

Prog. Orientada a Objetos

Carrera Programador full-stack

Patrones de Diseño

Agenda

- Aspectos del trabajo en equipo en el software
 - Evitar reinventar la rueda
 - Problemas conocidos → Soluciones conocidas
- Casos de problemas conocidos
- Patrones de Diseño
 - Builder
 - Factory
 - Singleton
 - Composite
 - Adapter
 - Mediator
 - Strategy
 - Observer
 - Decorator
- Recomendaciones
- Ejercicios

Importancia del trabajo en equipo

- En un proyecto, un aspecto central es el *delivery*
 - Entregar las cosas en tiempo y forma
- El uso del tiempo tiene que ser eficiente
 - Hay que evitar perder mucho tiempo en temas que no lo merecen → siempre buscar ayuda
- Si un equipo que hace un buen trabajo y un buen uso del tiempo → cliente contento
- Si un equipo tiene malas prácticas y no usa el tiempo de forma eficiente
 - Problemas con el cliente
 - Problemas entre los miembros del equipo de trabajo

Evitar reinventar la rueda

- En un proyecto, uno de los aspectos centrales es evitar implementar algo que ya esté hecho
- La idea es evitar perder tiempo de forma innecesaria *reinventando la rueda*
- Además, reutilizar un código hecho normalmente implica un código revisado, y que está probado que funciona como debiera
 - En caso de tratarse de una librería, también se tiene documentación para entender su funcionamiento
- Si reinventamos algo que ya está hecho, perdemos tiempo en implementar, testear y documentar el código

Problemas Conocidos Soluciones Conocidas

- Normalmente, muchos de los problemas que tenemos, es muy probable que otras personas ya los hayan tenido
 - Por lo tanto, es muy probable que ya haya una solución a ese tipo de problema
- Es importante conocer las formas típicas de resolver un determinado tipo de problema
- De esta manera, cuando le expliquemos a otra persona cómo resolvimos un problema, al decir que se aplicó tal tipo de solución, nos entiendan mucho más fácil
 - Hasta sin necesidad de ver el código

Ejemplos de Problemas Conocidos

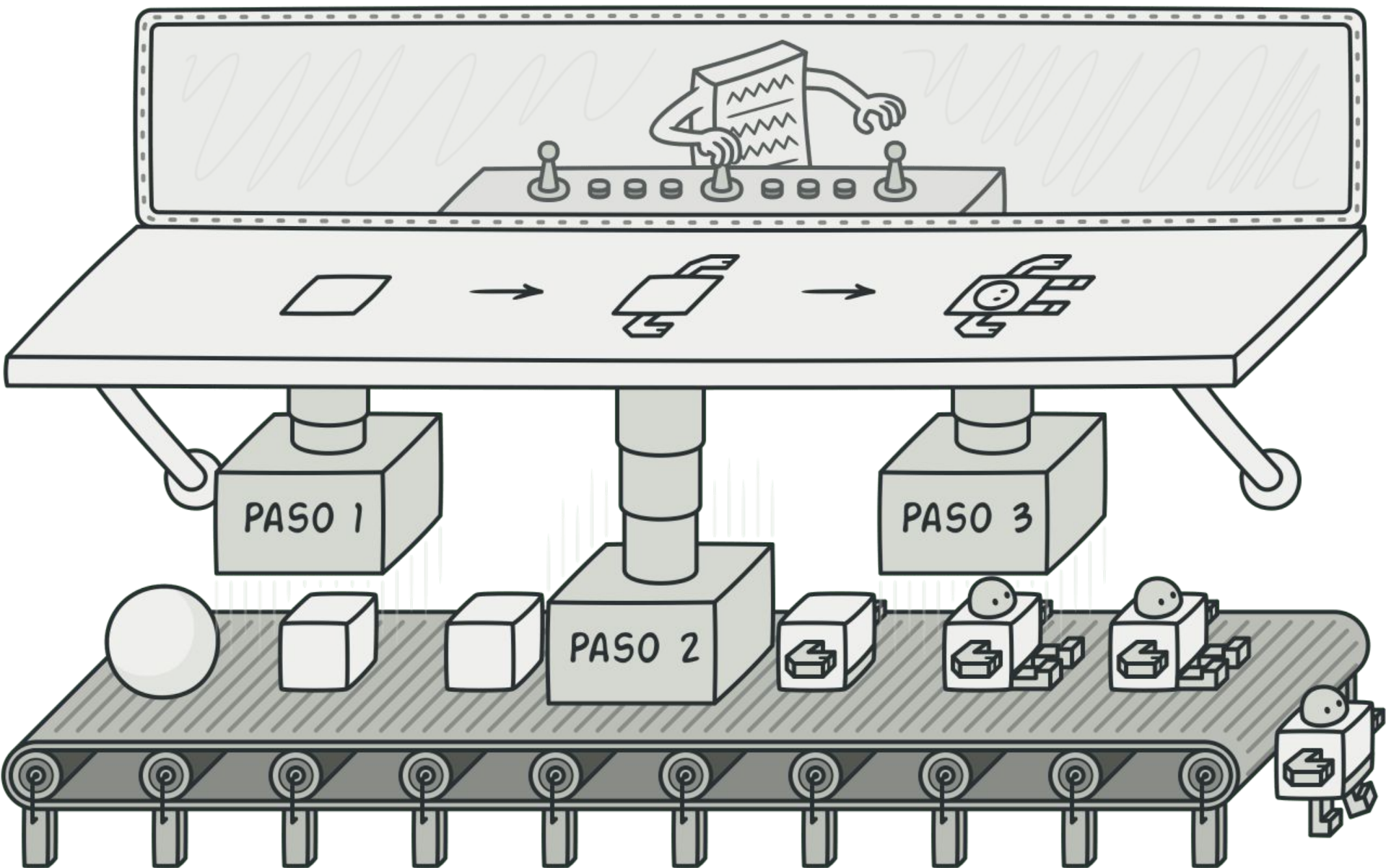
- Constructores con muchos parámetros
- Instanciar diferentes clases con un mismo método
- Restringir la cantidad de veces que se instancie nuestra clase
- Clases que estén compuestas por clases, y a su vez esas clases las componen otras clases
 - Y así sucesivamente
 - Ejemplo clásico → sistema de archivos
- El tipo de retorno de una función nos complica la implementación de nuestro sistema
- Interacción con un sistema complejo, compuesto por varios subsistemas

Patrones de Diseño

- Los problemas conocidos en programación tienen soluciones conocidas
- Las soluciones conocidas se llaman *patrones de diseño*
- Se tratan de formas de programar que facilitan la resolución de un problema
- Existen muchos patrones de diseño
 - Vamos a ver los patrones que usan como solución, los mecanismos de programación orientada a objetos

Patrones de Diseño - Builder

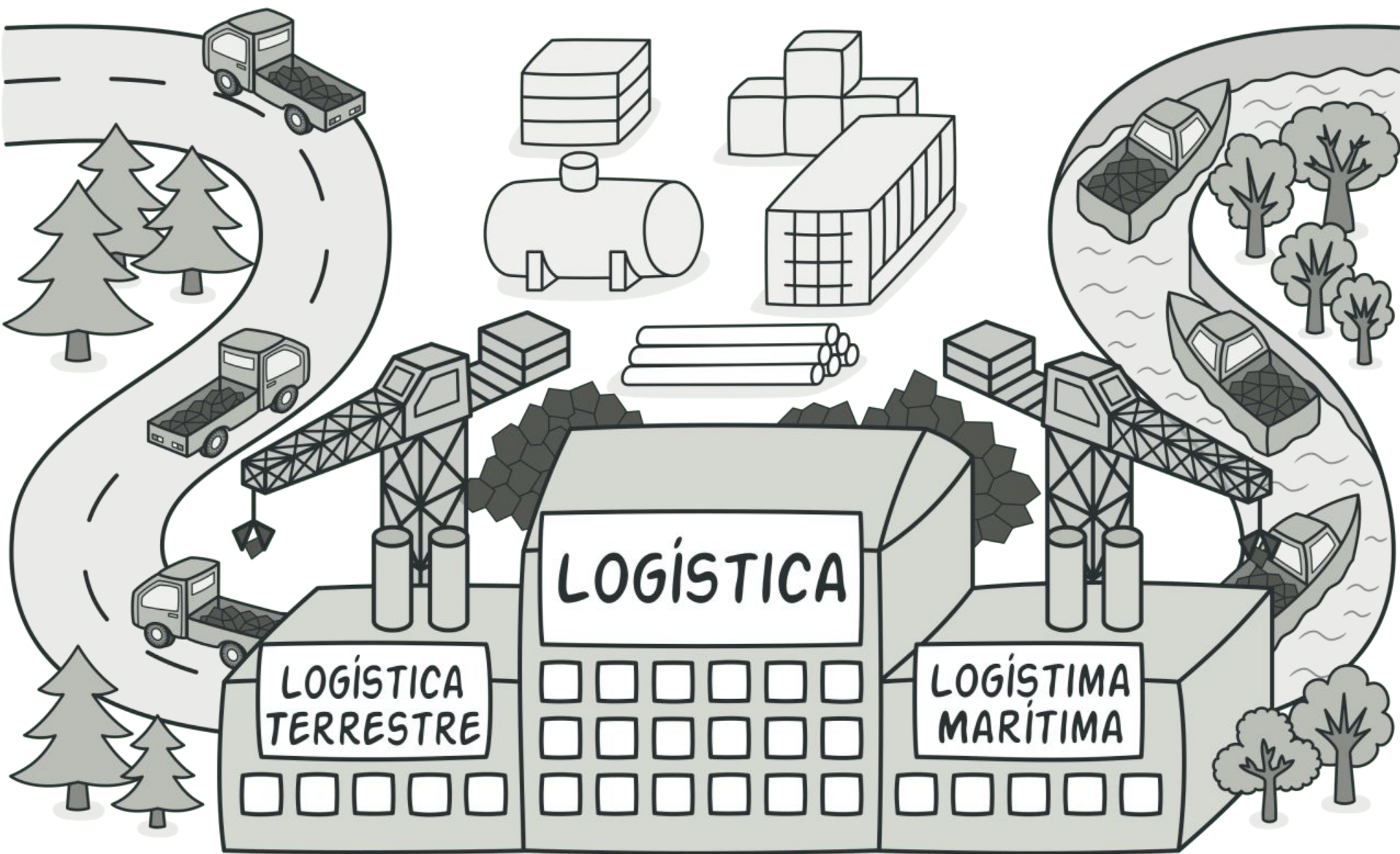
- ¿Cuándo se usa?
 - Cuando queremos construir objetos complejos → con muchos parámetros
 - Cuando queremos hacer la construcción de forma más prolija y clara
- ¿Cómo se usa?
 - Básicamente tenemos la clase compleja a construir
 - Después varias clases que se encargan de construir la clase compleja → cada una construye con un criterio
 - Finalmente se invoca a las clases encargadas de construir la clase compleja, para obtener una instancia de dicha clase compleja



Patrones de Diseño - Factory

- ¿Cuándo se usa?
 - Cuando tenemos varias formas de trabajar dentro de un determinado método
- ¿Cómo se usa?
 - Cada forma diferente de trabajar, es una clase diferente
 - Se define una forma estándar de trabajar → interfaces
 - Cada clase que implemente la interfaz, es un criterio diferente para trabajar

https://sourcemaking.com/design_patterns/factory_method

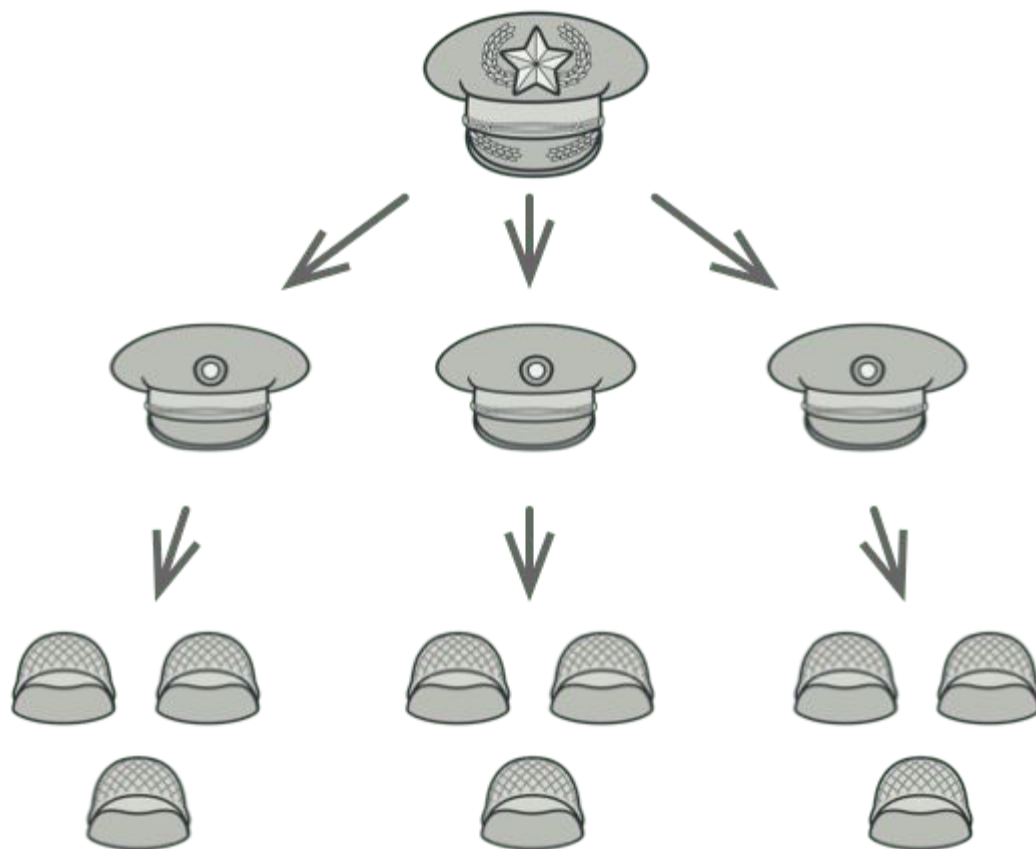


Patrones de Diseño - Singleton

- ¿Cuándo se usa?
 - Cuando se necesita *solamente* una instancia de una clase
 - Se puede usar por ejemplo en conexiones, loggers, etc.
- ¿Cómo se usa?
 - La idea es que la persona que implementa la clase, use los mecanismos de POO para restringir la instanciación de la clase
 - Variable interna del mismo tipo de la clase
 - Instanciada una sola vez
 - El constructor es privado
 - Se tiene un getter para obtener la instancia única

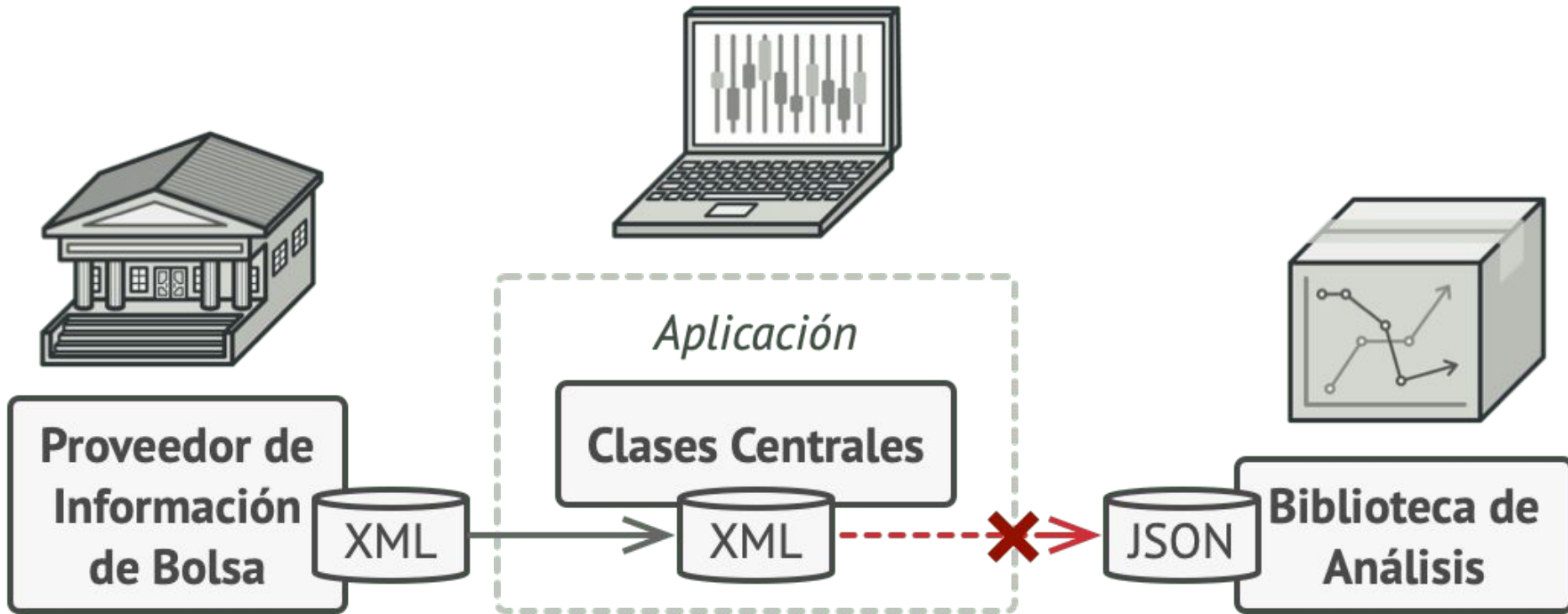
Patrones de Diseño - Composite

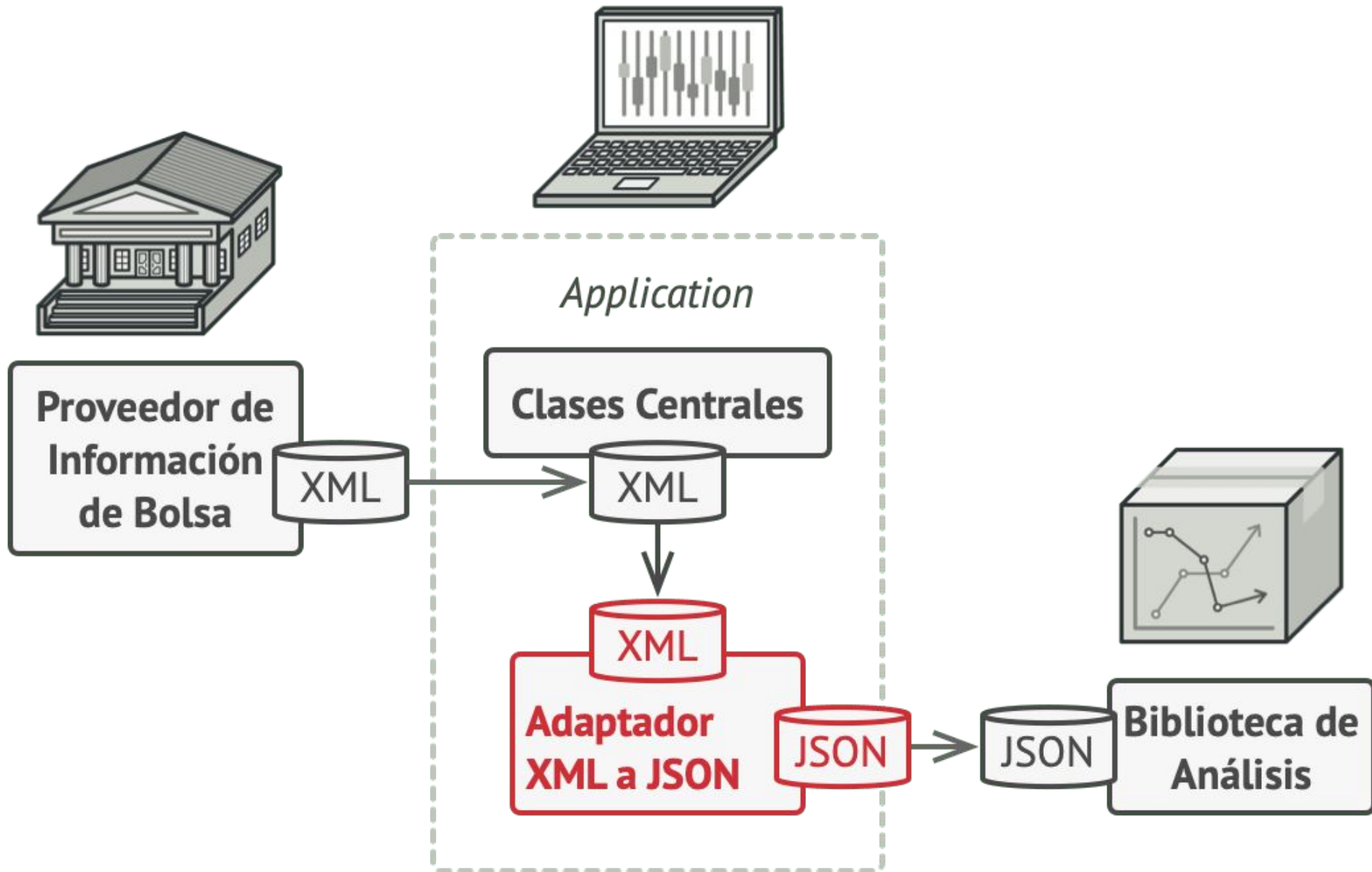
- ¿Cuándo se usa?
 - Cuando queremos modelar composiciones anidadas, por ejemplo un sistema de archivos, una jerarquía de empleados, etc.
- ¿Cómo se usa?
 - Se recae fuertemente en la herencia
 - Se tiene una clase básica, y otra clase que lleva un listado de clases básicas
 - Ambas clases implementan una interfaz
 - La clave está en que la clase que lleva el listado, tiene como variable interna una lista de objetos que implementen la interfaz definida anteriormente



Patrones de Diseño - Adapter

- ¿Cuándo se usa?
 - Queremos utilizar una clase, pero la forma en que trabaja no se adapta al sistema que ya tenemos
 - Por ejemplo tenemos un cargador con un determinado enchufe, pero la toma de corriente tiene otro enchufe → la solución es el adaptador
- ¿Cómo se usa?
 - Se implementa una clase Adapter que tenga como variable interna la clase que queremos utilizar originalmente
 - Dicha clase se encargará de abstraer la forma en que el sistema existente interactúa con la clase que originalmente era incompatible con nuestro sistema

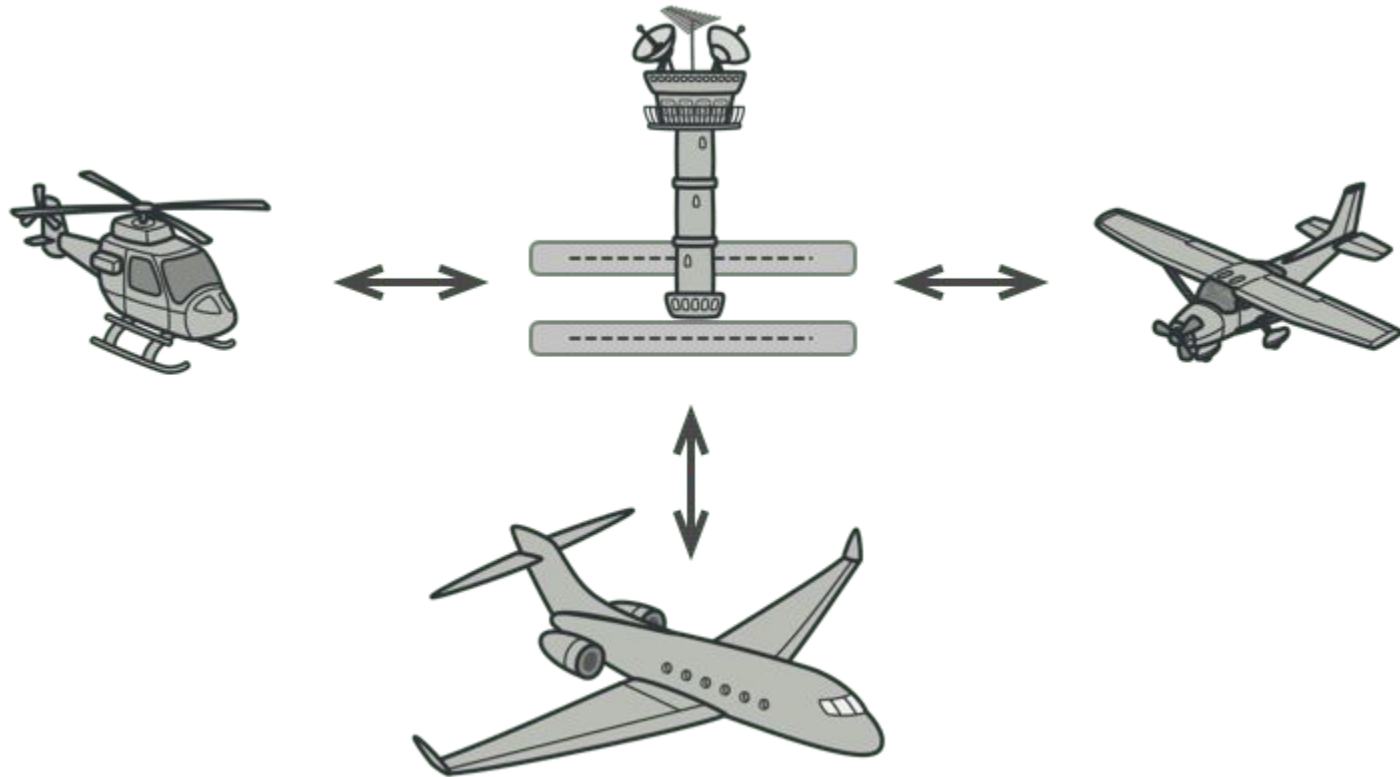




Patrones de Diseño - Mediator

- ¿Cuándo se usa?
 - Tenemos varios objetos que interactúan en reiteradas ocasiones → acoplamiento elevado
 - Queremos abstraer la interacción entre dichos objetos a fin de reducir la complejidad
- ¿Cómo se usa?
 - Implementamos una clase que contenga las instancias que originalmente interactúan
 - En la clase nueva se gestiona/simplifica la interacción a fin de abstraerla de cara al usuario

https://sourcemaking.com/design_patterns/mediator



Patrones de Diseño - Strategy

- ¿Cuándo se usa?
 - Tenemos varios algoritmos disponibles para resolver un mismo problema
- ¿Cómo se usa?
 - Usamos una interfaz, de la cual luego vamos a tener una clase por cada algoritmo disponible
 - Cada clase implementa la interfaz
 - Luego un método recibe como parámetro un objeto que implemente la interfaz

https://sourcemaking.com/design_patterns/strategy



€0
⌚⌚⌚



€€
⌚⌚



€€€
⌚

Recomendaciones

- Si bien los patrones son soluciones elegantes a ciertos problemas, no todos los problemas que tengamos se resuelven mediante un patrón
 - No forzarlos
- Una de las ideas de los patrones es que en caso de aplicar alguno, al momento de explicar nuestra solución, se nos facilita mucho
 - Tener en cuenta la justificación de por qué usamos cada patrón
 - Pensar la aplicación de cada patrón en términos de *cuál es el problema*, y qué *beneficios me traería* aplicar un patrón

Prog. Orientada a Objetos

Carrera Programador full-stack

Ejercicios

Ejercicios - Opcionales

Para todos los ejercicios plantear diagrama de clases. También usar tareas NPM y GitHub

Ejercicio 1

- Implementar un sistema de archivos
 - Investigar patrón *composite*

Ejercicio 2

- Extender clase Auto implementada anteriormente, para que soporte seis tipos de datos más → todos pasados en el constructor
 - Investigar patrón *builder*