SOLID:
In software programming, **SOLID** is a mnemonic acronym for five design principles intended to make object-oriented designs more understandable, flexible, and maintainable. Although the SOLID principles apply to any object-oriented design, they can also form a core philosophy for methodologies such as agile development or adaptive software development.

Principles[edit]

- Single-responsibility principle: "There should never be more than one reason for a class to change." In other words, every class should have only one responsibility.
- Open–closed principle: "Software entities ... should be open for extension, but closed for modification."
- Liskov substitution principle: "Functions that use pointers or references to base classes must be able to use objects of derived classes without knowing it."
- Interface segregation principle: "Clients should not be forced to depend upon interfaces that they do not use."
- Dependency inversion principle: "Depend upon abstractions, [not] concretes."

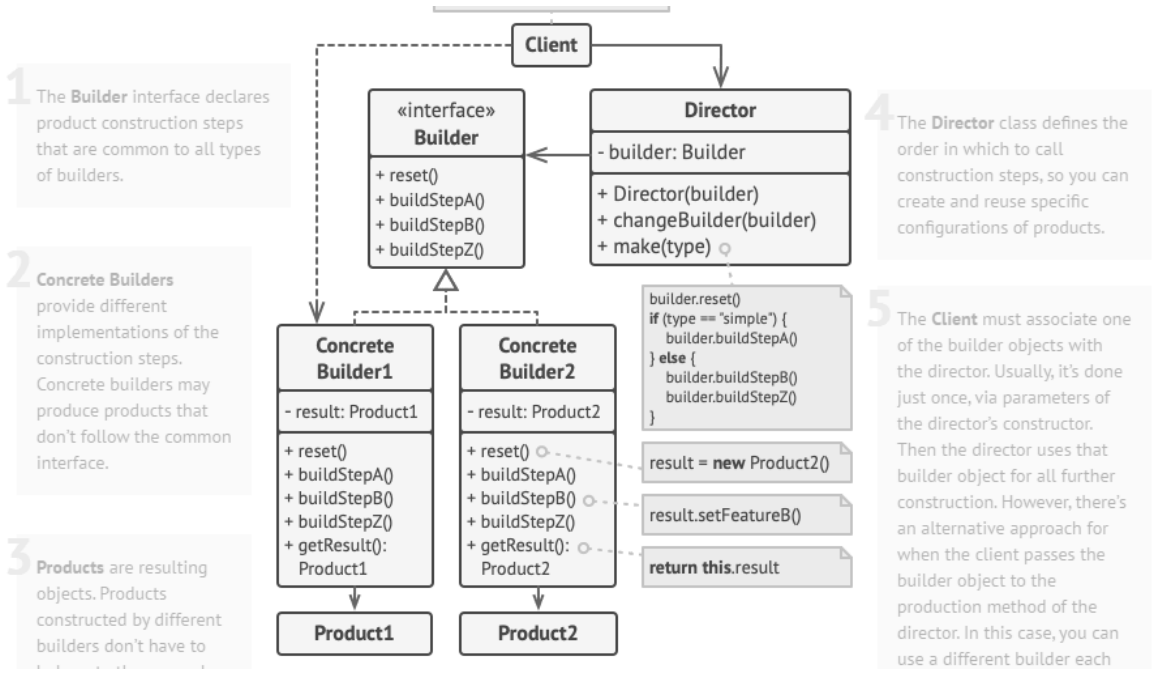
Creational Patterns:
Creational design patterns provide various object creation mechanisms, which increase flexibility and reuse of existing code.

Builder:
lets you construct complex objects step by step. The pattern allows you to produce different types and representations of an object using the same construction code.

extract the object construction code out of its own class and move it to separate objects called *builders*.

The pattern organizes object construction into a set of steps (`buildWalls`, `buildDoor`, etc.). To create an object, you execute a series of these steps on a builder object. The important part is that you don't need to call all of the steps. You can call only those steps that are necessary for producing a particular configuration of an object.

Director
You can go further and extract a series of calls to the builder steps you use to construct a product into a separate class called *director*. The director class defines the order in which to execute the building steps, while the builder provides the implementation for those steps.



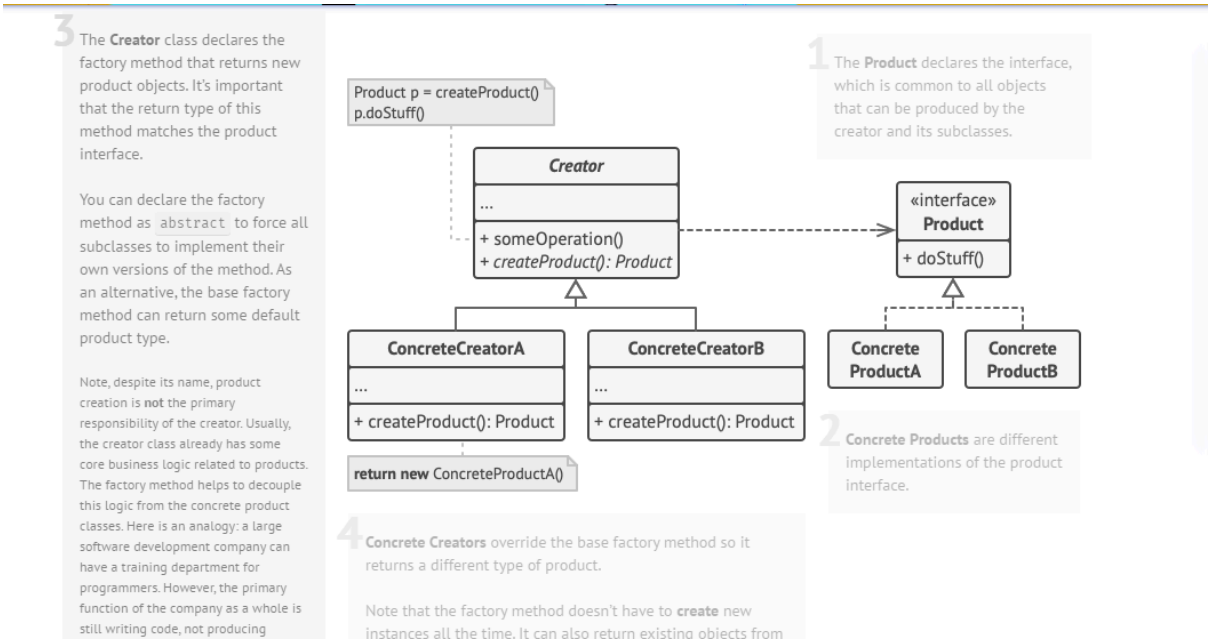
Factory Method
Provides an interface for creating objects in a superclass, but allows subclasses to alter the type of objects that will be created.

Replace direct object construction calls (using the `new` operator) with calls to a special *factory* method. Objects returned by a factory method are often referred to as *products.*

you can override the factory method in a subclass and change the class of products being created by the method.

There's a slight limitation though: subclasses may return different types of products only if these products have a common base class or interface. Also, the factory method in the base class should have its return type declared as this interface.

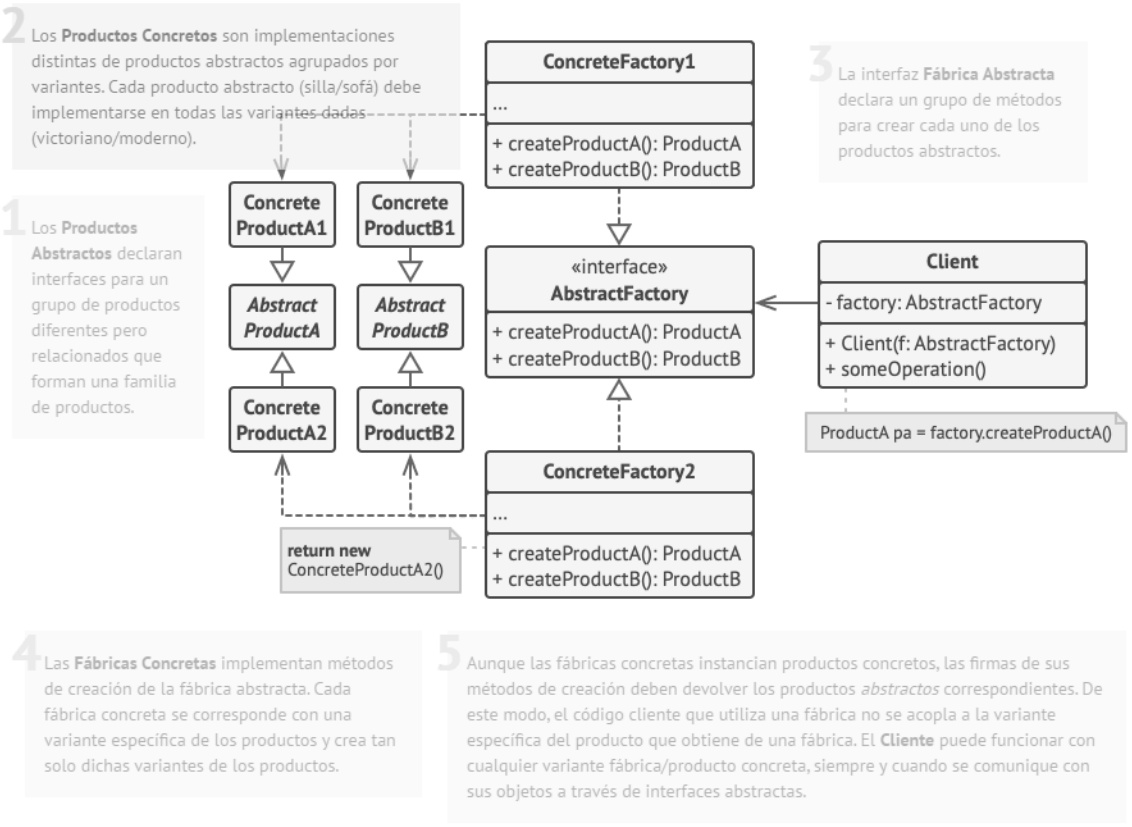
*The code that uses the factory method (often called the **client code**) doesn't see a difference between the actual products returned by various subclasses. The client treats all the products as abstract `Transport`. The client knows that all transport objects are supposed to have the `deliver` method, but exactly how it works isn't important to the client.*



Abstract Factory
Lets you produce families of related objects without specifying their concrete classes.

Para cada variante de una familia de productos, creamos una clase de fábrica independiente basada en la interfaz `FábricaAbstracta`. Una fábrica es una clase que devuelve productos de un tipo particular. Por ejemplo, la `FábricadeMueblesModernos` sólo puede crear objetos de `SillaModerna`, `SofáModerno` y `MesillaModerna`.

Estructura



Prototype:
Nos permite copiar objetos existentes sin que el código dependa de sus clases.

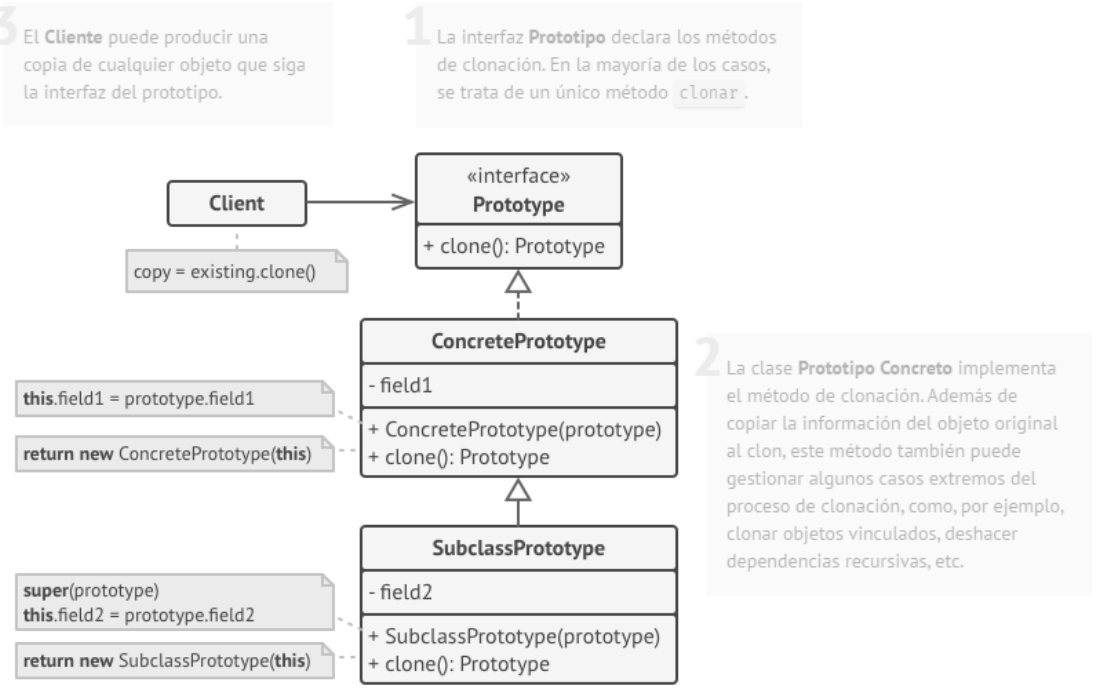
Copiar y pegar los valores de un objeto no es una vía correcta ya que algunos de los campos del objeto pueden ser privados e invisibles desde fuera del propio objeto.
Y otro problema de ese enfoque directo es que debes conocer la clase del objeto para crear un duplicado, el código se vuelve dependiente de esa clase

El patrón Prototype **delega el proceso de clonación a los propios objetos que están siendo clonados**. El patrón declara una interfaz común para todos los objetos que soportan la clonación. Esta interfaz nos permite clonar un objeto sin acoplar el código a la clase de ese objeto. Normalmente, dicha interfaz contiene un único método `clonar`.

El método crea un objeto a partir de la clase actual y lleva todos los valores de campo del viejo objeto, al nuevo. Se puede incluso copiar campos privados.

Un objeto que soporta la clonación se denomina *prototipo*.

Implementación básica

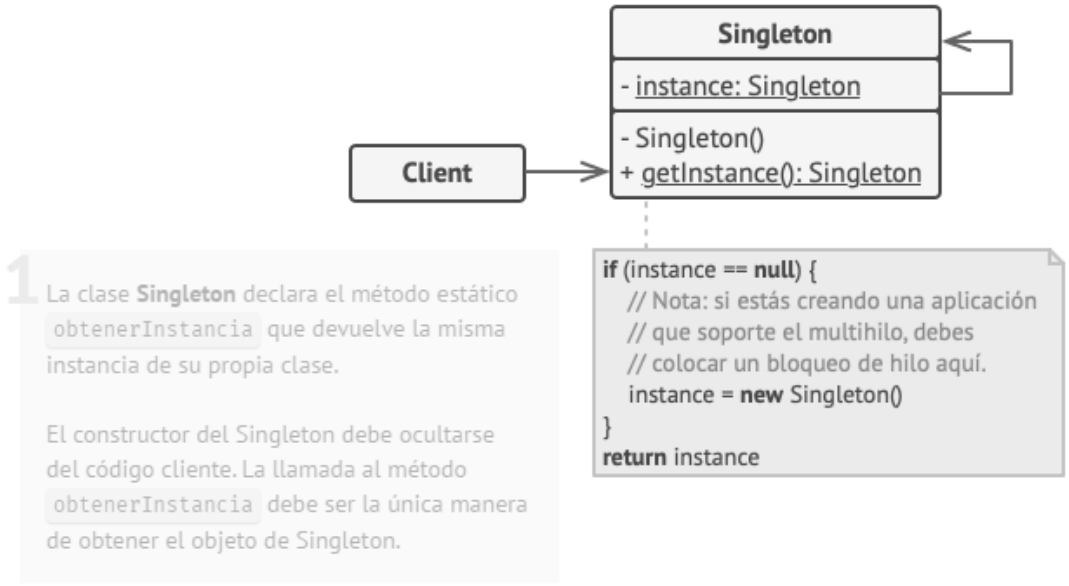


Singleton
Nos permite asegurarnos de que una clase tenga una única instancia, a la vez que proporciona un punto de acceso global a dicha instancia.

Todas las implementaciones del patrón Singleton tienen estos dos pasos en común:

- Hacer privado el constructor por defecto para evitar que otros objetos utilicen el operador `new` con la clase Singleton.
- Crear un método de creación estático que actúe como constructor. Tras bambalinas, este método invoca al constructor privado para crear un objeto y lo guarda en un campo estático. Las siguientes llamadas a este método devuelven el objeto almacenado en caché.

Si tu código tiene acceso a la clase Singleton, podrá invocar su método estático. De esta manera, cada vez que se invoque este método, siempre se devolverá el mismo objeto.



Structural patterns:
Los patrones estructurales explican cómo ensamblar objetos y clases en estructuras más grandes, a la vez que se mantiene la flexibilidad y eficiencia de estas estructuras.

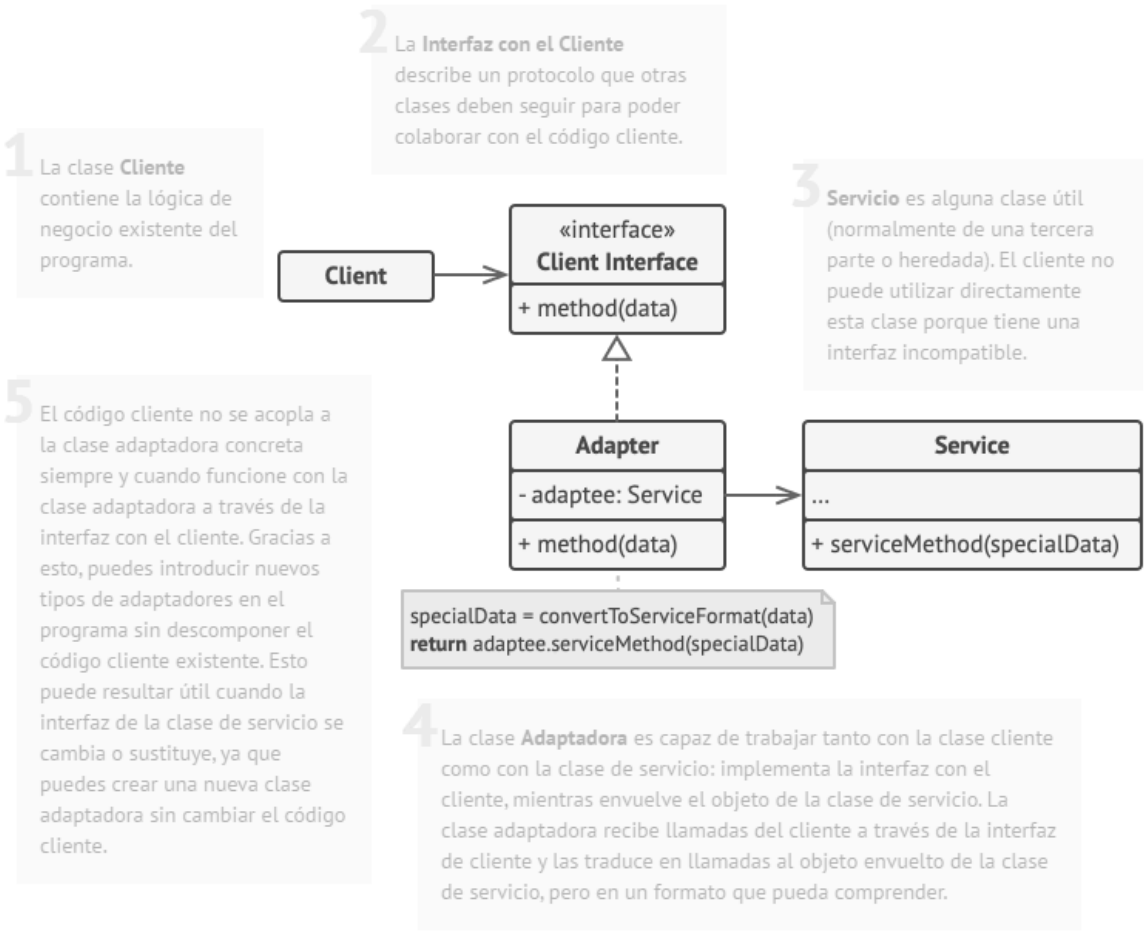
Adapter:
Permite la colaboración entre objetos con interfaces incompatibles.

Es un objeto especial que convierte la interfaz de un objeto, de forma que otro objeto pueda comprenderla.

Los adaptadores no solo convierten datos a varios formatos, sino que también ayudan a objetos con distintas interfaces a colaborar. Funciona así:

1. El adaptador obtiene una interfaz compatible con uno de los objetos existentes.
2. Utilizando esta interfaz, el objeto existente puede invocar con seguridad los métodos del adaptador.
3. Al recibir una llamada, el adaptador pasa la solicitud al segundo objeto, pero en un formato y orden que ese segundo objeto espera.

En ocasiones se puede incluso crear un adaptador de dos direcciones que pueda convertir las llamadas en ambos sentidos.

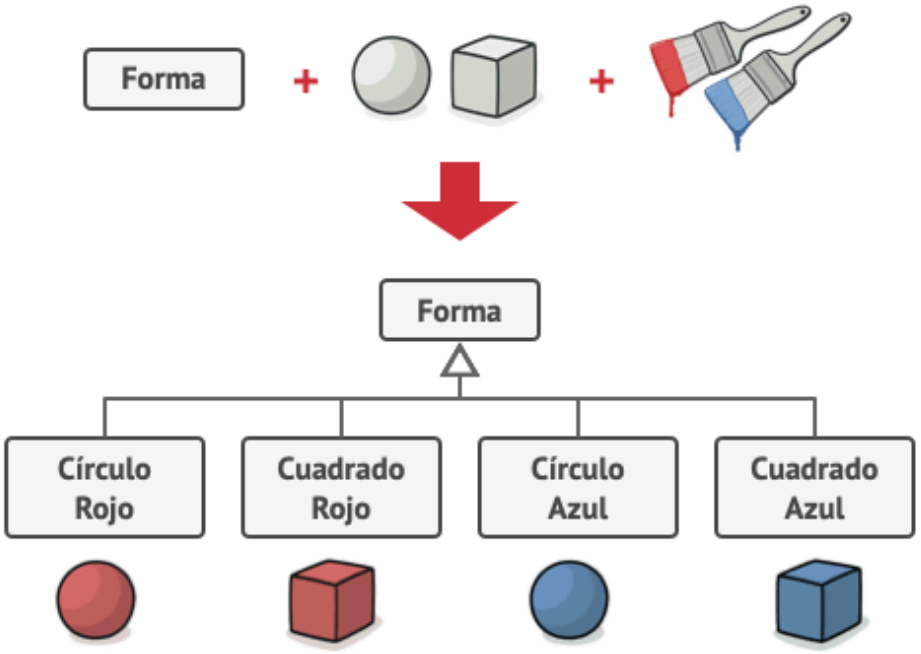


Bridge:

Permite dividir una clase grande, o un grupo de clases estrechamente relacionadas, en dos jerarquías separadas (abstracción e implementación) que pueden desarrollarse independientemente la una de la otra.

Problema

Digamos que tienes una clase geométrica `Forma` con un par de subclases: `Círculo` y `Cuadrado`. Deseas extender esta jerarquía de clase para que incorpore colores, por lo que planeas crear las subclases de forma `Rojo` y `Azul`. Sin embargo, como ya tienes dos subclases, tienes que crear cuatro combinaciones de clase, como `CírculoAzul` y `CuadradoRojo`.

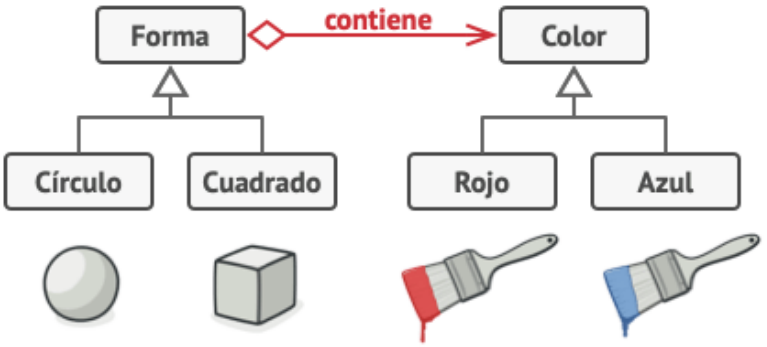


El número de combinaciones de clase crece en progresión geométrica.

Añadir nuevos tipos de forma y color a la jerarquía hará que ésta crezca exponencialmente. Por ejemplo, para añadir una forma de triángulo deberás introducir dos subclases, una para cada color. Y, después, para añadir un nuevo color habrá que crear tres subclases, una para cada tipo de forma. Cuanto más avancemos, peor será.

Solución

El patrón Bridge intenta resolver este problema pasando de la herencia a la composición del objeto. Esto quiere decir que se extrae una de las dimensiones a una jerarquía de clases separada, de modo que las clases originales referencian un objeto de la nueva jerarquía, en lugar de tener todo su estado y sus funcionalidades dentro de una clase.



Puedes evitar la explosión de una jerarquía de clase transformándola en varias jerarquías relacionadas.

Abstracción e implementación

La *Abstracción* (también llamada *interfaz*) es una capa de control de alto nivel para una entidad. Esta capa no tiene que hacer ningún trabajo real por su cuenta, sino que debe delegar el trabajo a la capa de *implementación* (también llamada *plataforma*).

Cuando hablamos de aplicación reales, la abstracción puede representarse por una interfaz gráfica de usuario (GUI), y la implementación puede ser el código del sistema operativo subyacente (API) a la que la capa GUI llama en respuesta a las interacciones del usuario.

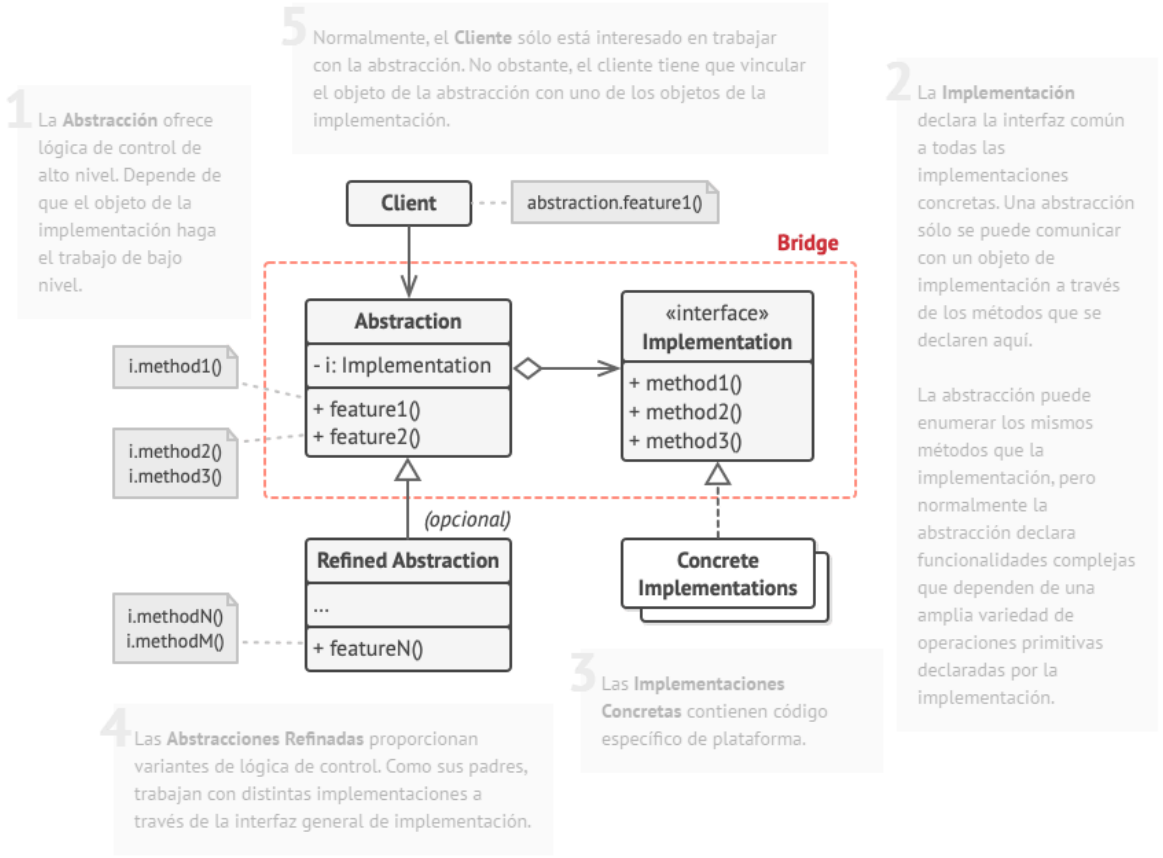
En términos generales, puedes extender esa aplicación en dos direcciones independientes:

- Tener varias GUI diferentes (por ejemplo, personalizadas para clientes regulares o administradores).
- Soportar varias API diferentes (por ejemplo, para poder lanzar la aplicación con Windows, Linux y macOS).

El patrón Bridge, que nos sugiere que dividamos las clases en dos jerarquías:

- Abstracción: la capa GUI de la aplicación.
- Implementación: las API de los sistemas operativos.

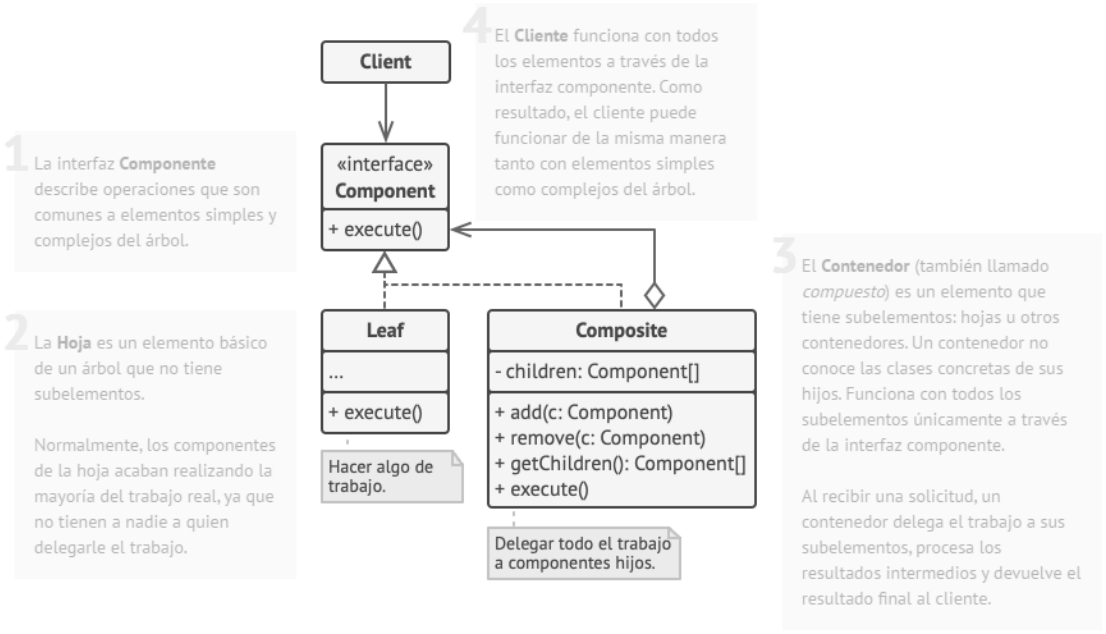
El objeto de la abstracción controla la apariencia de la aplicación, delegando el trabajo real al objeto de la implementación vinculado. Las distintas implementaciones son intercambiables siempre y cuando sigan una interfaz común, permitiendo a la misma GUI funcionar con Windows y Linux.



Composite:
Permite componer objetos en estructuras de árbol y trabajar con esas estructuras como si fueran objetos individuales.

El patrón Composite sugiere que trabajes con `Productos` y `Cajas` a través de una interfaz común que declara un método para calcular el precio total.

¿Cómo funcionaría este método? Para un producto, sencillamente devuelve el precio del producto. Para una caja, recorre cada artículo que contiene la caja, pregunta su precio y devuelve un total por la caja. Si uno de esos artículos fuera una caja más pequeña, esa caja también comenzaría a repasar su contenido y así sucesivamente, hasta que se calcule el precio de todos los componentes internos. Una caja podría incluso añadir costos adicionales al precio final, como costos de empaquetado.



Decorator:
Permite añadir funcionalidades a objetos colocando estos objetos dentro de objetos encapsuladores especiales que contienen estas funcionalidades.

Cuando tenemos que alterar la funcionalidad de un objeto, lo primero que se viene a la mente es extender una clase. No obstante, la herencia tiene varias limitaciones importantes de las que debes ser consciente.

- La herencia es estática. No se puede alterar la funcionalidad de un objeto existente durante el tiempo de ejecución. Sólo se puede sustituir el objeto completo por otro creado a partir de una subclase diferente.
- Las subclases sólo pueden tener una clase padre. En la mayoría de lenguajes, la herencia no permite a una clase heredar comportamientos de varias clases al mismo tiempo.

Una de las formas de superar estas limitaciones es empleando la *Agregación* o la *Composición* en lugar de la *Herencia*. Ambas alternativas funcionan prácticamente del mismo modo: un objeto *tiene una* referencia a otro y le delega parte del trabajo,

mientras que con la herencia, el propio objeto *puede* realizar ese trabajo, heredando el comportamiento de su superclase.

Con esta nueva solución puedes sustituir fácilmente el objeto “ayudante” vinculado por otro, cambiando el comportamiento del contenedor durante el tiempo de ejecución. Un objeto puede utilizar el comportamiento de varias clases con referencias a varios objetos, delegándoles todo tipo de tareas. La agregación/composición es el principio clave que se esconde tras muchos patrones de diseño, incluyendo el Decorator. A propósito, regresemos a la discusión sobre el patrón.

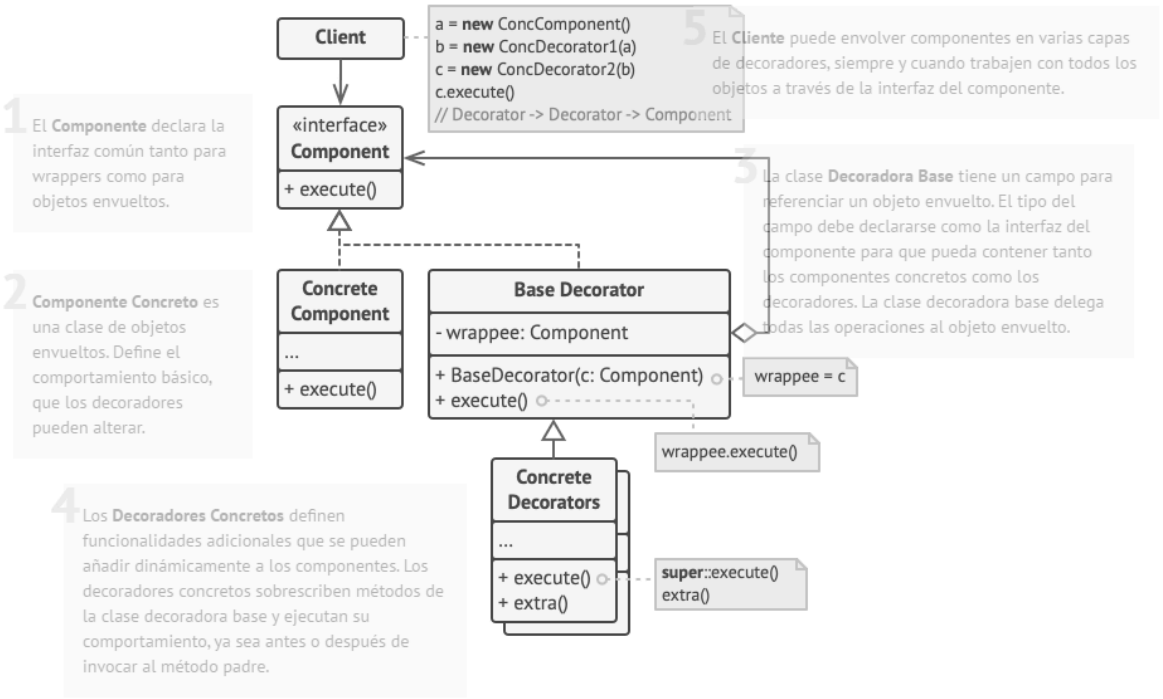
Cuando tenemos que alterar la funcionalidad de un objeto, lo primero que se viene a la mente es extender una clase. No obstante, la herencia tiene varias limitaciones importantes de las que debes ser consciente.

- La herencia es estática. No se puede alterar la funcionalidad de un objeto existente durante el tiempo de ejecución. Sólo se puede sustituir el objeto completo por otro creado a partir de una subclase diferente.
- Las subclases sólo pueden tener una clase padre. En la mayoría de lenguajes, la herencia no permite a una clase heredar comportamientos de varias clases al mismo tiempo.

Una de las formas de superar estas limitaciones es empleando la *Agregación* o la *Composición* en lugar de la *Herencia*. Ambas alternativas funcionan prácticamente del mismo modo: un objeto *tiene una* referencia a otro y le delega parte del trabajo, mientras que con la herencia, el propio objeto *puede* realizar ese trabajo, heredando el comportamiento de su superclase.

Con esta nueva solución puedes sustituir fácilmente el objeto “ayudante” vinculado por otro, cambiando el comportamiento del contenedor durante el tiempo de ejecución. Un objeto puede utilizar el comportamiento de varias clases con referencias a varios objetos, delegándoles todo tipo de tareas. La agregación/composición es el principio clave que se esconde tras muchos patrones de diseño, incluyendo el Decorator. A propósito, regresemos a la discusión sobre el patrón.

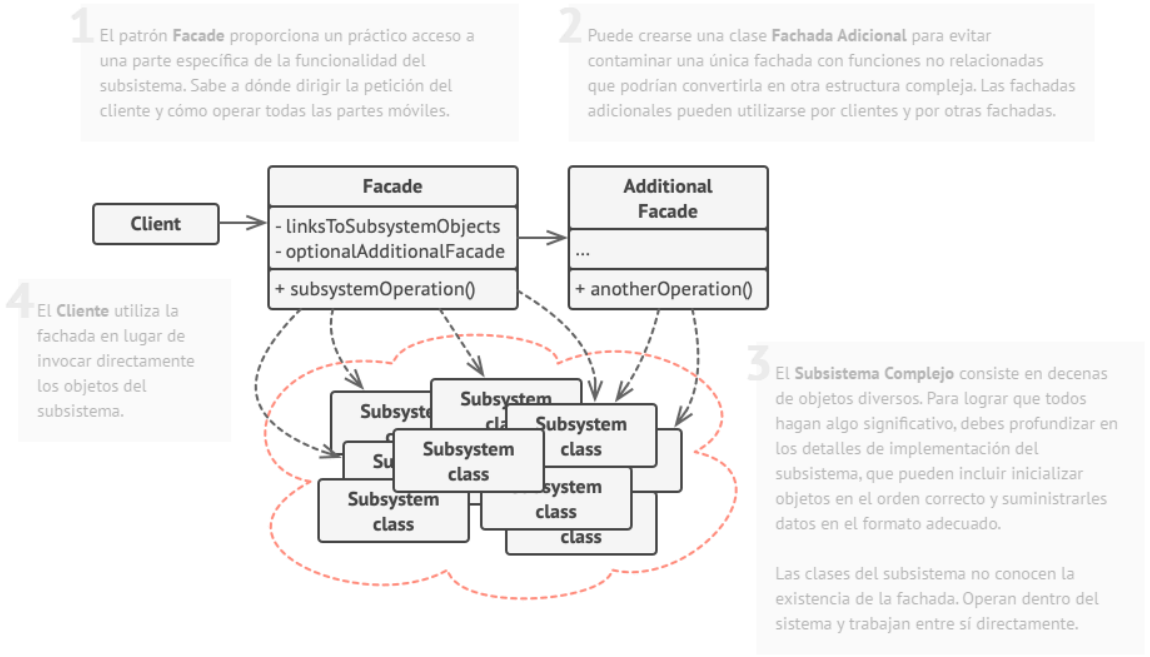
“Wrapper” (envoltorio, en inglés) es el sobrenombre alternativo del patrón Decorator, que expresa claramente su idea principal. Un *wrapper* es un objeto que puede vincularse con un objeto *objetivo*. El wrapper contiene el mismo grupo de métodos que el objetivo y le delega todas las solicitudes que recibe. No obstante, el wrapper puede alterar el resultado haciendo algo antes o después de pasar la solicitud al objetivo.



Facade:
Proporciona una interfaz simplificada a una biblioteca, un framework o cualquier otro grupo complejo de clases.

Una fachada es una clase que proporciona una interfaz simple a un subsistema complejo que contiene muchas partes móviles. Una fachada puede proporcionar una funcionalidad limitada en comparación con trabajar directamente con el subsistema. Sin embargo, tan solo incluye las funciones realmente importantes para los clientes.

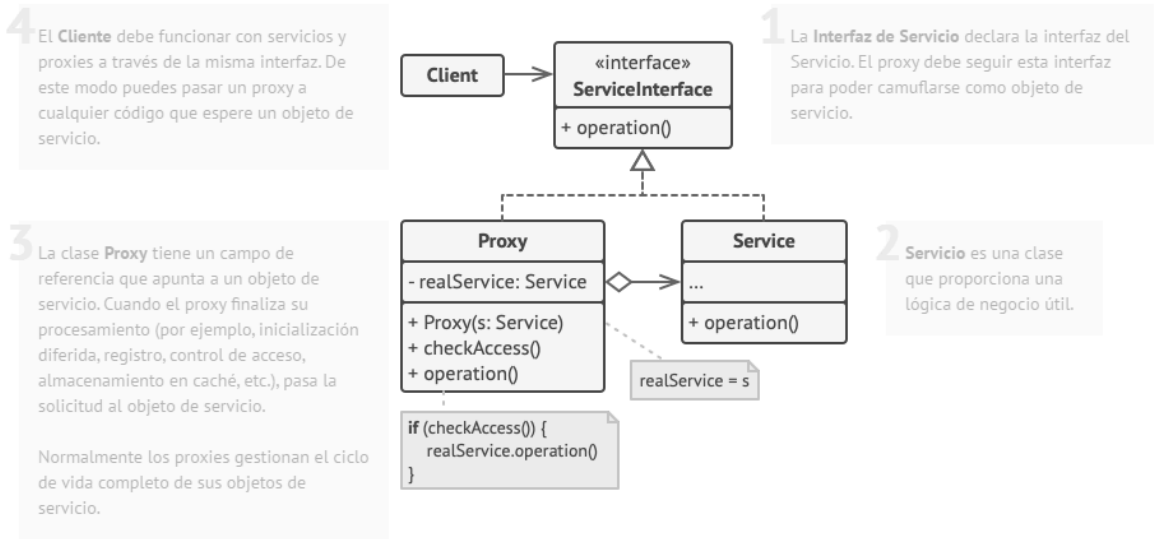
Tener una fachada resulta útil cuando tienes que integrar tu aplicación con una biblioteca sofisticada con decenas de funciones, de la cual sólo necesitas una pequeña parte.



Proxy:
Permite proporcionar un sustituto o marcador de posición para otro objeto. Un proxy controla el acceso al objeto original, permitiéndote hacer algo antes o después de que la solicitud llegue al objeto original.

El patrón Proxy sugiere que crees una nueva clase proxy con la misma interfaz que un objeto de servicio original. Después actualizas tu aplicación para que pase el objeto proxy a todos los clientes del objeto original. Al recibir una solicitud de un cliente, el proxy crea un objeto de servicio real y le delega todo el trabajo.

Estructura



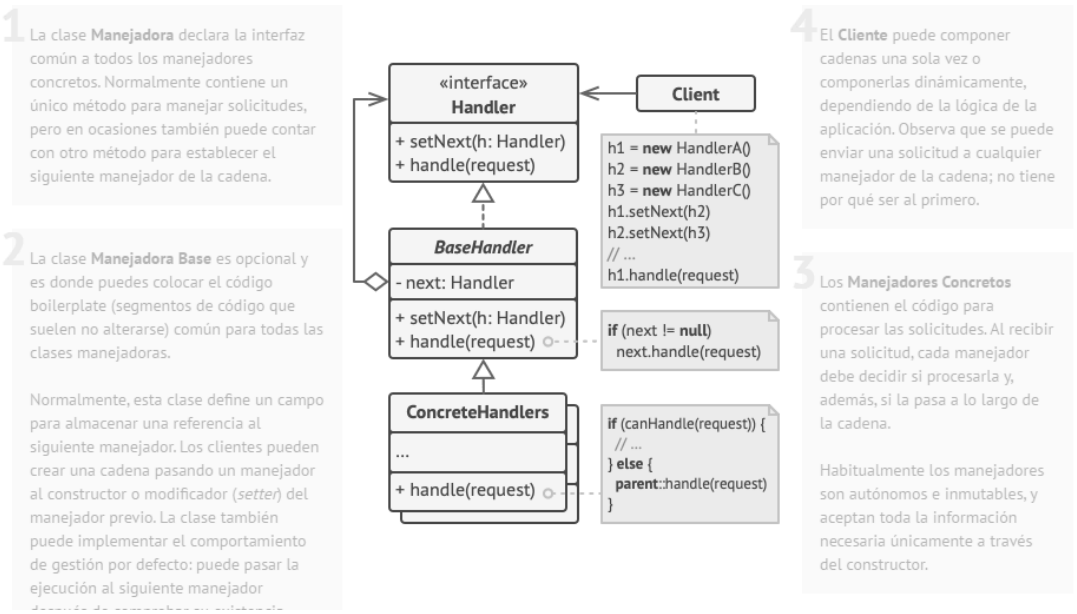
Behavioral
Los patrones de comportamiento tratan con algoritmos y la asignación de responsabilidades entre objetos.

Chain of Responsability:
Permite pasar solicitudes a lo largo de una cadena de manejadores. Al recibir una solicitud, cada manejador decide si la procesa o si la pasa al siguiente manejador de la cadena.

El Chain of Responsibility se basa en transformar comportamientos particulares en objetos autónomos llamados *manejadores*. En nuestro caso, cada comprobación debe ponerse dentro de su propia clase con un único método que realice la comprobación. La solicitud, junto con su información, se pasa a este método como argumento.

El patrón sugiere que vincules esos manejadores en una cadena. Cada manejador vinculado tiene un campo para almacenar una referencia al siguiente manejador de la cadena. Además de procesar una solicitud, los manejadores la pasan a lo largo de la cadena. La solicitud viaja por la cadena hasta que todos los manejadores han tenido la oportunidad de procesarla.

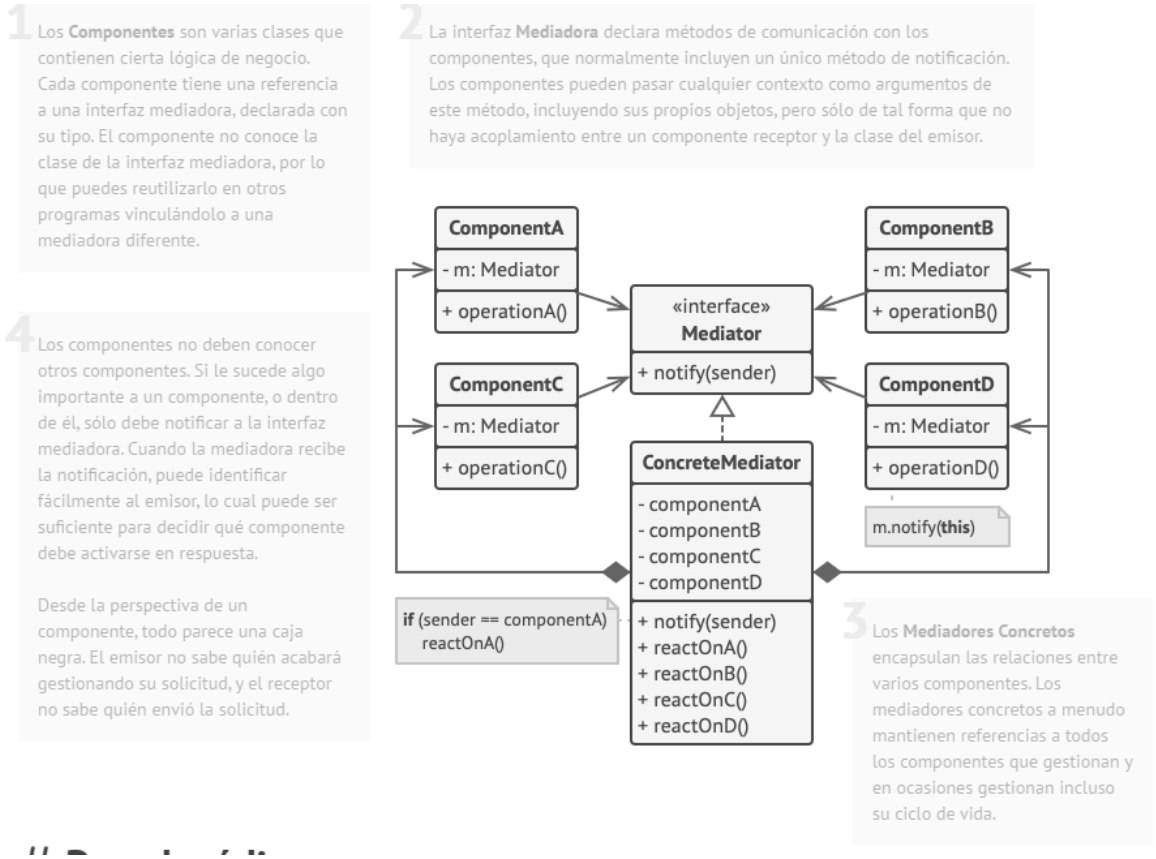
Y ésta es la mejor parte: un manejador puede decidir no pasar la solicitud más allá por la cadena y detener con ello el procesamiento.



Mediator

Permite reducir las dependencias caóticas entre objetos. El patrón restringe las comunicaciones directas entre los objetos, forzándolos a colaborar únicamente a través de un objeto mediador.

El patrón Mediator sugiere que detengas toda comunicación directa entre los componentes que quieres hacer independientes entre sí. En lugar de ello, estos componentes deberán colaborar indirectamente, invocando un objeto mediador especial que redireccione las llamadas a los componentes adecuados. Como resultado, los componentes dependen únicamente de una sola clase mediadora, en lugar de estar acoplados a decenas de sus colegas.



Memento:

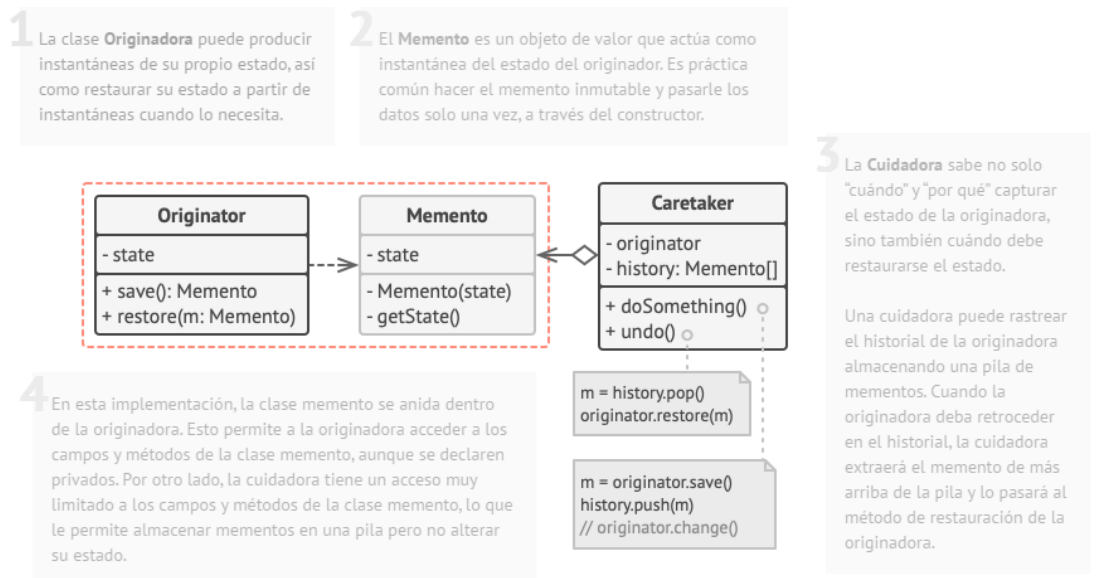
Permite guardar y restaurar el estado previo de un objeto sin revelar los detalles de su implementación.

El patrón Memento delega la creación de instantáneas de estado al propietario de ese estado, el objeto *originador*. Por lo tanto, en lugar de que haya otros objetos

intentando copiar el estado del editor desde el “exterior”, la propia clase editora puede hacer la instantánea, ya que tiene pleno acceso a su propio estado.

El patrón sugiere almacenar la copia del estado del objeto en un objeto especial llamado *memento*. Los contenidos del memento no son accesibles para ningún otro objeto excepto el que lo produjo. Otros objetos deben comunicarse con mementos utilizando una interfaz limitada que pueda permitir extraer los metadatos de la instantánea (tiempo de creación, el nombre de la operación realizada, etc.), pero no el estado del objeto original contenido en la instantánea.

Una política tan restrictiva te permite almacenar mementos dentro de otros objetos, normalmente llamados *cuidadores*. Debido a que el cuidador trabaja con el memento únicamente a través de la interfaz limitada, no puede manipular el estado almacenado dentro del memento. Al mismo tiempo, el originador tiene acceso a todos los campos dentro del memento, permitiéndole restaurar su estado previo a voluntad.



Observer

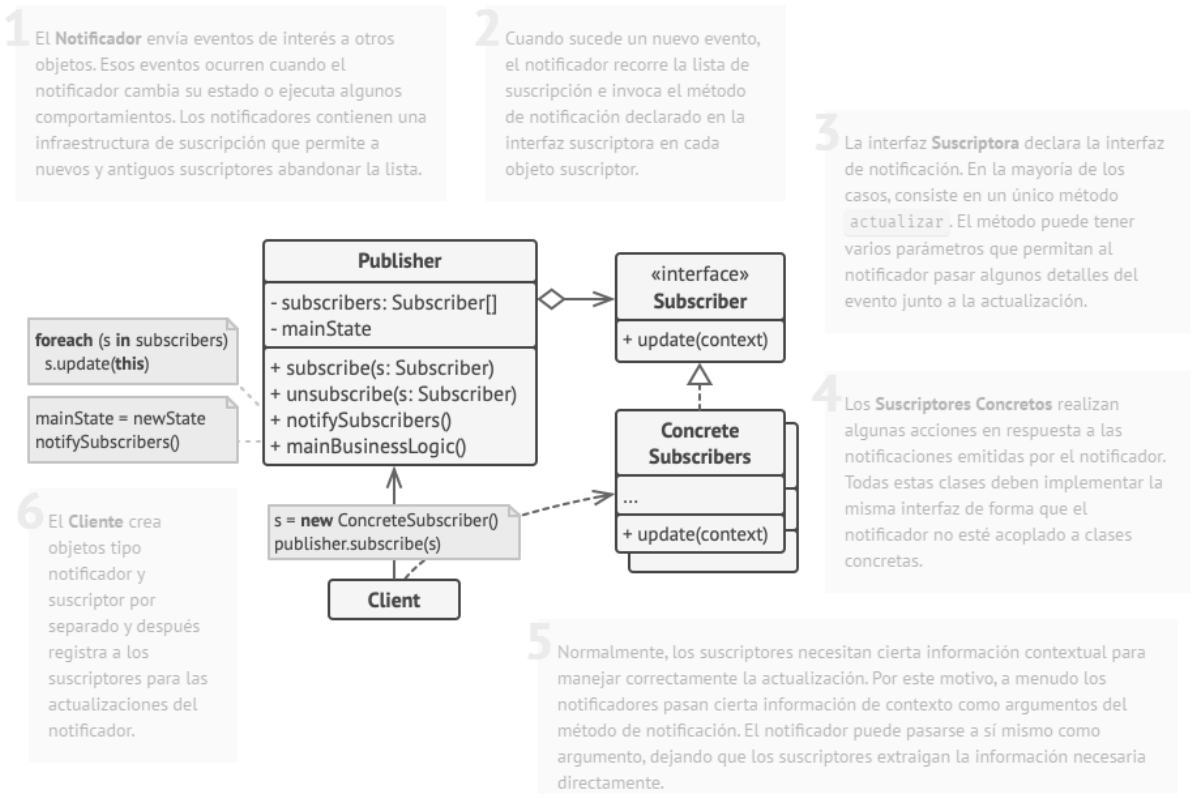
Permite definir un mecanismo de suscripción para notificar a varios objetos sobre cualquier evento que le suceda al objeto que están observando.

El objeto que tiene un estado interesante suele denominarse *sujeto*, pero, como también va a notificar a otros objetos los cambios en su estado, le llamaremos *notificador* (en ocasiones también llamado *publicador*). El resto de los objetos que

quieren conocer los cambios en el estado del notificador, se denominan *suscriptores*.

El patrón Observer sugiere que añadas un mecanismo de suscripción a la clase notificadora para que los objetos individuales puedan suscribirse o cancelar su suscripción a un flujo de eventos que proviene de esa notificadora. ¡No temas! No es tan complicado como parece. En realidad, este mecanismo consiste en: 1) un campo matriz para almacenar una lista de referencias a objetos suscriptores y 2) varios métodos públicos que permiten añadir suscriptores y eliminarlos de esa lista.

Ahora, cuando le sucede un evento importante al notificador, recorre sus suscriptores y llama al método de notificación específico de sus objetos. Es fundamental que todos los suscriptores implementen la misma interfaz y que el notificador únicamente se comunique con ellos a través de esa interfaz.



State

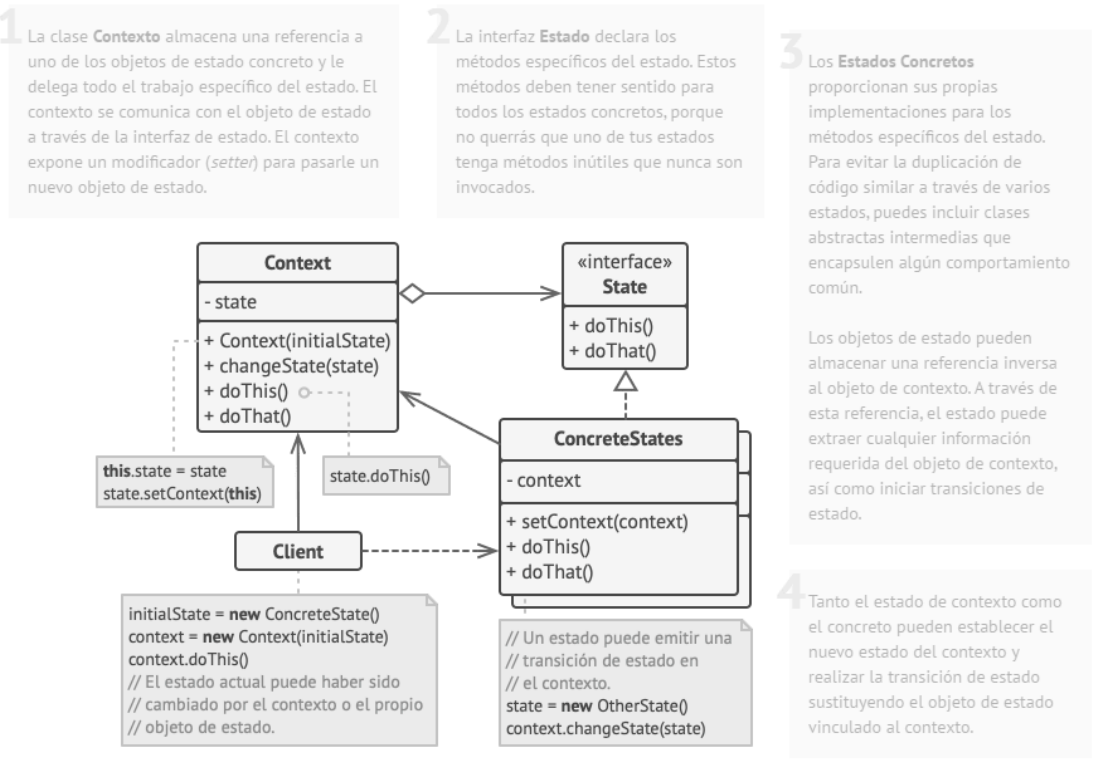
Permite a un objeto alterar su comportamiento cuando su estado interno cambia. Parece como si el objeto cambiara su clase.

El patrón State sugiere que crees nuevas clases para todos los estados posibles de un objeto y extraigas todos los comportamientos específicos del estado para colocarlos dentro de esas clases.

En lugar de implementar todos los comportamientos por su cuenta, el objeto original, llamado *contexto*, almacena una referencia a uno de los objetos de estado que representa su estado actual y delega todo el trabajo relacionado con el estado a ese objeto.

Para la transición del contexto a otro estado, sustituye el objeto de estado activo por otro objeto que represente ese nuevo estado. Esto sólo es posible si todas las clases de estado siguen la misma interfaz y el propio contexto funciona con esos objetos a través de esa interfaz.

Esta estructura puede resultar similar al patrón **Strategy**, pero hay una diferencia clave. En el patrón State, los estados particulares pueden conocerse entre sí e iniciar transiciones de un estado a otro, mientras que las estrategias casi nunca se conocen.



Strategy

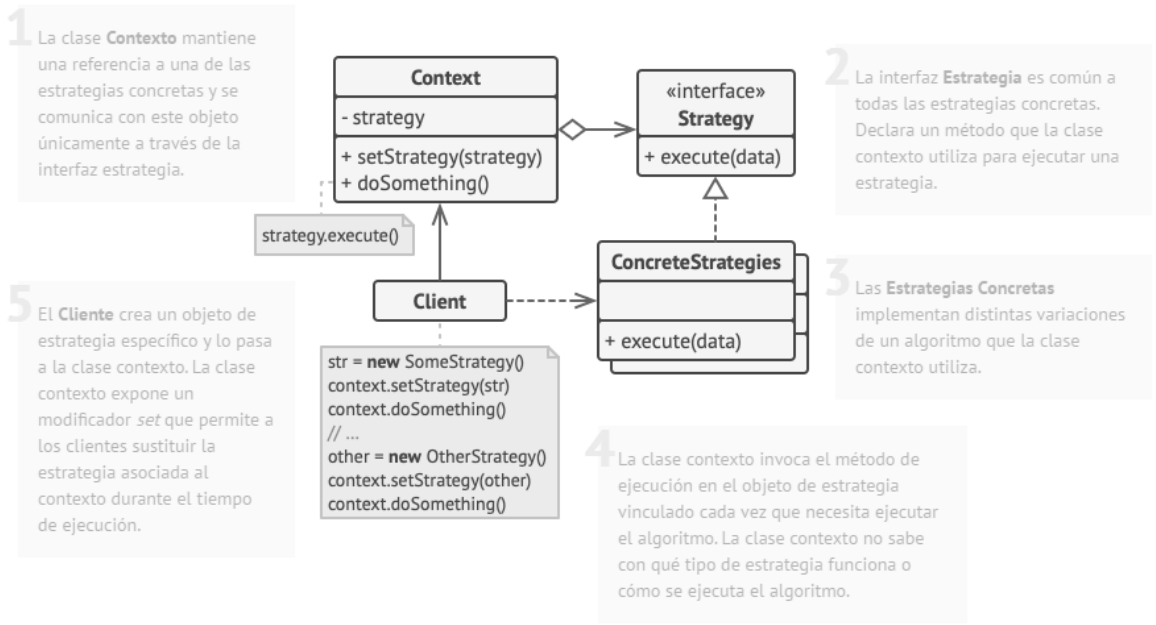
Permite definir una familia de algoritmos, colocar cada uno de ellos en una clase separada y hacer sus objetos intercambiables.

El patrón Strategy sugiere que tomes esa clase que hace algo específico de muchas formas diferentes y extraigas todos esos algoritmos para colocarlos en clases separadas llamadas *estrategias*.

La clase original, llamada *contexto*, debe tener un campo para almacenar una referencia a una de las estrategias. El contexto delega el trabajo a un objeto de estrategia vinculado en lugar de ejecutarlo por su cuenta.

La clase contexto no es responsable de seleccionar un algoritmo adecuado para la tarea. En lugar de eso, el cliente pasa la estrategia deseada a la clase contexto. De hecho, la clase contexto no sabe mucho acerca de las estrategias. Funciona con todas las estrategias a través de la misma interfaz genérica, que sólo expone un único método para disparar el algoritmo encapsulado dentro de la estrategia seleccionada.

De esta forma, el contexto se vuelve independiente de las estrategias concretas, así que puedes añadir nuevos algoritmos o modificar los existentes sin cambiar el código de la clase contexto o de otras estrategias.



Visitor

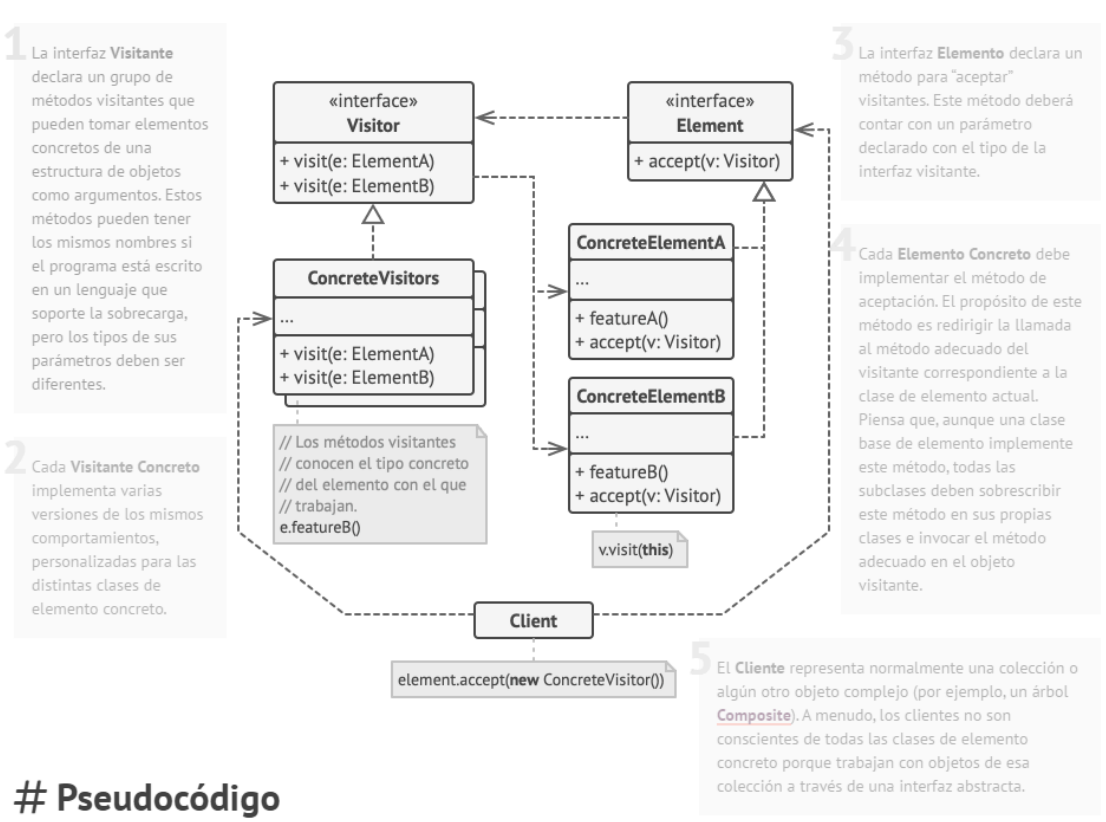
Permite separar algoritmos de los objetos sobre los que operan.

El patrón Visitor sugiere que coloques el nuevo comportamiento en una clase separada llamada *visitante*, en lugar de intentar integrarlo dentro de clases existentes. El objeto que originalmente tenía que realizar el comportamiento se pasa ahora a uno de los métodos del visitante como argumento, de modo que el método accede a toda la información necesaria contenida dentro del objeto.

La clase visitante puede definir un grupo de métodos en lugar de uno solo, y cada uno de ellos podría tomar argumentos de distintos tipos

Estos métodos tienen distintas firmas, por lo que no podemos utilizar el polimorfismo. Para elegir un método visitante adecuado que sea capaz de procesar un objeto dado, debemos revisar su clase. ¿No suena esto como una pesadilla?

Sin embargo, el patrón Visitor ataja este problema. Utiliza una técnica llamada **Double Dispatch**, que ayuda a ejecutar el método adecuado sobre un objeto sin complicados condicionales. En lugar de permitir al cliente seleccionar una versión adecuada del método a llamar, ¿qué tal si delegamos esta opción a los objetos que pasamos al visitante como argumento? Como estos objetos conocen sus propias clases, podrán elegir un método adecuado en el visitante más fácilmente. “Aceptan” un visitante y le dicen qué método visitante debe ejecutarse.



Pseudocódigo

Antipatterns

What Is an AntiPattern?

An AntiPattern is a literary form that describes a commonly occurring solution to a problem that generates decidedly negative consequences.

This form reduces the most common mistake in using design patterns: applying a particular design pattern in the improper context. AntiPatterns provide real−world experience in recognizing recurring problems in the software industry and provide a detailed remedy for the most common predicaments

AntiPatterns provide a common vocabulary for identifying problems and discussing solutions.

AntiPatterns support the holistic resolution of conflicts, utilizing organizational resources at several levels, where possible.

AntiPatterns provide stress release in the form of shared misery for the most common pitfalls in the software industry

The essence of an AntiPattern is **two solutions**, instead of a problem and a solution for ordinary design patterns. The **first solution is problematic**. It is a commonly occurring solution that generates overwhelmingly negative consequences. **The second solution is called the refactored solution**. The refactored solution is a commonly occurring method in which the AntiPattern can be resolved and reengineered into a more beneficial form.

Advice for Using AntiPatterns

While assigning blame and pointing fingers may provide a temporary rush of satisfaction, this is not the intended use of software AntiPatterns.

However, the absence of AntiPatterns does not guarantee that an organization will be successful

AntiPatterns are most appropriate for resolving chronic problems, especially when they must be proactively addressed in order to meet the organizational goals. Heed the advice: “If it’s not broken, don’t fix it; leave well enough alone.”

The purpose of **AntiPatterns is not to focus on dysfunctional software practices**. Rather, **the purpose is the development and implementation of strategies to fix the problems that arise**.

A plan that involves the simultaneous correction of several AntiPatterns is risky and ill−advised. By focusing on improving processes on a case−by−case basis, according to a well−conceived plan, the chances of successfully implementing a viable solution are markedly increased.

Dysfunctional Environments

A dysfunctional work environment in the software industry is one in which **discussions of organizational politics, controversy, and negativity dominate technical discourse**.

Fixed and Improvisational Responses

An improvisational response involves making something up on the spur of the moment. It’s not a pattern

A fixed response is something learned or practiced. It is a pattern or AntiPattern

Design patterns start with a recurring solution

AntiPatterns start with a recurring problem.

. Use study groups. It is useful to discuss a written description of the AntiPattern and its refactored solution in a group environment

Become an AntiPattern author. New AntiPattern authors can focus on recurring problems that are observed in several distinct contexts

We believe AntiPatterns are a more effective way to communicate software knowledge than ordinary design patterns because:

• AntiPatterns clarify problems for software developers, architects, and managers by **identifying the symptoms and consequences that lead to the dysfunctional software** development processes.

• AntiPatterns **convey the motivation for change** and the need for refactoring poor processes.

• AntiPatterns are **necessary to gain an understanding of common problems** faced by most software developers. Learning from other developers' successes and failures is valuable and necessary. Without this wisdom, AntiPatterns will continue to persist.

Proper AntiPatterns define a migration (or refactoring) from negative solutions to positive solutions.

Software Refactoring

A key goal of development AntiPatterns is to describe useful forms of software refactoring. Software refactoring is a form of code modification, used to improve the software structure in support of subsequent extension and long−term maintenance. In most cases, the goal is to transform code without impacting correctness.

Refactoring is strongly recommended prior to performance optimization

Formal Refactoring Transformations

These formal refactorings originated in Opdyke's PhD thesis [Opdyke 92]. They are called formal refactorings because **implementations can be proven not to affect program correctness**.

- Superclass abstraction
- Conditional elimination
- Aggregate abstraction.

Development AntiPattern Summaries

Development AntiPatterns utilize various formal and informal refactoring approaches. The following summaries provide an overview of the Development AntiPatterns found in this chapter and focus on the development AntiPattern problem:

The Blob: Procedural−style design leads to **one object with a lion's share of the responsibilities**, while most other objects only hold data or execute simple processes. **The solution includes refactoring the design to distribute responsibilities more uniformly and isolating the effect of changes**.

Lava Flow: **Dead code and forgotten design information is frozen in an ever−changing design**. This is analogous to a Lava Flow with hardening globules of rocky material. The refactored **solution includes a configuration management process that eliminates dead code and evolves or refactors design** toward increasing quality.

Golden Hammer: A Golden Hammer is a **familiar technology or concept applied obsessively to many software problems**. The **solution involves expanding the knowledge of developers through education, training, and book study** groups to expose developers to alternative technologies and approaches.

Spaghetti Code: **Ad hoc software structure makes it difficult to extend and optimize code**. Frequent code refactoring can improve software structure, support software maintenance, and enable iterative development.

Cut−and−Paste Programming: **Code reused by copying source statements leads to significant maintenance problems**. Alternative forms of reuse, including **black−box reuse, reduce maintenance issues by having common source code, testing, and documentation**

The blob:

The Blob is found in designs where one class monopolizes the processing, and other classes primarily encapsulate data. This AntiPattern is characterized by a class diagram composed of a single complex controller class surrounded by simple data classes

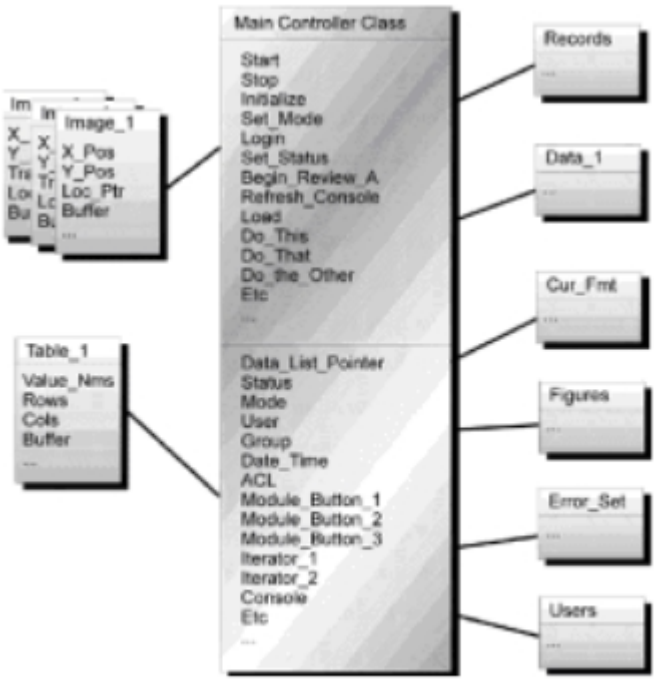


Figure 5.2 Controller Class.

The Blob AntiPattern is acceptable when wrapping legacy systems.

Refactored Solution

As with most of the AntiPatterns in this section, the solution involves a form of refactoring. The key is to move behavior away from the Blob. It may be appropriate to reallocate behavior to some of the encapsulated data objects in a way that makes these objects more capable and the Blob less complex.

Lava Flow:

The Lava Flow AntiPattern is commonly found in systems that originated as research but ended up in production. It is characterized by the lavalike “flows” of previous developmental versions strewn about the code landscape, which have now hardened into a basaltlike, immovable, generally useless mass of code that no one can remember much, if anything, about (see Figure 5.11)

Refactored Solution

There is only one sure−fire way to prevent the Lava Flow AntiPattern: Ensure that sound architecture precedes production code development. This architecture must be backed up by a configuration management process that ensures architectural compliance and accommodates “mission creep” (changing requirements).

Golden Hammer

A software development team has gained a high level of competence in a particular solution or vendor product, referred to here as the Golden Hammer. As a result, every new product or development effort is viewed as something that is best solved with it. In many cases, the Golden Hammer is a mismatch for the problem, but minimal effort is devoted to exploring alternative solutions.

Refactored Solution

This solution involves a philosophical aspect as well as a change in the development process. Philosophically, an organization needs to develop a commitment to an exploration of new technologies. Without such a commitment, the lurking danger of overreliance on a specific technology or vendor tool set exists. This solution requires a two−pronged approach: A greater commitment by management in the professional development of their developers, along with a development strategy that requires explicit software boundaries to enable technology migration

Spaghetti Code

Spaghetti Code appears as a program or system that contains very little software structure. Coding and progressive extensions compromise the software structure to such an extent that the structure lacks clarity, even to the original developer, if he or she is away from the software for any length of time. If developed using an object−oriented language, the software may include a small number of objects that contain methods with very large implementations that invoke a single, multistage process flow. Furthermore, the object methods are invoked in a very predictable manner, and there is a negligible degree of dynamic interaction between the objects in the system. The system is very difficult to maintain and extend, and there is no opportunity to reuse the objects and modules in other similar systems.

Refactored Solution

Software refactoring (or code cleanup) is an essential part of software development [Opdyke 92].

Ideally, code cleanup should be a natural part of the development process. As each feature (or group of features) is added to the code, code cleanup should follow what restores or improves the code structure.

Cut and paste

This AntiPattern is identified by the presence of several similar segments of code interspersed throughout the software project. Usually, the project contains many programmers who are learning how to develop software by following the examples of more experienced developers. However, they are learning by modifying code that has been proven to work in similar situations, and potentially customizing it to support new data types or slightly customized behavior. This creates code duplication, which may have positive short−term consequences such as boosting line count metrics, which may be used in performance evaluations. Furthermore, it’s easy to extend the code as the developer has full control over the code used in his or her application and can quickly meet short−term modifications to satisfy new requirements.

Refactored Solution

Cloning frequently occurs in environments where white−box reuse is the predominant form of system extension. In white−box reuse, developers extend systems primarily though inheritance.

Restructuring software to reduce or eliminate cloning requires modifying code to emphasize black−box reuse of duplicated software segments. In the case where Cut−and−Paste Programming has been used extensively throughout the lifetime of a software project, the most effective method of recovering your investment is to refactor the code base into reusable libraries or components that focus on black−box reuse of functionality