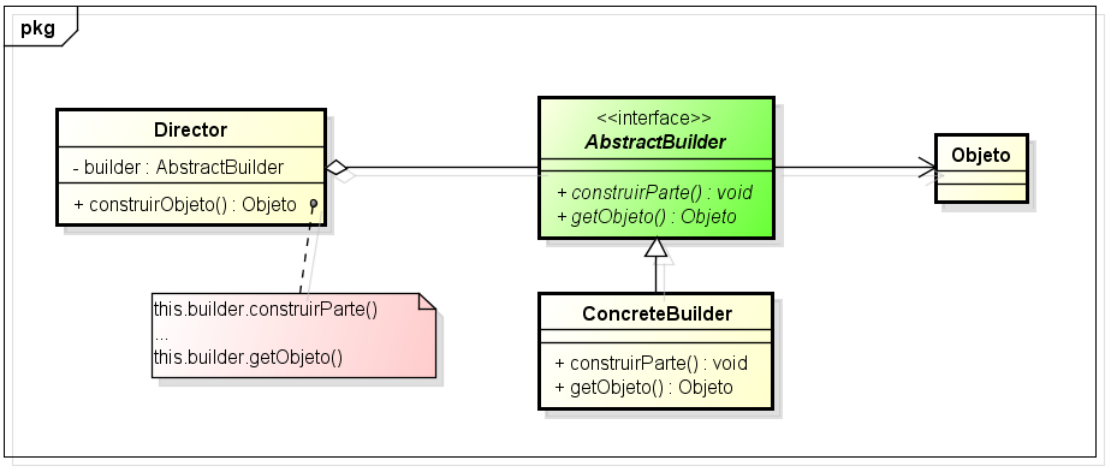


Figura 11: Patrón *Builder*

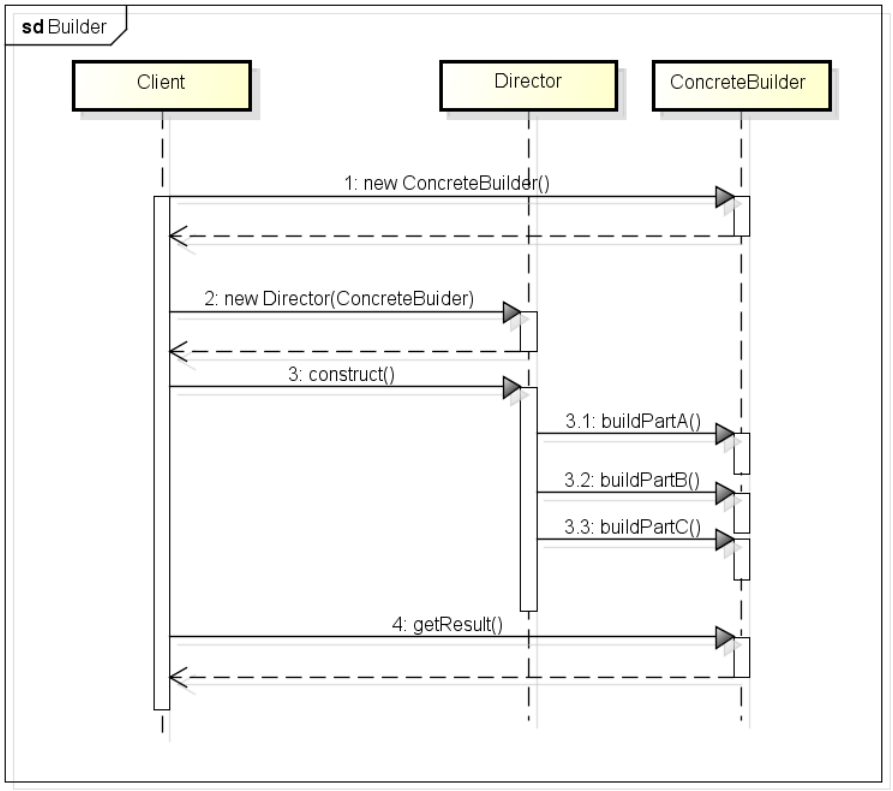


Figura 12: Diagrama de secuencia del patrón *Builder*

15 Estructurales

15.1 Composite

Allows you to compose objects into tree structures to represent part-whole hierarchies. Lets clients treat individual objects and composition of objects uniformly.

Contexto objetos complejos

Problema ¿cómo representar jerarquías parte-todo? ¿cómo hacer que los clientes ignoren las diferencias entre objetos compuestos y objetos individuales?

Solución composición recursiva de objetos similares.

Ventajas	Desventajas
Es fácil añadir nuevos tipos de Componentes	Hay que imponer restricciones en ciertas composiciones

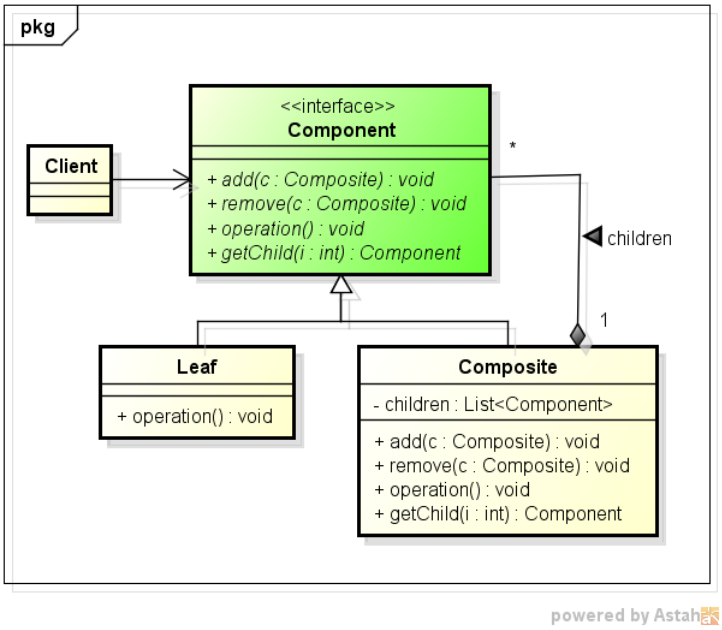


Figura 13: Patrón *Composite*

15.2 Adapter

An object takes an interface and adapts it to one that a client is expecting.

Contexto: Código viejo que no se puede modificar, que implementa una interfaz que nuestro código no conoce.

Problema: ¿Cómo adaptar nuestro código sin modificarlo, solo extenderlo?

Solución: Crear un Adaptador, que convierte la interfaz de una clase en otra interfaz (la que espera nuestro código).

Ventajas	Desventajas
Bajo acoplamiento entre el cliente y el sistema externo	No se puede adaptar una clase y todas sus subclases

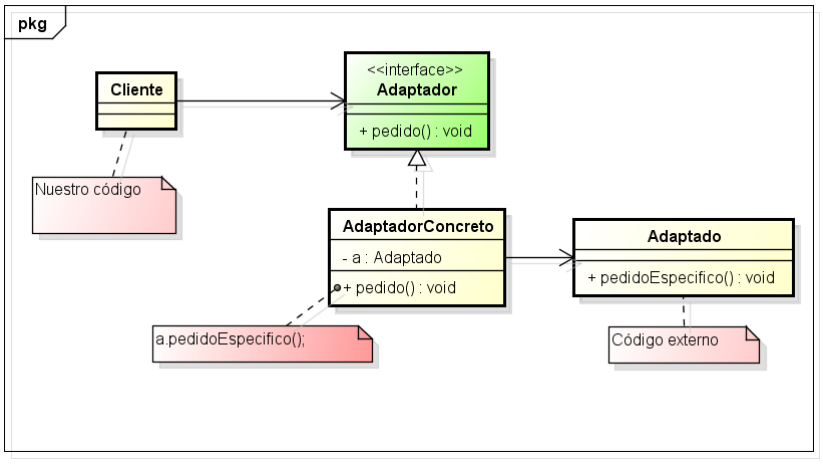


Figura 14: Patrón *Adapter*

15.3 Decorator

Use composition to give objects new responsibilities at runtime without making any code changes.

Contexto se quiere agregar nuevas responsabilidades a una clase.

Problema ¿cómo agregar nuevo comportamiento a una clase sin modificarla?

Solución crear una clase Decorador que contiene a un Decorado. Los Decoradores son *wrappers*. Son del mismo tipo que los objetos que decoran, pero añaden nuevo comportamiento.

Ventajas	Desventajas
Se evita la herencia, que puede resultar inmanejable	Muchas clases pequeñas
	Hay problemas cuando el código depende de clases específicas
	Mucha complejidad para instanciar un Componente

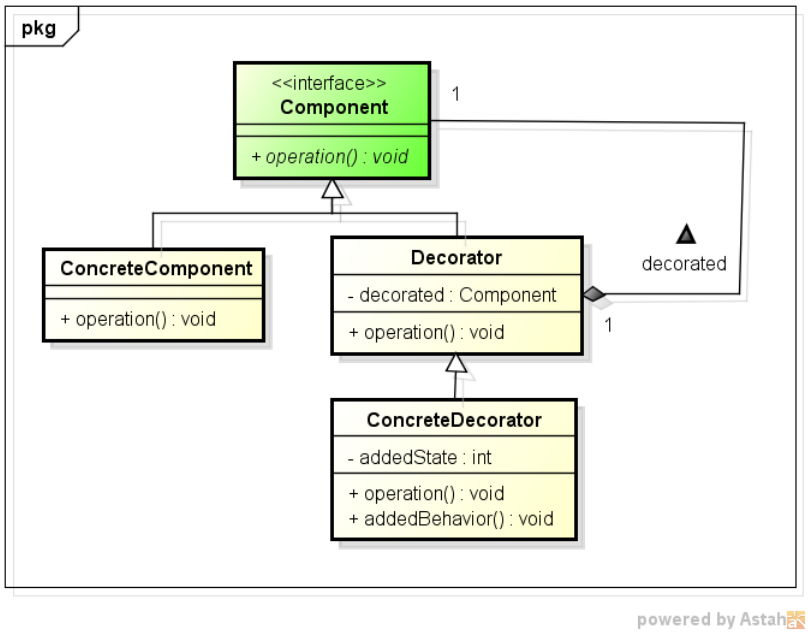


Figura 15: Patrón *Decorator*

```
public class SteamedMilk : CondimentDecorator
{
    Beverage beverage;

    1 reference
    public SteamedMilk(Beverage beverage)
    {
        this.beverage = beverage;
    }

    17 references
    public override string GetDescription()
    {
        return beverage.GetDescription() + ", Steamed Milk";
    }

    19 references
    public override double Cost()
    {
        return .10 + beverage.Cost();
    }
}
```

Figura 16: Ejemplo del patrón *Decorator*

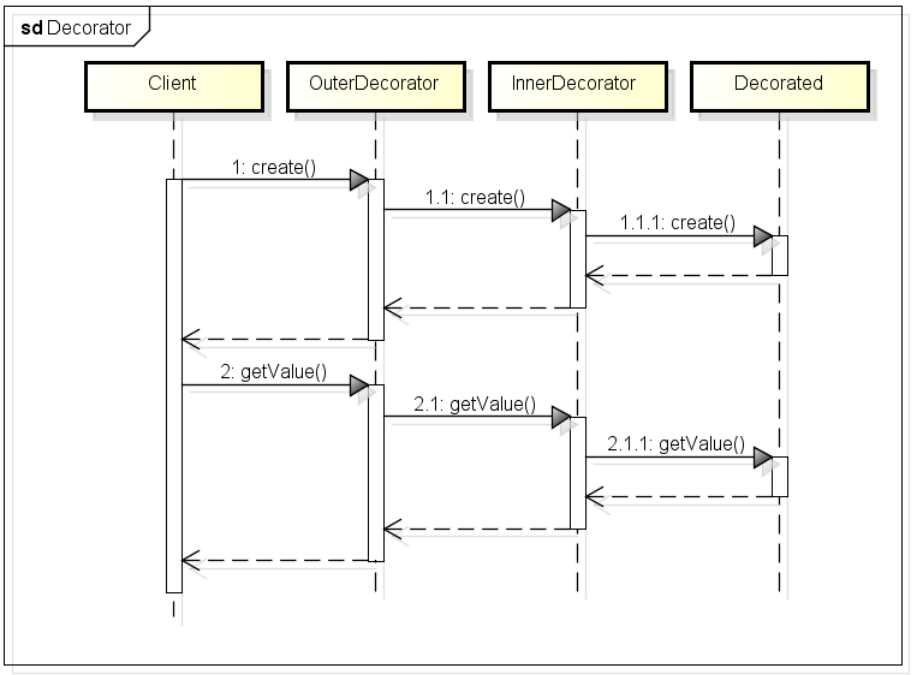


Figura 17: Diagrama de secuencia del patrón *Decorator*

15.4 Facade

Alters an interface to simplify it.

Contexto Un subsistema con una interfaz muy complicada.

Problema ¿cómo simplificar el acceso al subsistema en cuestión?

Solución Una clase que ordena el acceso al subsistema.

Ventajas	Desventajas
Oculta a los clientes del subsistema	
Bajo acoplamiento entre el subsistema y los clientes	
No impide que las aplicaciones utilicen el subsistema en caso de ser necesario	

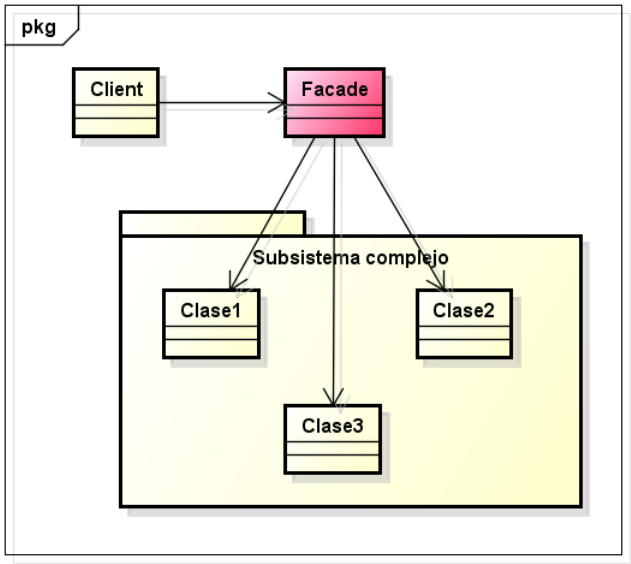


Figura 18: Patrón *Facade*

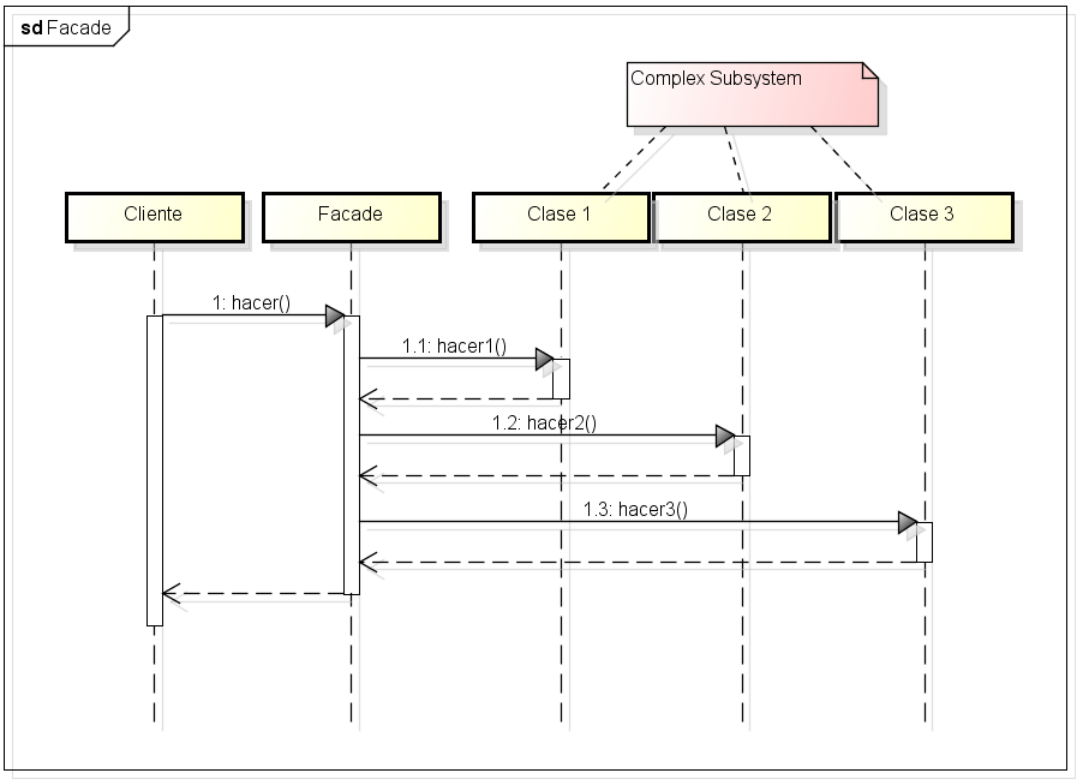


Figura 19: Diagrama de secuencia del patrón *Facade*

15.5 Proxy

Proxies control and manage access to an object.

Contexto un objeto cuyo acceso necesita ser controlado.

Solución Proxy controla el acceso a un objeto que puede ser remoto, caro para construir, o que necesita seguridad. El Proxy contiene al objeto en cuestión, y a veces puede incluso crearlo.

Ejemplo el manejo de punteros con *smart pointers*.

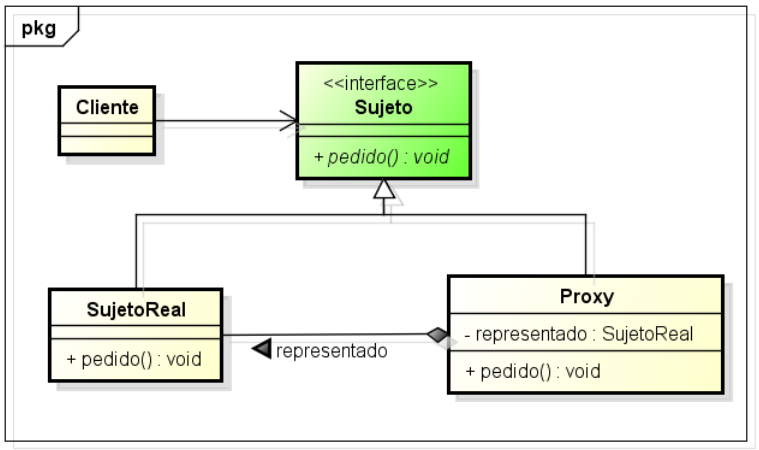


Figura 20: Patrón *Proxy*

15.6 Bridge

How to vary the implementation AND the abstraction by placing the two in separate class hierarchies. Concrete subclasses are implemented in terms of the abstraction.

Contexto muchas implementaciones para resolver un mismo problema.

Problema ¿cómo variar la implementación y la abstracción?

Solución descomponer la interfaz y la implementación en dos jerarquías ortogonales. La clase interfaz contiene un puntero a la clase abstracta de la implementación. Esta composición es el *bridge*. Todos los métodos de la abstracción se programan contra la implementación.

Ejemplo una librería gráfica. Un "botón" es una abstracción, y un "botón en Windows" es una implementación.

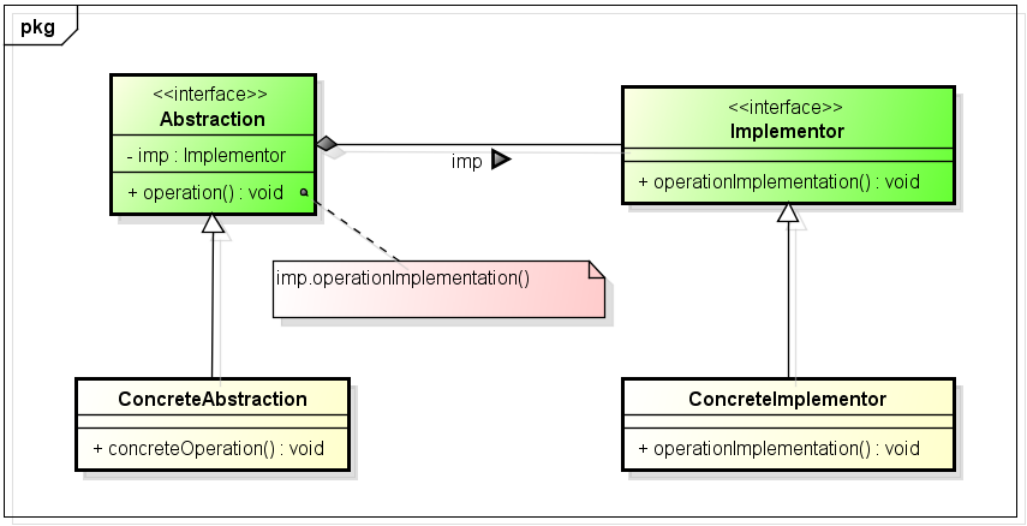


Figura 21: Patrón *Bridge*

16 Comportamiento

16.1 Template Method

The Template Method defines the steps of an algorithm and allows subclasses to provide the implementation for one or more steps.

Contexto FALTA

Problema FALTA

Solución una clase abstracta provee la implementación de un algoritmo, y las subclases implementan partes de ese algoritmo. La clase abstracta también puede proporcionar *hook methods*: métodos opcionales.

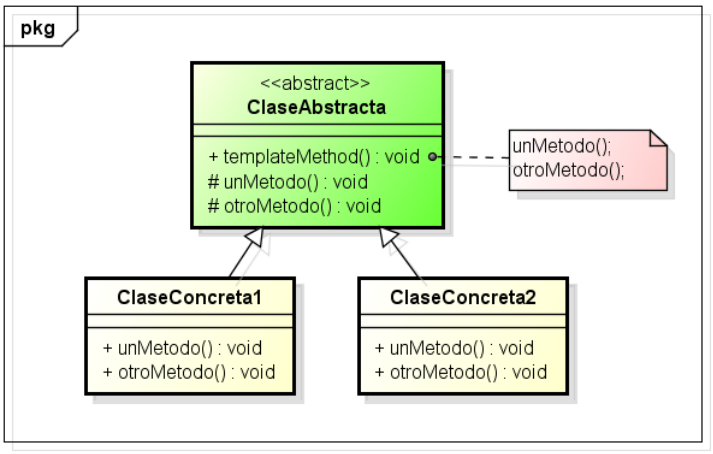


Figura 22: Patrón *Template Method*

16.2 Command

Encapsulate method invocation as an object.

Problema ¿cómo enviar un mensaje a un objeto sin conocer el objeto receptor?

Solución Encapsular un pedido como un objeto. Permite:

- Parametrizar dinámicamente los Comandos
- Comandos macro (i.e. con más de un Comando)
- Sistemas de logging y transaccionales

Ventajas	Desventajas
Bajo acoplamiento entre el objeto que realiza el pedido y el que lo lleva a cabo	

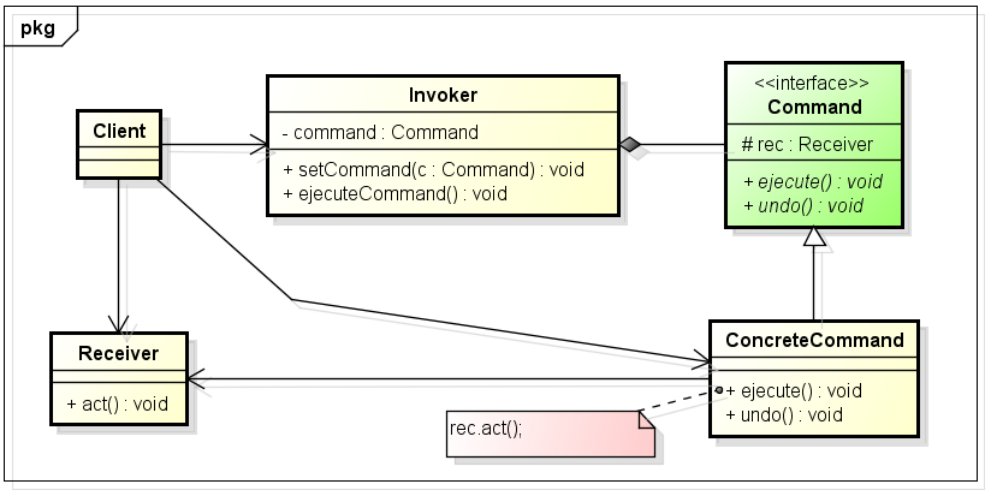


Figura 23: Patrón *Command*

```
[TestMethod]
0 references
public void TestTurningSimpleOn()//Command Pattern Client
{
    //Command Pattern Invoker
    SimpleRemoteControl remote = new SimpleRemoteControl();

    //Command Pattern Receivers
    Light light = new Light("Kitchen");
    GarageDoor garageDoor = new GarageDoor("");

    //Commands for the receivers
    LightOnCommand lightOn = new LightOnCommand(light);
    GarageDoorUpCommand garageDoorOpen = new GarageDoorUpCommand(garageDoor);

    //Passing the light on command to the invoker
    remote.SetCommand(lightOn);
    //Simulate the button being pressed on the invoker
    Assert.AreEqual("Kitchen light is on", remote.ButtonWasPressed());

    //Passing the garage door open command to the invoker
    remote.SetCommand(garageDoorOpen);
    //Simulate the button being pressed on the invoker
    Assert.AreEqual("Garage door is up", remote.ButtonWasPressed());
}
```

Figura 24: Ejemplo de patrón *Command*

16.3 Observer

Establishes a 1-to-many dependency between objects, so that when one obect changes state, all of its dependants are notified automatically.

Contexto se tiene un objeto cuyo estado varía. Se requiere que otros objetos sean notificados de los cambios de estado.

Problema ¿cómo desacoplar los Observadores del Observado?

Solución existen dos modelos:

- *Pull*: los Observadores solicitan al Observado su estado. `notifyObservers() + getState()`
- *Push*: el Observado le proporciona su estado a los Observadores. `notifyObservers(Object o)`

Ventajas	Desventajas
Bajo acoplamiento entre Observador y Observado	
La cantidad y el tipo de los Observadores puede variar en tiempo de ejecución	

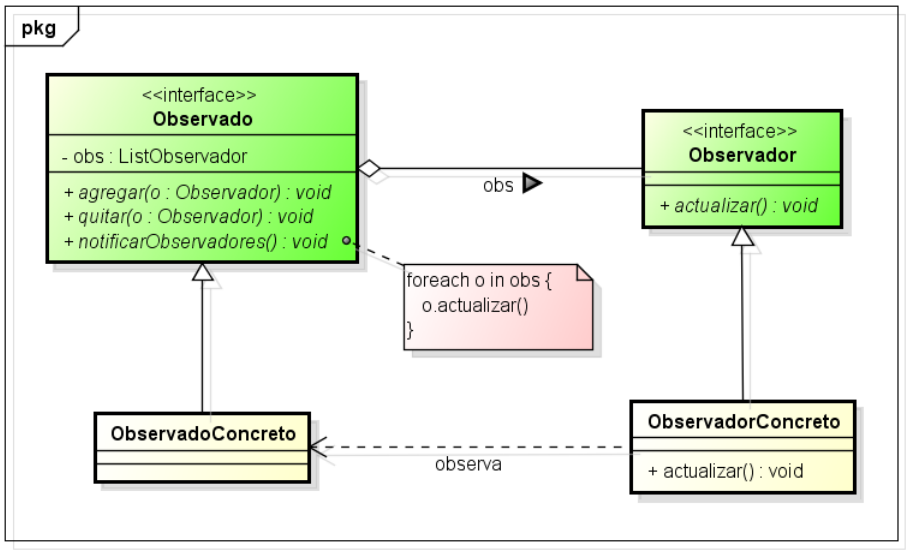


Figura 25: Patrón *Observer*

16.4 Iterator

Provides a way to access the elements of a collection, without exposing its underlying representation.

Contexto se necesita acceder al contenido de una Colección.

Problema ¿cómo acceder a la Colección sin exponer su implementación interna?

Solución una clase Iterador que sabe cómo recorrer los objetos de la Colección. El Iterador no sabe el orden de los objetos. El Iterador se lo suele crear con *Factory Method*.

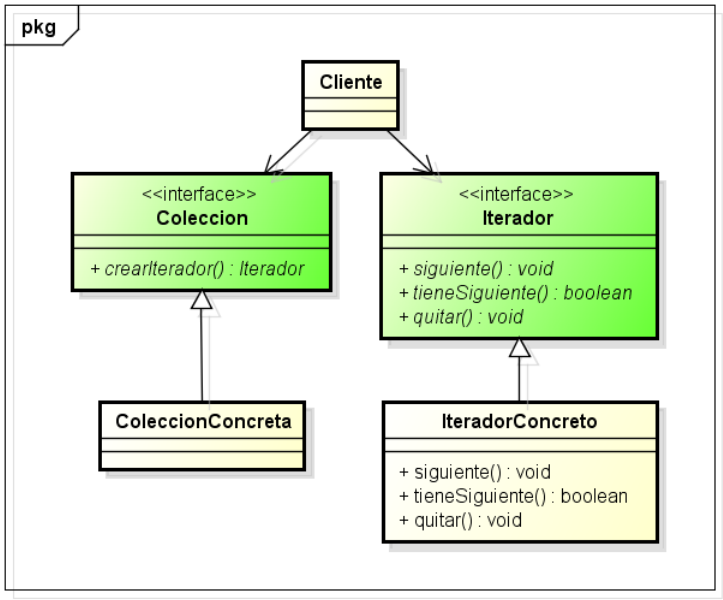


Figura 26: Patrón *Iterator*

16.5 State

Put each state's behavior in its own class. Allows an object to alter its behavior when its internal state changes. The object will appear to change its class.

Contexto el comportamiento de un objeto depende de su estado.

Problema ¿cómo eliminar las sentencias CASE que deciden el comportamiento en base al estado del objeto?

Solución el objeto mantiene una referencia a su estado actual. Todos los estados deben poder manejar todas las transiciones posibles, y ello puede implicar cambiarle el estado al Contexto. Es decir, las transiciones de estado las puede manejar el Contexto o cada Estado.

Cada nodo del diagrama de estado es una clase que implementa la interfaz Estado. La interfaz Estado define como métodos todos las transiciones de Estado posibles.

Ventajas	Desventajas
Agregar un Estado solo implica agregar una clase	Muchas clases
	Puede haber código repetido en los Estados

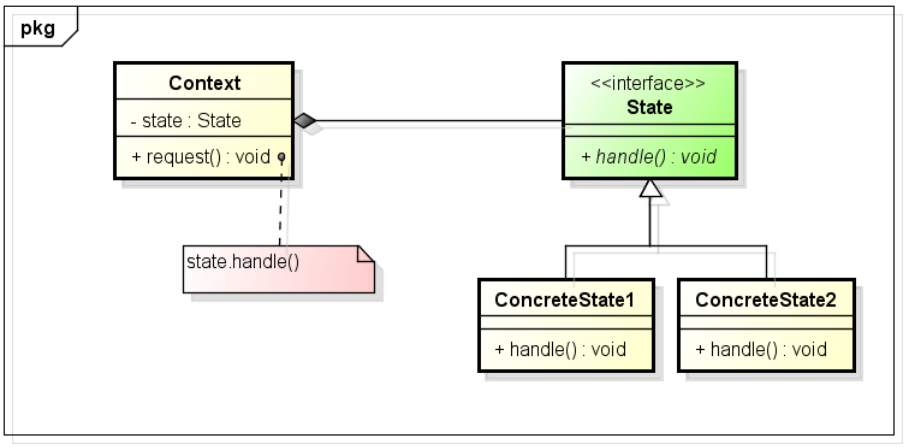


Figura 27: Patrón *State*

```
public class GumballMachine
{

    State soldOutState;
    State noQuarterState;
    State hasQuarterState;
    State soldState;
    State currentState;

    int count = 0;

    1 reference | 1/1 passing
    public GumballMachine(int numberOfGumballs)
    {
        soldOutState = new SoldOutState(this);
        noQuarterState = new NoQuarterState(this);
        hasQuarterState = new HasQuarterState(this);
        soldState = new SoldState(this);
        this.count = numberOfGumballs;
        if (numberOfGumballs > 0)
        {
            currentState = noQuarterState;
        }
        else
        {
            currentState = soldOutState;
        }
    }

    3 references | 1/1 passing
    public string InsertQuarter()
    {
        return currentState.InsertQuarter();
    }
```

Figura 28: Ejemplo de patrón *State*

16.6 Strategy

Defines a family of algorithms, encapsulates each one and makes them interchangeable.

Contexto una clase con muchos comportamientos que aparece con sentencias CASE en sus métodos.

Problema ¿cómo cambiar el comportamiento de un objeto?

Solución utilizar delegación y composición para cambiar el comportamiento en tiempo de ejecución.

Ventajas	Desventajas
Nos ahorra tener que utilizar herencia para obtener nuevo comportamiento	
Permite cambiar comportamiento en tiempo de ejecución	

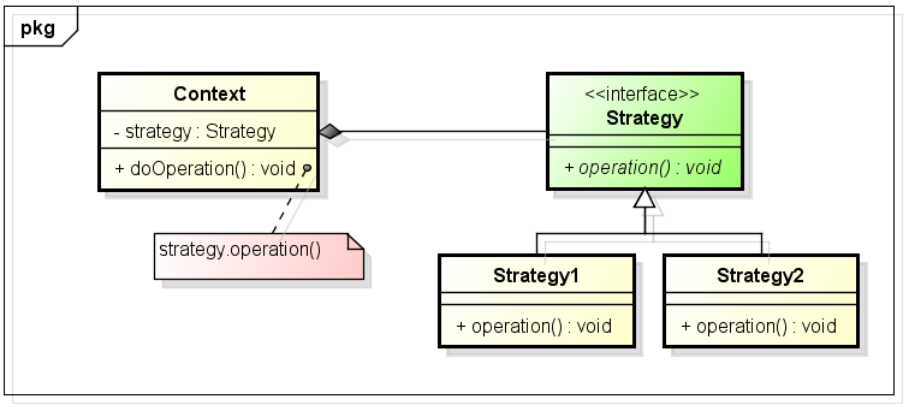


Figura 29: Patrón *Strategy*

```
public abstract class Duck
{
    protected IFlyBehavior flyBehavior;
    protected IQuackBehavior quackBehavior;

    3 references
    public abstract object Display();

    4 references
    public object PerformFly()
    {
        return FlyBehavior.Fly();
    }

    4 references
    public object PerformQuack()
    {
        return QuackBehavior.Quacking();
    }

    0 references
    public string Swim()
    {
        return "All ducks float, even decoys!";
    }
```

Figura 30: Ejemplo del patrón *Strategy*

16.7 Mediator

Contexto las interdependencias entre objetos son poco estructuradas y difíciles de comprender.

Problema ¿cómo reducir el acoplamiento entre objetos?

Solución Introducir un objeto que encapsula cómo interactúan una serie de objetos. El mediador añade funcionalidad combinando funcionalidades ya existentes.

Ventajas	Desventajas
Desacopla a los objetos	El Mediador puede ser difícil de mantener
Centraliza el control	

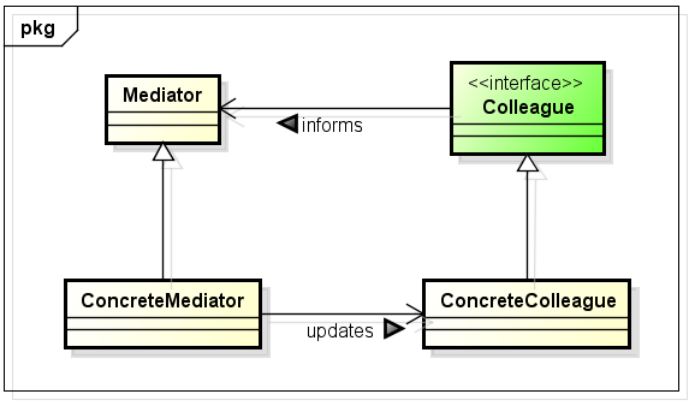


Figura 31: Patrón *Mediator*

16.8 Visitor

Contexto se necesita agregar nueva funcionalidad a un objeto compuesto

Problema ¿cómo agregar nueva funcionalidad sin modificar el código?

Solución Una clase **Visitor** visita cada elemento del objeto compuesto y devuelve su estado.

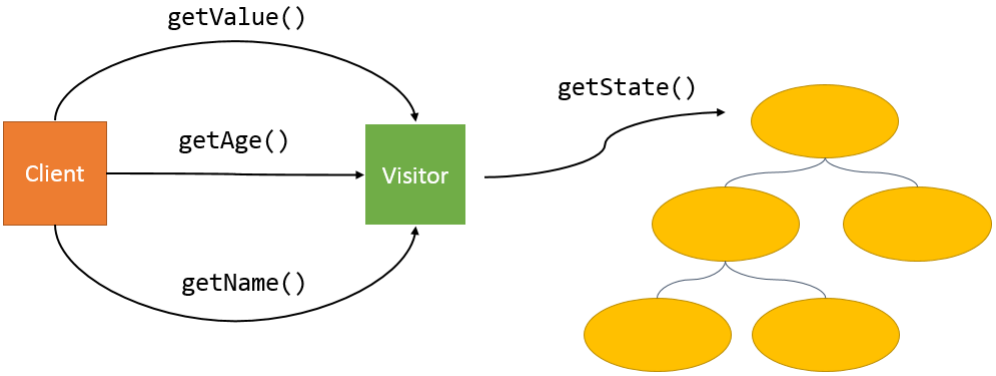


Figura 32: Patrón *Visitor*

Ventajas	Desventajas
	Se pierde la encapsulación del objeto
	Hay que extender el objeto visitado una vez

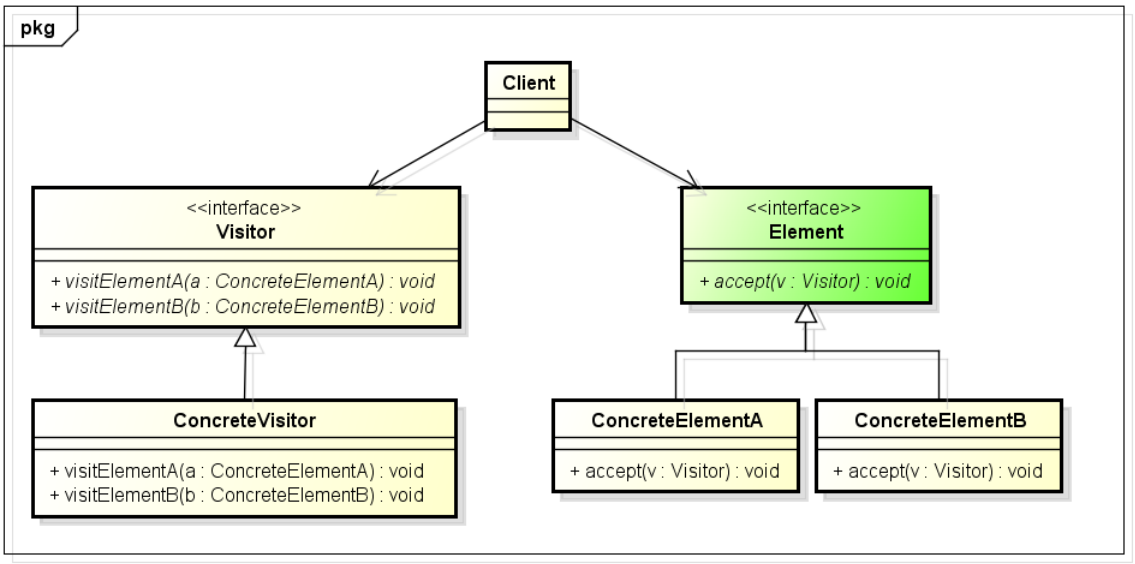


Figura 33: Patrón *Visitor*

16.9 Chain of Responsibility

Contexto hay un número variable de objetos que pueden procesar un pedido, y el manejador no se conoce en tiempo de compilación.

Problema ¿cómo darle la posibilidad a más de un objeto de procesar un pedido?

Solución encadenar los objetos que pueden procesar el pedido y pasar el mismo hasta que un objeto lo procese.

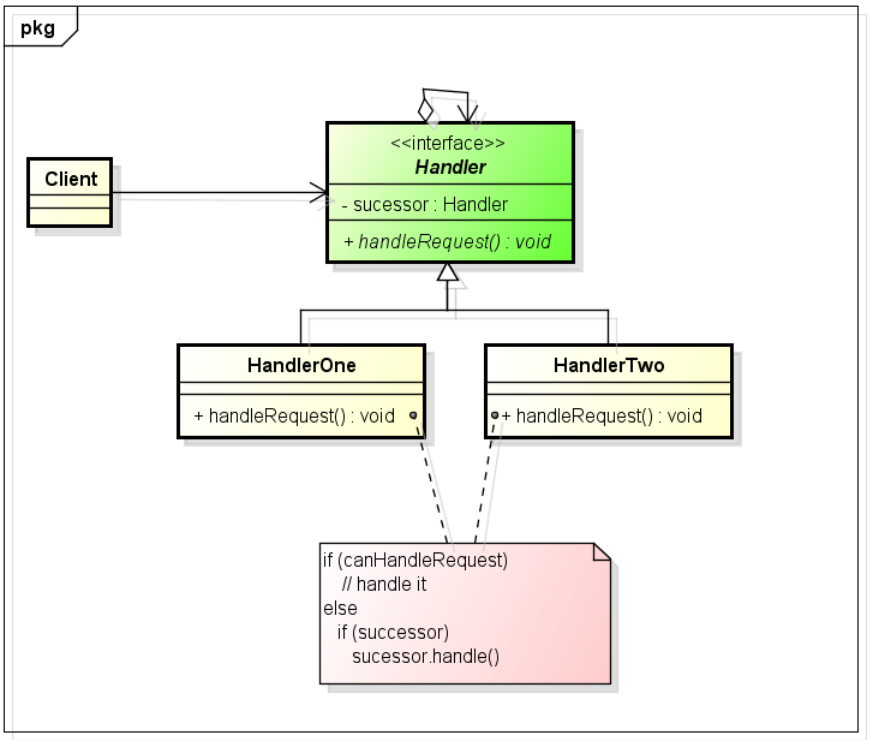


Figura 34: Patrón *Chain of responsibility*

Parte III
Patrones de Arquitectura

Problema	Patrones	Solución
Cimientos	*Layers, Pipes and Filters*	Evitar un mar de objetos
Sistemas distribuidos	*Broker*	
Sistemas interactivos	*MVC*	Permitir interacción con personas
Sistemas adaptables	*Microkernel, Reflection*	Permitir la extensión de aplicaciones

17 Layers

Contexto: sistema grande que requiere ser descompuesto en partes más pequeñas.

Problema: ¿cómo particionar la funcionalidad de un sistema para que

a. la funcionalidad a diferentes niveles de abstracción esté desacoplado?

b. la funcionalidad a un nivel particular de abstracción pueda evolucionar fácilmente?

Solución: formar al sistema como un conjunto de capas jerárquicas. Cada capa usa servicios de la capa inferior y ofrece sus servicios a la capa inmediatamente superior. Cada capa tiene una responsabilidad específica y los componentes tienen el mismo nivel de abstracción. Las capas base deberían ser más chicas.

Aplicabilidad: los cambios deben estar confinados a una capa. Promueve el uso de interfaces estándares y estables. Las capas deberían poder ser intercambiables. Las responsabilidades están agrupadas.

Ejemplo: protocolos de red.

Ventajas	Desventajas
Las capas (deberían poder) ser reutilizadas	Un mal diseño puede hacer que un cambio en una capa afecte a otras
Soporte para la estandarización	Poca eficiencia si hay muchas capas
Las dependencias se conservan localmente	Dificultad para establecer correcta cantidad de capas

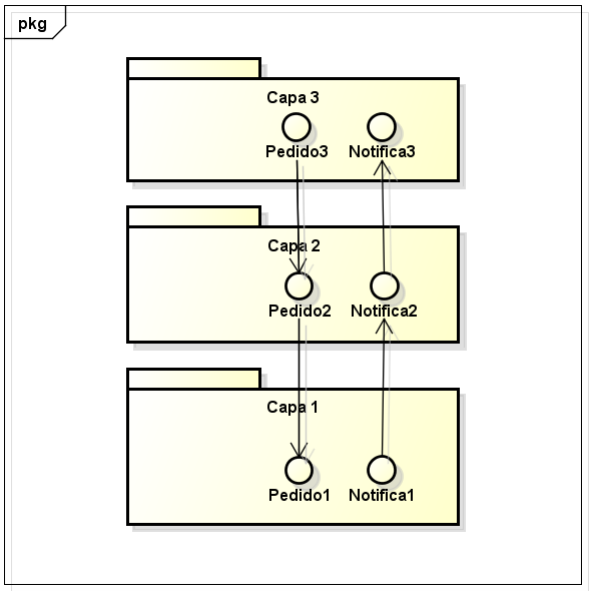


Figura 35: Patrón Layer

18 MVC (*Model View Controller*)

Contexto: sistemas interactivos.

Problema: se requieren cambios de interfaces simples y rápidos, que no dependan del modelo de datos.

Solución: dividir la aplicación en tres partes desacopladas:

Modelo representa las entidades de la aplicación.

Utiliza el patrón *Observer* para mantener informados a la Vista y al Controlador. Puede utilizar el patrón *Adapter* para adaptarse a controladores y vistas.

Vista(s) presentan datos al usuario.

Utiliza el patrón *Strategy* para cambiar su controlador. También utiliza el patrón *Composite* para manejar los componentes de la vista (ventanas, botones, etc.).

Controlador(es) se asocian con las vistas y permiten manipularlas. También manipula el modelo.

Aplicabilidad: la misma información puede ser representada de muchas formas. La vista debe reflejar cambios en el modelo subyacente. La vista debe poder cambiarse fácilmente sin afectar al modelo.

Ventajas	Desventajas
Muchas vistas de un modelo	Incremento de complejidad
Portabilidad del modelo	Conexión entre la vista y el controlador
	Vista y controlador dependen de la plataforma

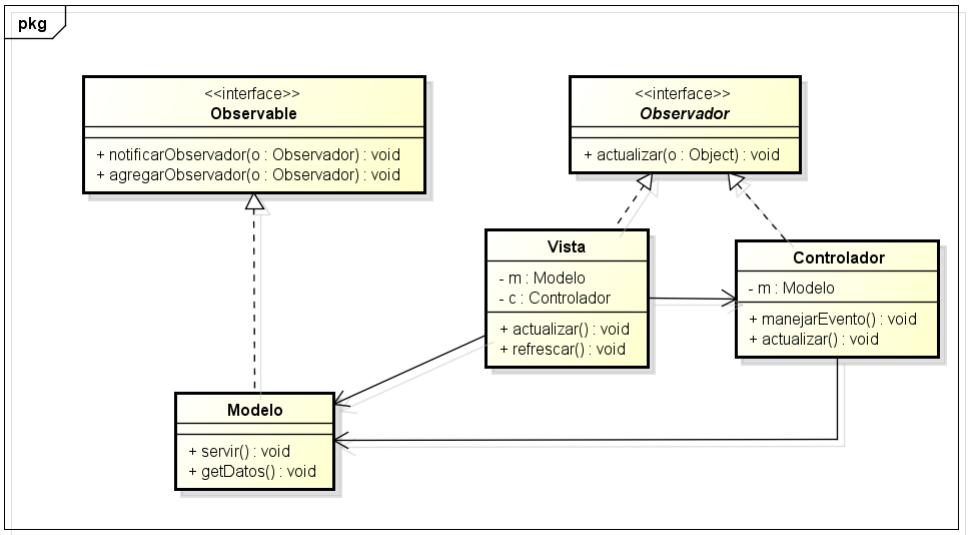
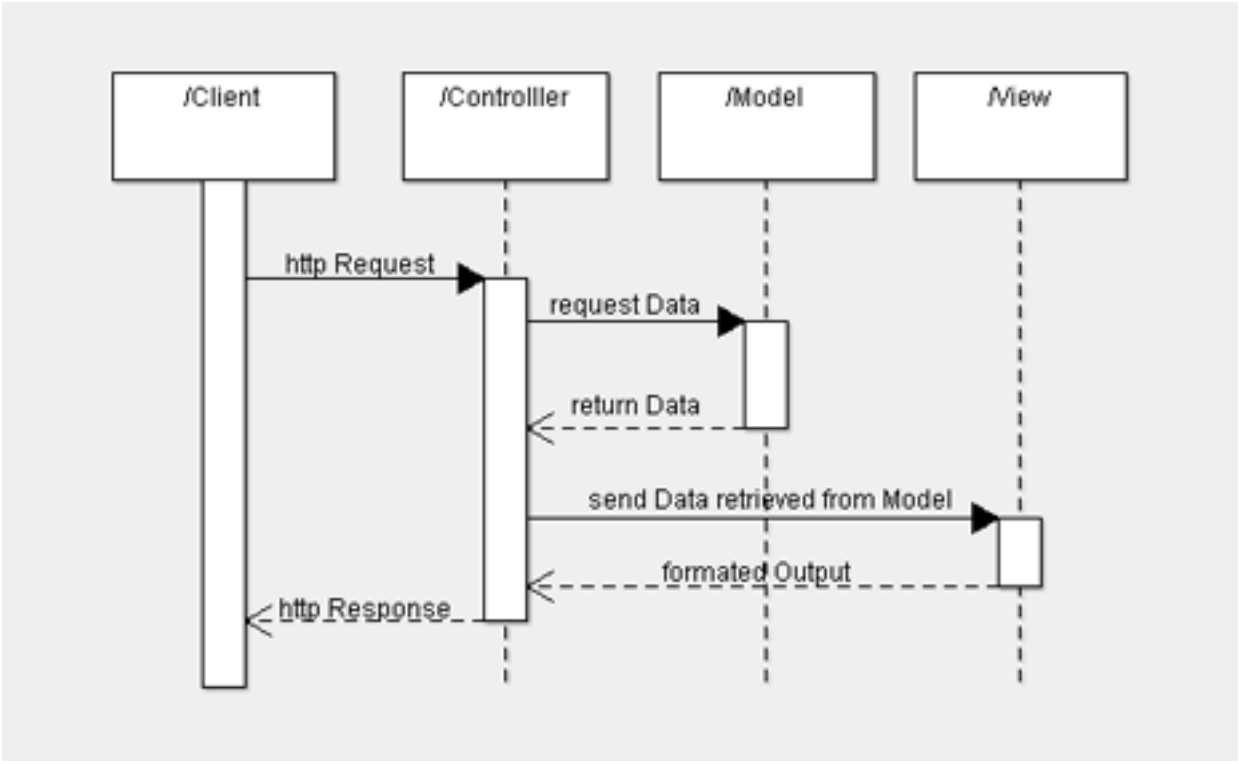
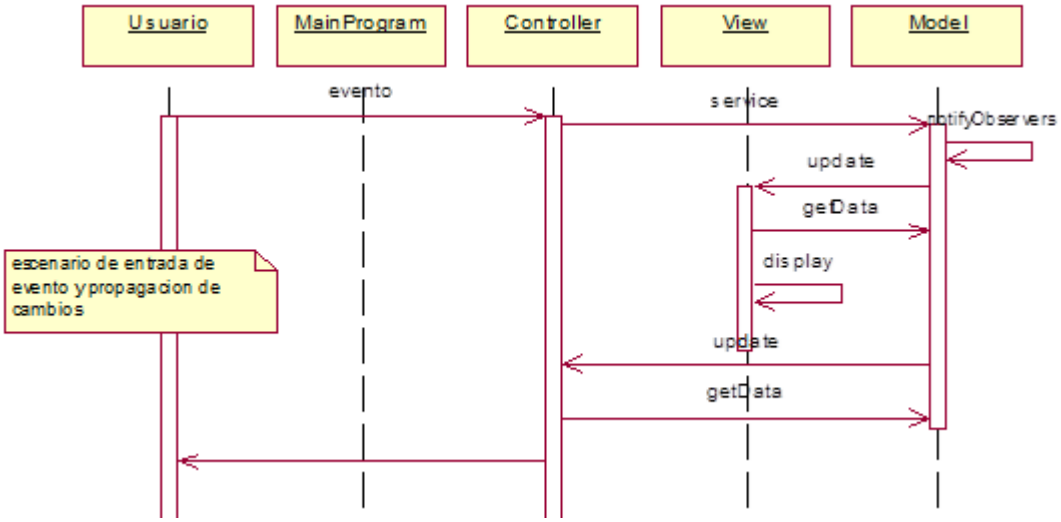


Figura 36: Patrón MVC



(a) MVC aplicado a la Web



(b) MVC genérico. El controlador no conoce a la Vista

Figura 37: Diagrama de secuencia del patrón MVC

19 *Broker*

Contexto: objetos en un sistema distribuido que interactúan de forma sincrónica o asíncrona.

Problema: se requiere que la comunicación entre objetos del sistema sea independiente de la localización.

Solución: separar la funcionalidad del sistema de la funcionalidad de la comunicación. Los clientes y servidores proveen funcionalidad en cualquier nodo de la red. Los *brokers* son mediadores entre clientes y servidores. Los *proxies cliente / servidor* protegen a los clientes y a los servidores de los problemas en la red.

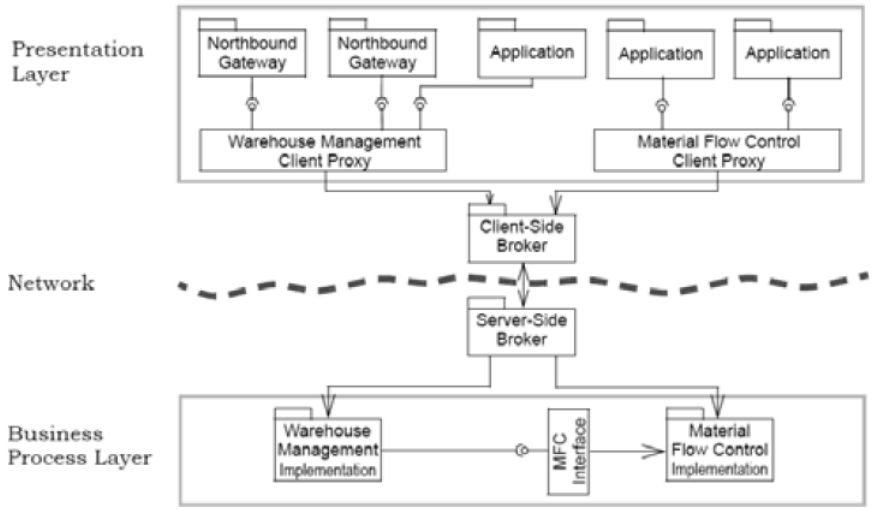


Figura 38: Patrón Broker

20 Microkernel

Contexto: sistemas complejos que deben poder adaptarse a cambios de requisitos.

Problema: desarrollar una familia de aplicaciones similares que usan interfaces parecidas para acceder a una funcionalidad básica común.

Solución: un objeto ***microkernel*** (pequeño, consume pocos recursos) que ofrece servicios básicos, servicios de comunicación entre procesos y encapsula partes dependientes del hardware. Los servidores internos extienden la funcionalidad del microkernel. Los servidores externos proveen funcionalidad más compleja.

Ejemplo: Symbian OS tiene un microkernel que ofrece servicios de *scheduling*, manejo de memoria y drivers de dispositivos.

Aplicabilidad:

Ventajas	Desventajas
Permite la portabilidad (solo se necesita cambiar el microkernel)	Menor eficiencia frente a un sistema monolítico
Alta flexibilidad y extensibilidad	Es difícil delimitar las tareas del microkernel

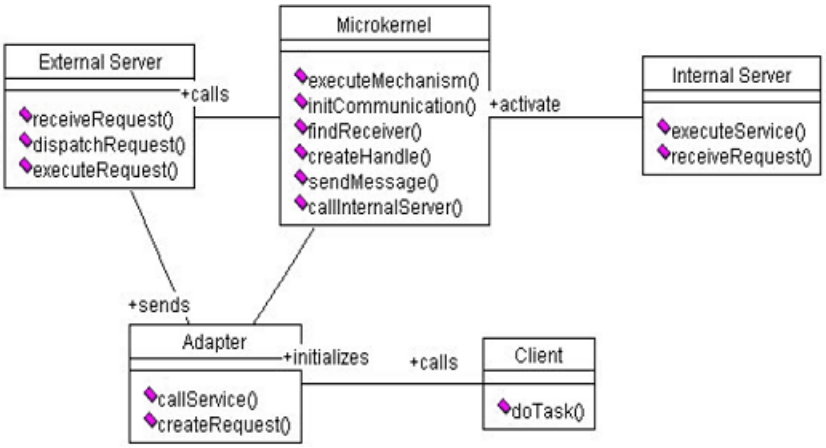


Figura 39: Patrón microkernel.

21 *Pipes and Filters*

Contexto: procesamiento de *streams* de datos.

Solución: cada paso de procesamiento se encapsula en un filtro. Los datos pasan por *pipes* entre filtros adyacentes.

Ventajas	Desventajas
Flexibilidad para intercambiar filtros	Poca eficiencia si un filtro necesita toda la información para comenzar a procesar
Reuso de filtros	Poca eficiencia si hay que transformar muchos datos
	Manejo de errores es complicado

Parte IV
Patrones de Aplicaciones Enterprise

Características de una aplicación *entreprise*:

- concurrencia - multiusuarios
- persisten datos de forma masiva
- lógica de negocio
- muchas interfaces de usuarios
- integración con otros sistemas

Ejemplos de aplicaciones *entreprise*:

- sistema contable
- home banking

Layers (capa lógica, ej: mvc)$\neq$ tiers (capa física, ej: cliente-servidor)
Vistas 4 + 1:

- Lógica
- Componentes
- Despliegue
- Procesos

Capas:

- Presentación
- Servicio
- Dominio
- Persistencia
- *Data source*

Aspectos a tener en cuenta:

- Seguridad: autenticación y autorización
- Transacciones
- Excepciones
- Concurrencia
- Internacionalización
- Accesibilidad
- Usabilidad

Patrones de lógica de dominio:

- *Transaction script*: hay un solo procedimiento para cada acción que requiere el usuario. Podría traer repetición de código y de lógica
- *Domain model*: las responsabilidades se distribuyen a cada objeto de dominio
- *Table module*: una clase maneja todas las instancias

- *Service layer*: brinda un único punto de acceso al dominio

Patrones de *data source* (persistencia):

- *Row data gateway*: dada una query a una base de datos, tiene una instancia del gateway por cada fila retornada
- *Table data gateway*: dada una query a una base de datos, tiene una instancia del gateway por cada tabla
- *Active record*: un objeto **de dominio** sabe cómo interactuar con la base de datos
- *Data mapper*: separa un objeto de dominio de una base de datos

Patrones de comportamiento DB:

- *Unit of work*: mantiene el seguimiento de todo lo que se hace durante la transacción de negocio que afecta a la base de datos. Al final realiza todos los cambios contra la base de datos (*commit*).
- *Identity map*: se asegura que un objeto de dominio se carga una sola vez en cada transacción (caché).
- *Lazy load*: un objeto que no contiene todos los datos necesarios, pero que sabe cómo obtenerlos.

Patrones de estructura DB:

- *Identity field*: permite identificar un objeto de dominio persistido en una base de datos.

Parte V
Artículos

22 *Architectural Blueprints - The 4+1 View Model of Software Architecture*

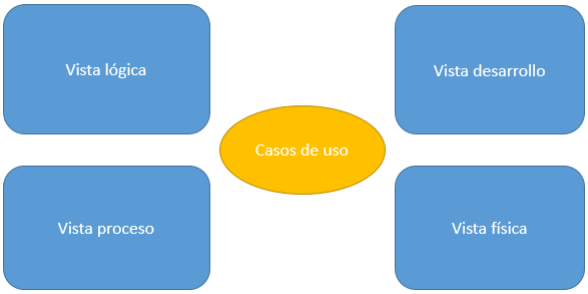


Figura 40: Vistas 4+1

Vista	**Lógico**	**Proceso**	**Desarrollo**	**Físico**	**Escenarios**
Componentes	Clase	Tarea	Módulo, subsistema	Nodo	Scripts
Conectores	Asociación, herencia...	Message broadcast, RCP...	Compilación, dependendencia	Medio de comunicación, LAN, WAN...	
Contenedores	Categoría de clase	Proceso	Subsistema	Subsistema físico	Web
Stakeholders	Usuario final	Diseñador de sistema, integrador	Desarrollador, gerente	Diseñador de sistema	Usuario final, desarrollador
Preocupación	Funcionalidad	Performance, disponibilidad, tolerancia a fallos, integridad	Organización, reuso, portabilidad	Escalabilidad, performance, desarrollo	

Cuadro 4: Resumen del modelo "4+1"

23 Java: method overloading vs. method overriding

Overloading *writing two or more methods in one class that satisfy one or both of these conditions:*

1. *The number of parameters is different.*
2. *The parameter types are different.*

The following actions do not clasify as overloading and will result in compiler errors:

1. *Just changing the return type of the methods.*
2. *Just changing the name of the methods' parameters (but not their types).*

Overriding *a method inherited from a parent class will be changed. Everything remains exactly the same except the method definition – basically what the method does is changed slightly to fit in with the needs of the child class.*

Method overriding is a run-time phenomenon that is the driving force behind polymorphism.

Algoritmo 7 Example of method overriding

```
public class Parent
{
  public string method()
  {
    return "I'm␣the␣parent";
  }
}

public class Child extends Parent
{
  public string method()
  {
    return "I'm␣the␣child";
  }
}
```

24 Java: methods common to all `Objects`

Value class *class that represents a value, e.g.* `Integer` *or* `Date`*.*

Mother class `Object` *has nonfinal methods designed to be overridden under certain conditions*

- `equals`*:*

When to override it	*When* not *to override it*
The class has logical equality that differs from object identity, and a superclass has not already overriden this method	*Each class instance is unique. Example: a Thread*
Most value classes	*You don't care whether the class provides a "logical equality" test*
	A superclass has already overriden equals
	You are certain that the method will never be invoked
	The class uses instance control to ensure that at most one object exists with each value

Contract of the `equals` *method, given x, y and z non-null reference values:*

- ***Reflexive:*** `x.equals(x)` *should return* `true`
- ***Symmetric:*** `x.equals(y)` *should be equal to* `y.equals(x)`
- ***Transitive:*** *if* `x.equals(y)` *is true and* `y.equals(z)` *is true, then* `x.equals(z)` *should return* `true`
- ***Consistent:*** *if no information used in the* `equals` *comparison on the objects is modified,* `x.equals(y)` *should consistently return* `true` *or consistently return* `false`
- ***Non-nullity:*** `x.equals(null)` *should always return* `false`

Problem of equivalence relations in OO languages: there is no way to extend an instantiable class and add a value component while preserving the `equals` *contract.*

Workaround: favor composition over inheritance.

Tips:

1. `Equals` *methods should perform deterministic computations on memory-resident objects.*
2. *Always override* `hashCode` *when you override equals*

- `hashCode`

Contract of the `hashCode` *method:*

- *When invoked on the same object more than once, it should consistently return the same integer, provided no information used in* `equals` *has changed.*

- *Equal objects must have equal hash codes.*

A good hash function tends to produce unequal hash codes for unequal objects.

- *toString*
- *clone*
- *finalize*