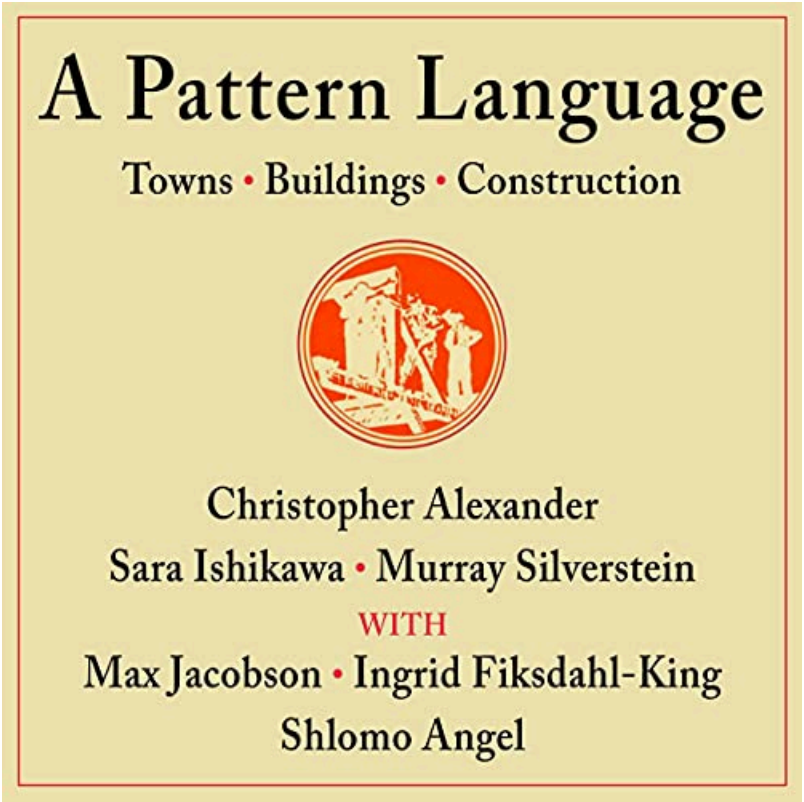
PATRONES DE DISEÑO

1. Introducción
2. Patrones del GoF
3. Otros patrones específicos

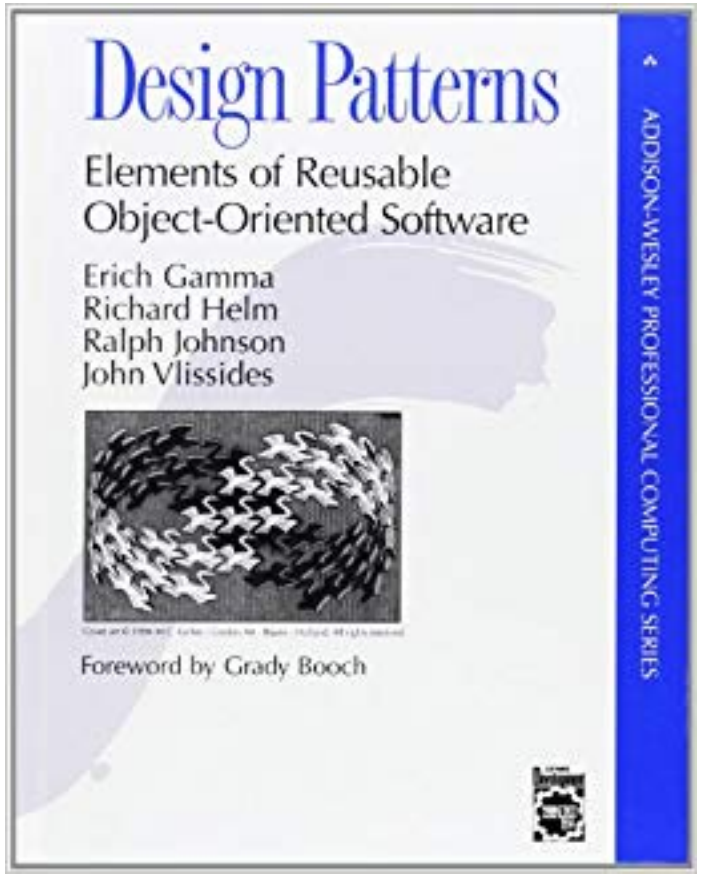
Introducción

Origen de los patrones de diseño



Diseño de software con patrones

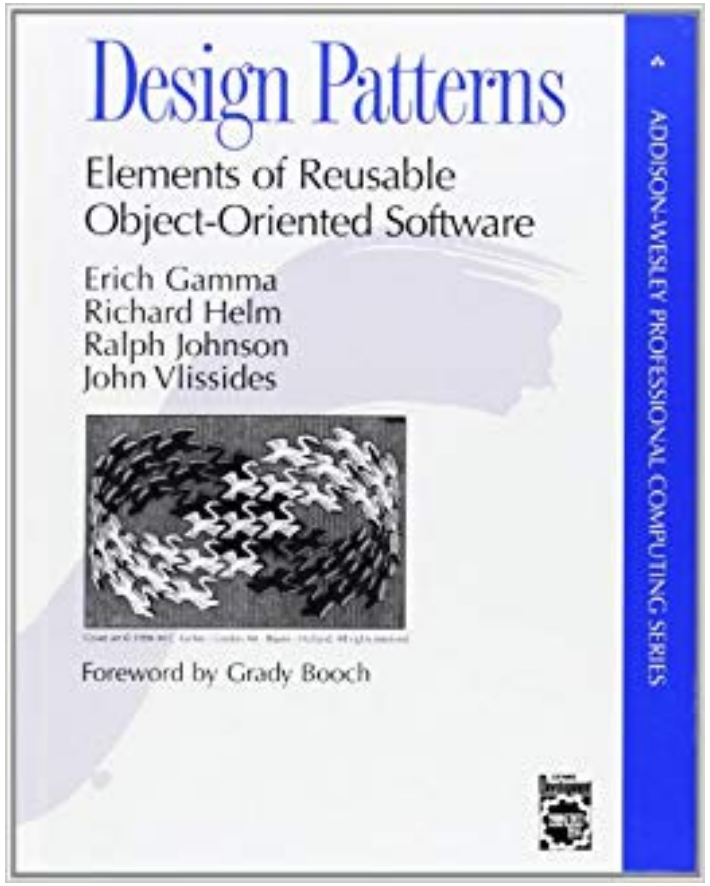
- Conocer un lenguaje OO no te hace un buen diseñador. ¿Qué diferencia hay entre los diseñadores expertos y los novatos? Los primeros usan recetas exitosas para los problemas habituales y no reinventan la rueda continuamente.
- Un grupo de expertos (*Gang of Four*) se basó en el trabajo de Alexander y lo aplicó al diseño de software, presentando el libro *Design Patterns* con un total de 23 patrones.



Patrones de diseño

- Patrón de diseño: Una **solución general** a un **problema general** que puede adaptarse a un problema concreto
- La aplicación de patrones depende del **contexto**.
- Ofrece un **vocabulario** de patrones (una jerga entre ingenieros de software)
- Los patrones **clásicos** son ampliamente conocidos: algunos muy aceptados y otros más discutidos...
- Deben usarse con cuidado. Deben simplificar el modelo, **no complicarlo**, por lo que deben surgir de manera natural.
- Han surgido nuevos patrones **específicos** de dominio: patrones de interfaces de usuario, patrones para la integración de aplicaciones empresariales, patrones de flujos de trabajo BPMN, patrones de concurrencia, etc.

Patrones del Gang of Four



Patrones creacionales

Corresponden a patrones de diseño de software que solucionan problemas de creación de instancias. Nos ayudan a encapsular y abstraer dicha creación. Vamos a ver:

- Factory Method
- Abstract Factory

Pero hay más...

- *Prototype*
- *Builder*
- *Singleton*

Patrones estructurales

Son los patrones de diseño software que solucionan problemas de composición/agregación de clases y objetos. Vamos a ver:

- Composite
- Decorator
- Adapter

Pero hay más...

- *Facade*
- *Bridge*
- *Flyweight*
- *Proxy*

Patrones de comportamiento

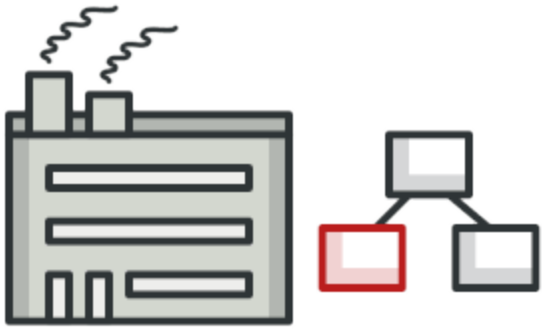
Son los relativos a la interacción y responsabilidades entre clases y objetos. Vamos a ver:

- Command
- Observer
- Strategy
- Visitor

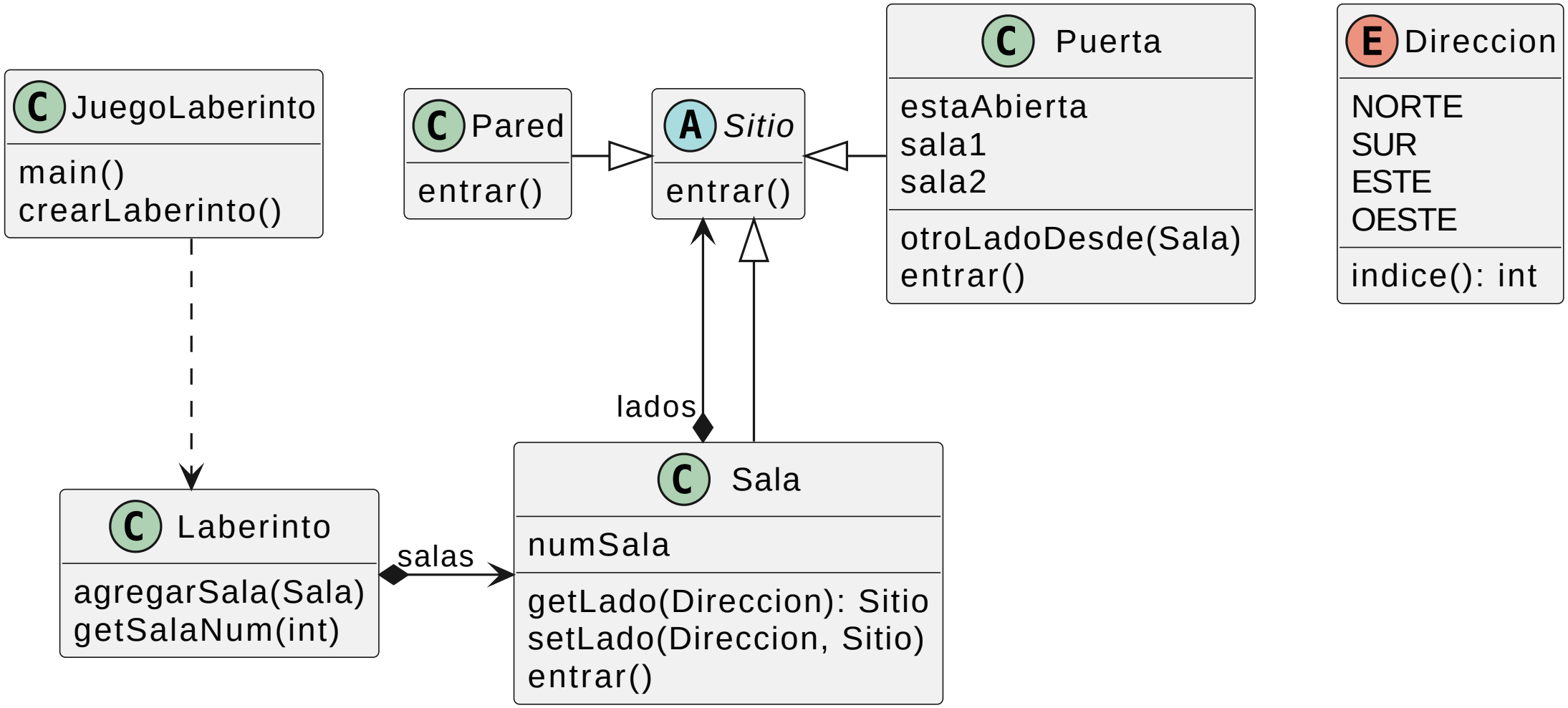
Pero hay más...

- *Template method*, *Chain of Responsibility*, *Interpreter*
- *Iterator*, *Mediator*, *Memento*, *State*

Factory Method



Ejemplo: Juego de laberinto



```java
public enum Direccion {

  NORTE(0), ESTE(1), SUR(2), OESTE(3);

  private final int indice;

  Direccion(int indice) {
    this.indice = indice;
  }

  int indice() {
    return indice;
  }

}
```

```java
public class Laberinto {
    Laberinto() {};
    void agregarSala(Sala sala) { ... };
    Sala getSalaNum(int numSala) { ... };
}
```

```java
public abstract class Sitio {
    boolean entrar() { ... };
}
```

```
public class Sala extends Sitio {
    private Sitio lados[];
    int numSala;

    Sala() {};
    Sala(int numSala) {};
    Sitio getLado(Direccion dir) {
      return lados[dir.indice()];
    };
    void setLado(Direccion dir, Sitio sitio) {
      lados[dir.indice()] = sitio;
    };
    boolean entrar() { ... };
}
```

```
public class Pared extends Sitio {
    Pared() {};
    boolean entrar() { ... };
}

public class Puerta extends Sitio {
    private Sala sala1;
    private Sala sala2;
    boolean estaAbierta;

    Puerta(Sala sala1, Sala sala2) { ... };
    boolean entrar() { ... };
    Sala otroLadoDesde(Sala unaSala) { ... };
}
```

¿Cómo se crean los laberintos?

```
Laberinto crearLaberinto () {
  Laberinto miLab = new Laberinto();
  Sala hab1 = new Sala(1);
  Sala hab2 = new Sala(2);
  Puerta unaPuerta = new Puerta(hab1, hab2);
  miLab.agregarSala(hab1);
  miLab.agregarSala(hab2);
  hab1.setLado(Direccion.NORTE, new Pared());
  hab1.setLado(Direccion.ESTE, unaPuerta);
  hab1.setLado(Direccion.SUR, new Pared());
  hab1.setLado(Direccion.OESTE, new Pared());
  hab2.setLado(Direccion.NORTE, new Pared());
  hab2.setLado(Direccion.ESTE, new Pared());
  hab2.setLado(Direccion.SUR, new Pared());
  hab2.setLado(Direccion.OESTE, unaPuerta);
  return miLab;
}
```

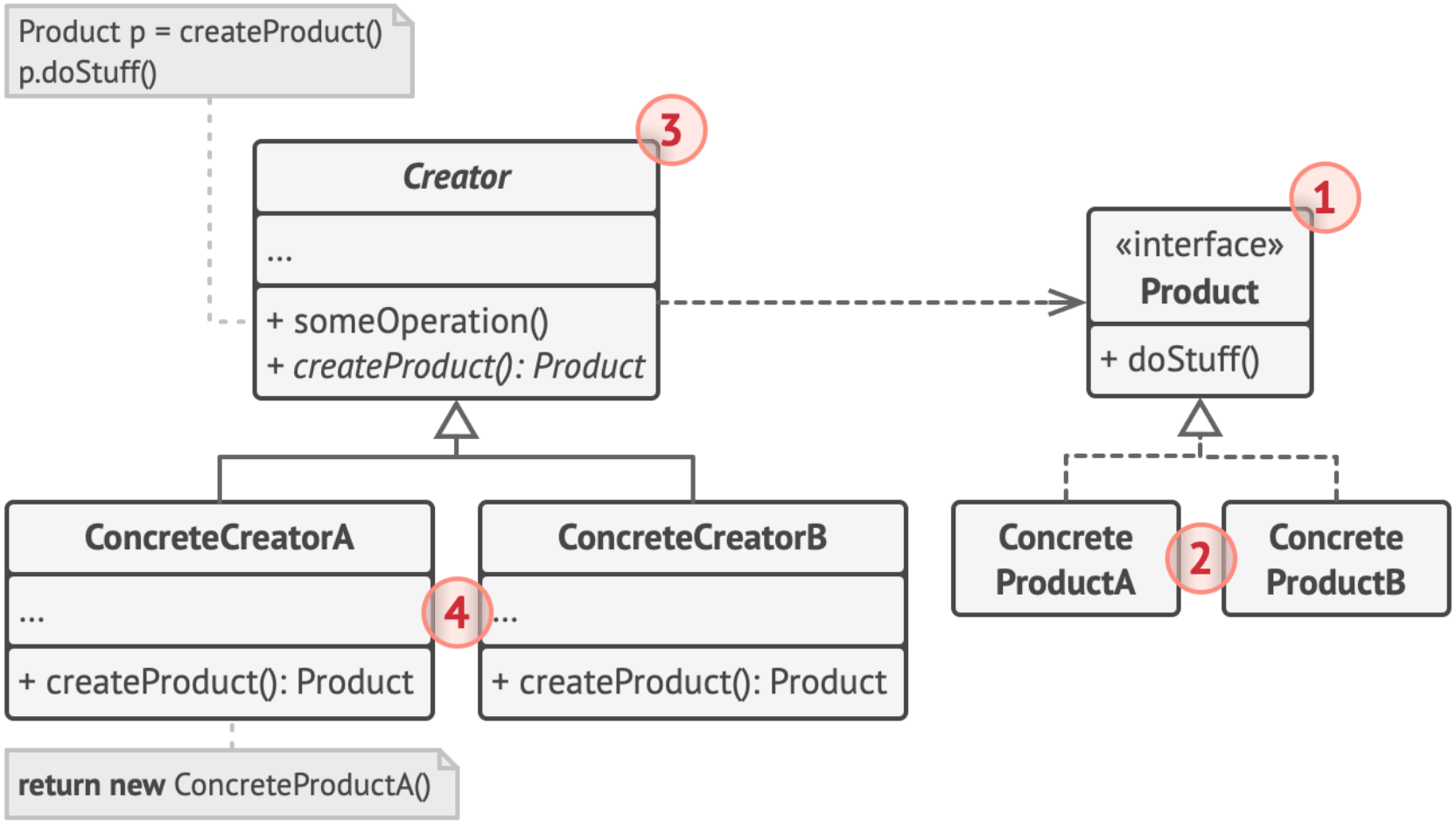
Críticas

- Creación poco flexible: instancias concretas cableadas.
- Supongamos $\exists$ `SalaHechizada`, `PuertaHechizada`. ¿Cómo cambiamos `crearLaberinto`?

Método de factoría

- El patrón *factory method* define una interfaz para la creación de un objeto, pero dejando en manos de las subclases la decisión de qué clase concreta instanciar.
- Permite que una clase delegue en sus subclases las instanciaciones.

Factory method: Estructura



1. El **Producto** declara la interfaz, que es común a todos los objetos que puede producir la clase creadora y sus subclases.
2. Los **Productos Concretos** son distintas implementaciones de la interfaz de producto.
3. La clase **Creadora** declara el método fábrica que devuelve nuevos objetos de producto. Es importante que el tipo de retorno de este método coincida con la interfaz de producto.
4. Los **Creadores Concretos** sobrescriben el Factory Method base, de modo que devuelva un tipo diferente de producto.

Factory method: Ventajas

- Se evita un acoplamiento fuerte entre el creador y los productos concretos.
- SRP: Se puede mover el código de creación de producto a un lugar del programa, haciendo que el código sea más fácil de mantener.
- OCP: Se pueden incorporar nuevos tipos de productos en el programa sin descomponer el código cliente existente.

Implementación de `JuegoLaberinto`

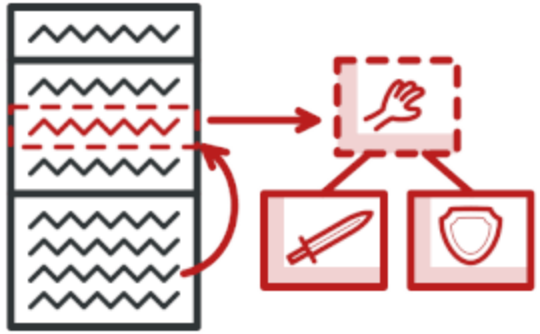
```
public class JuegoLaberinto {
  JuegoLaberinto() {};
  // factory methods:
  Laberinto makeLaberinto() { return new Laberinto(); }
  Sala makeSala(int numSala) { return new Sala(numSala); }
  Pared makePared() { return new Pared(); }
  Puerta makePuerta(Sala sala1, Sala sala2) {
    return new Puerta(sala1, sala2);
  }
  Laberinto crearLaberinto () { ... }
}
```

```
Laberinto crearLaberinto () {
  Laberinto miLab = makeLaberinto();
  Sala hab1 = makeSala(1);
  Sala hab2 = makeSala(2);
  Puerta unaPuerta = makePuerta(hab1, hab2);
  miLab.agregarSala(hab1);
  miLab.agregarSala(hab2);
  hab1.setLado(Direccion.NORTE, makePared());
  hab1.setLado(Direccion.ESTE, unaPuerta);
  hab1.setLado(Direccion.SUR, makePared());
  hab1.setLado(Direccion.OESTE, makePared());
  hab2.setLado(Direccion.NORTE, makePared());
  hab2.setLado(Direccion.ESTE, makePared());
  hab2.setLado(Direccion.SUR, makePared());
  hab2.setLado(Direccion.OESTE, unaPuerta);
  return miLab;
}
```

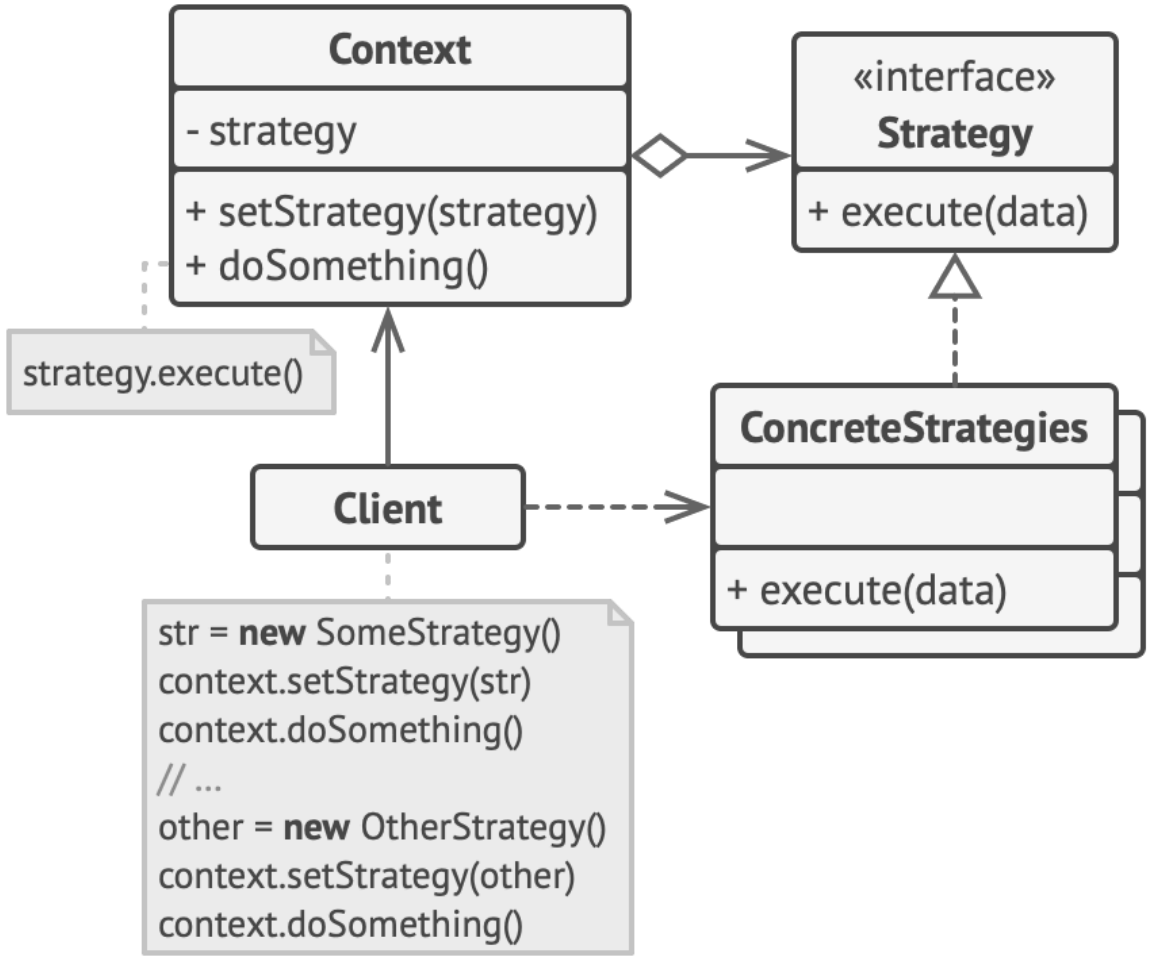
```
public class JuegoLaberintoMinado extends JuegoLaberinto {
  Pared makePared() {
    return new ParedMinada();
  }
  Sala makeSala(int numSala) {
    return new SalaMinada(numSala);
  }
}

public class JuegoLaberintoHechizado extends JuegoLaberinto {
  Sala makeSala(int numSala) {
    return new SalaHechizada(numSala, lanzarHechizo());
  }
  Puerta makePuerta(Sala sala1, Sala sala2) {
    return new PuertaHechizada(sala1, sala2);
  }
  private Hechizo lanzarHechizo() { ... }
}
```

Strategy



Strategy: Estructura



Strategy

- Define una familia de algoritmos, encapsula cada uno de ellos y los hace intercambiables
- Permite que el algoritmo varíe de forma independiente a quienes lo usan (el **Contexto**)

Ventajas:

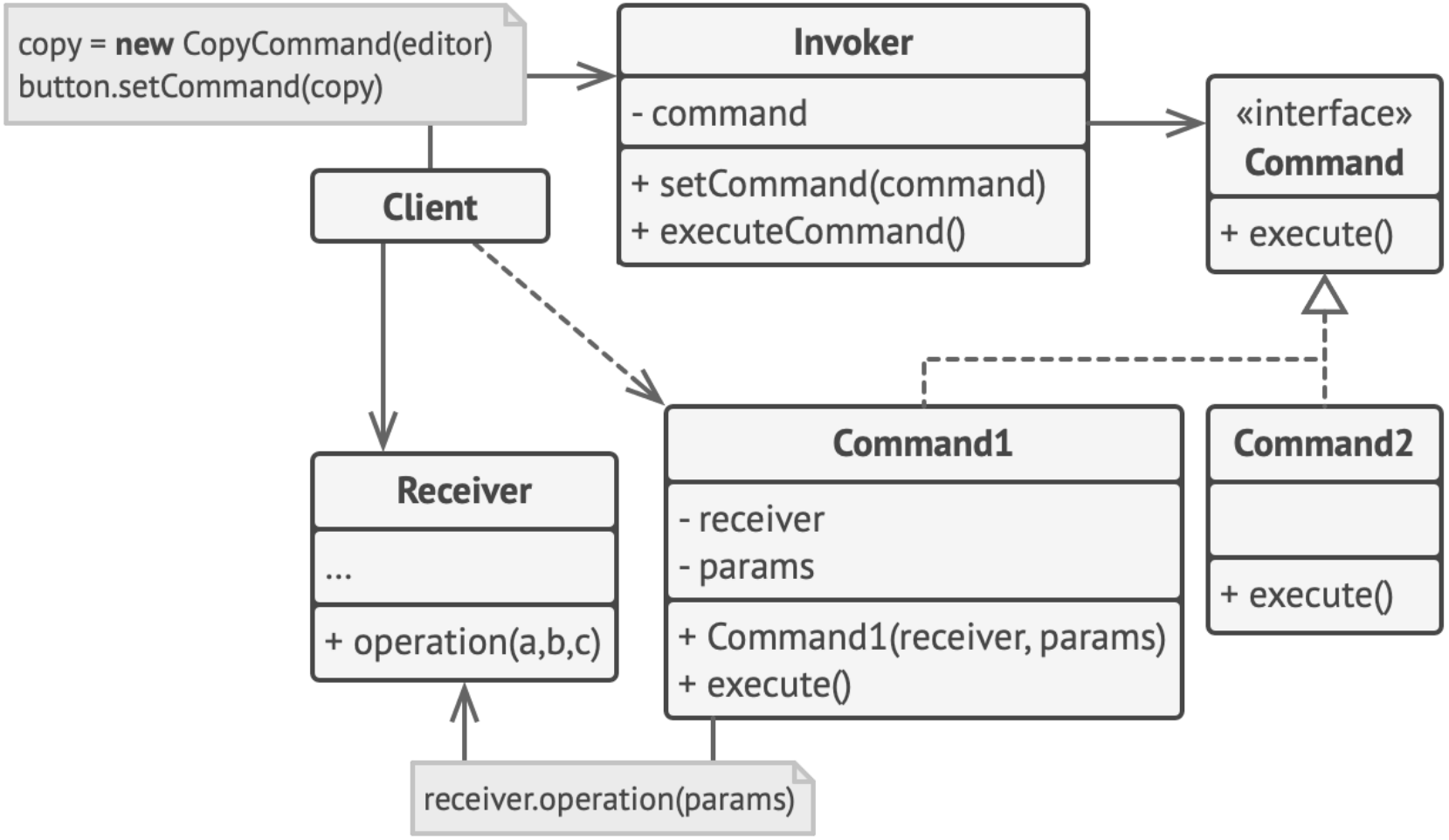
- Ayuda a sacar factor común (factorizar) funcionalidades
- La estrategia es sustituible en tiempo de ejecución
- Alternativa a la herencia estática

Desventajas:

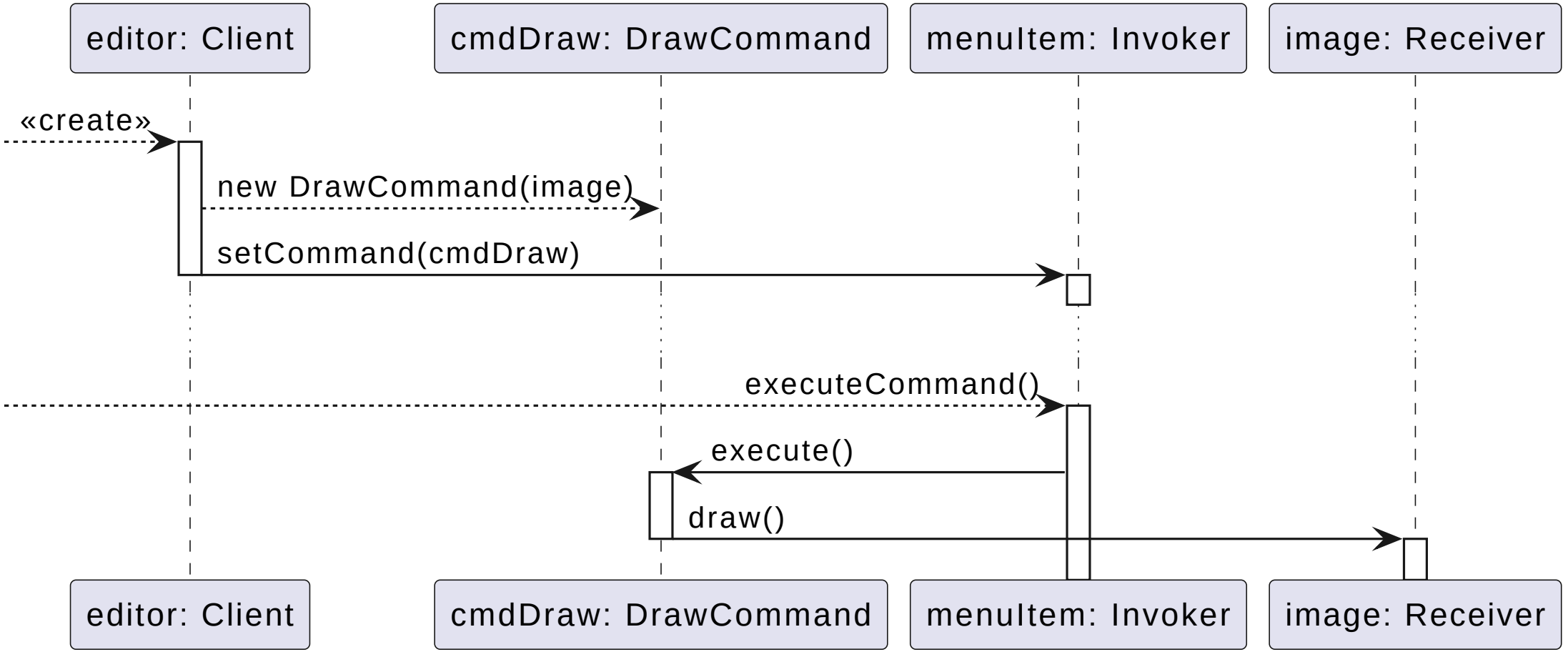
- Sobrecarga de la comunicación *Context-Strategy*

Command

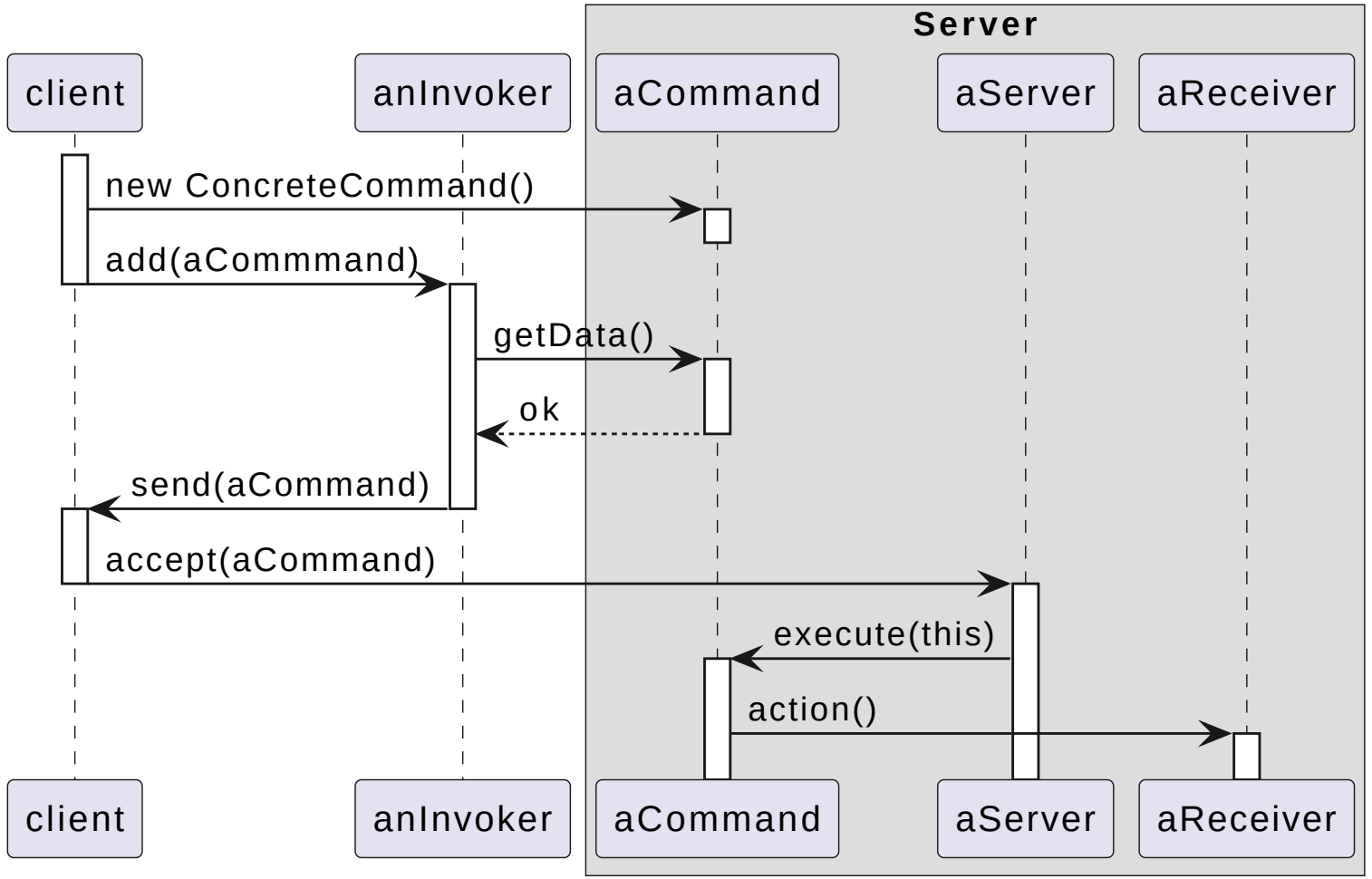
Command: Estructura



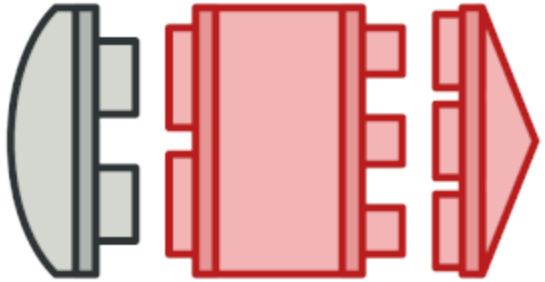
Command: Comportamiento



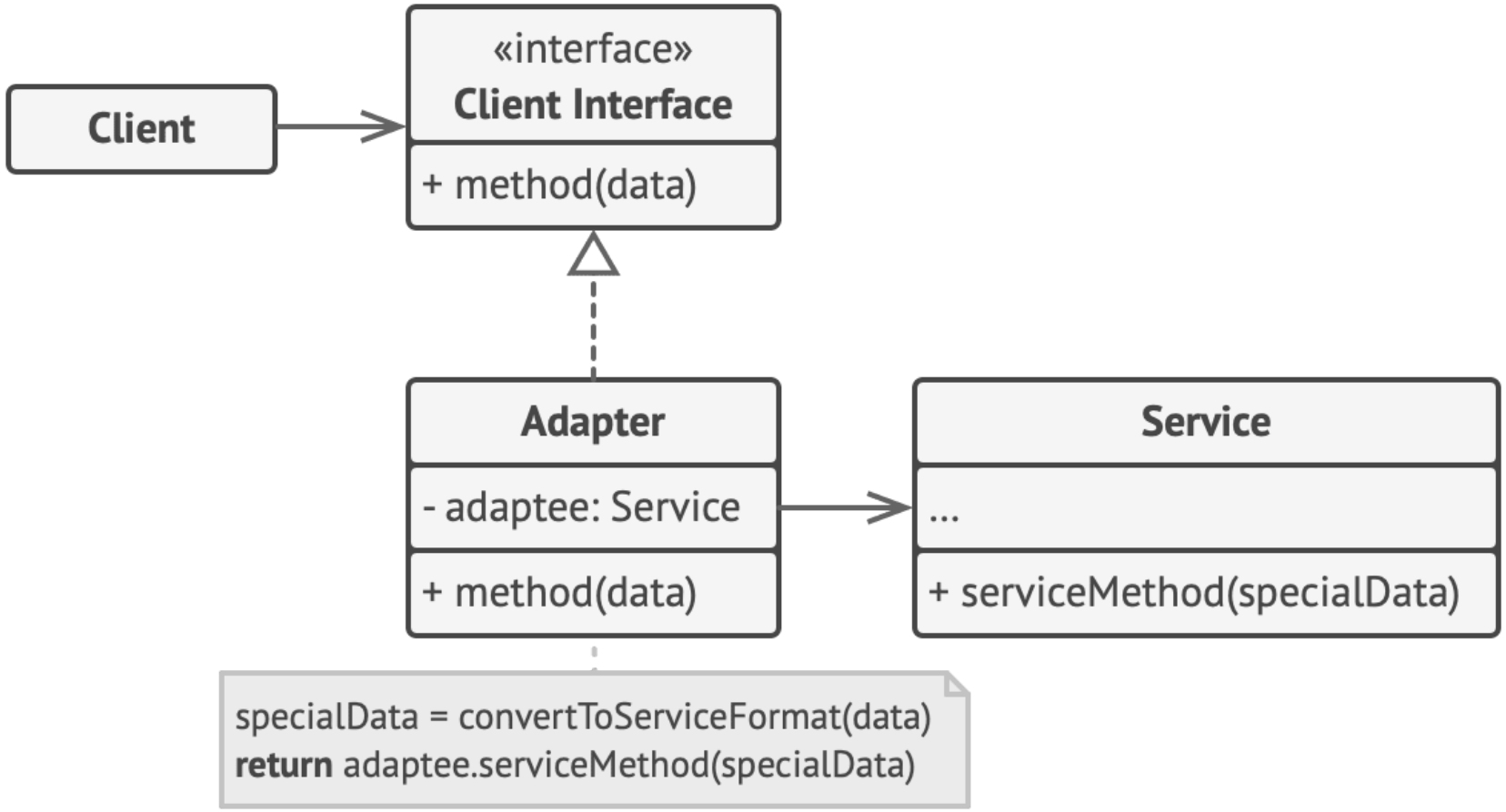
Versión cliente/servidor



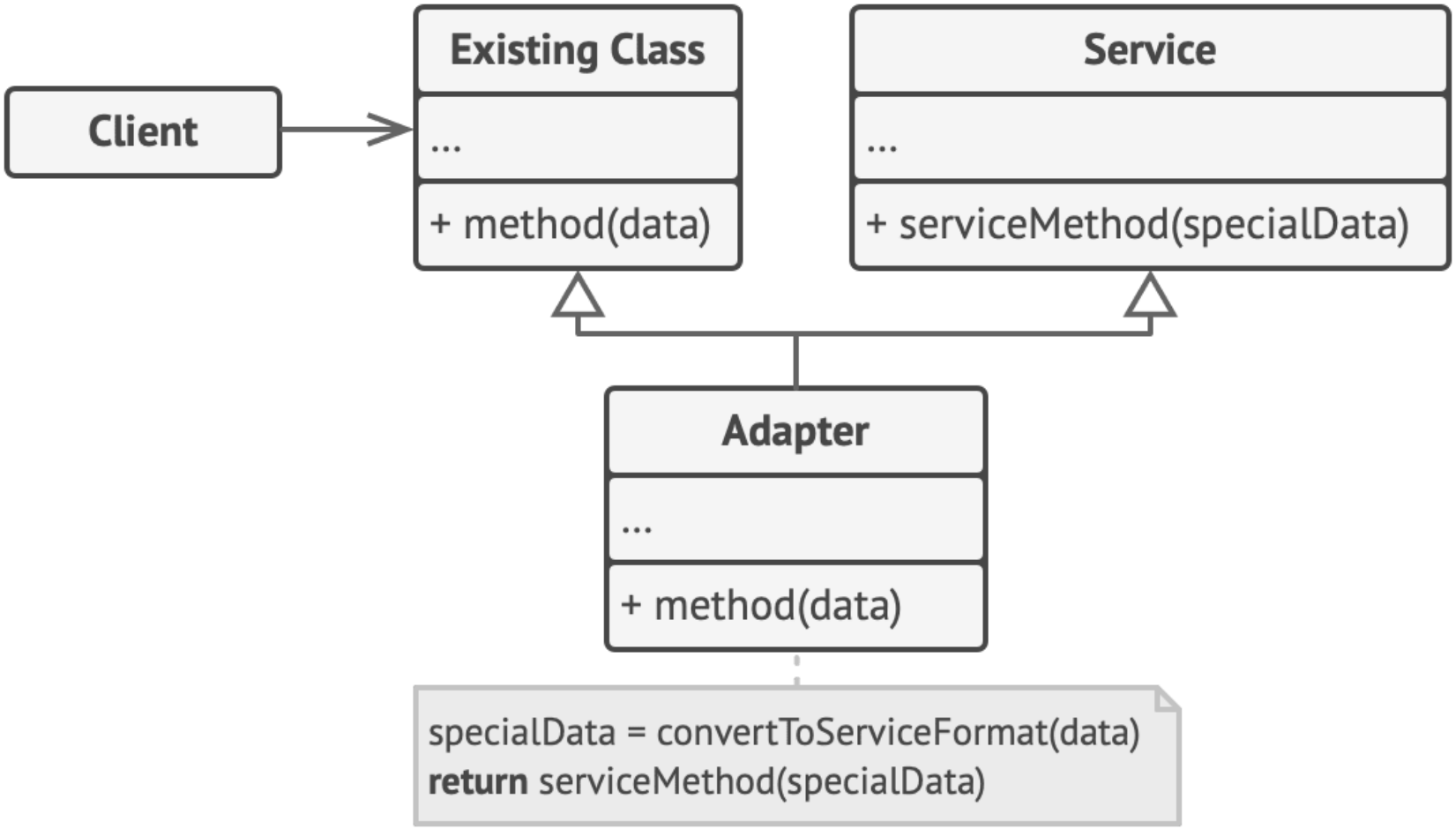
Adapter



Adaptador de objetos: Estructura



Adaptador de clases: Estructura



Adaptador de clases vs. objetos

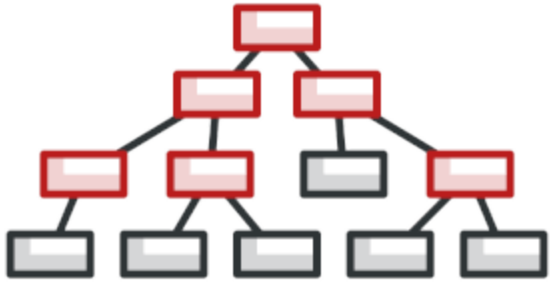
Class adapter:

- No sirve para adaptar una clase y sus subclases
- Se crea un único objeto, sin indirecciones adicionales

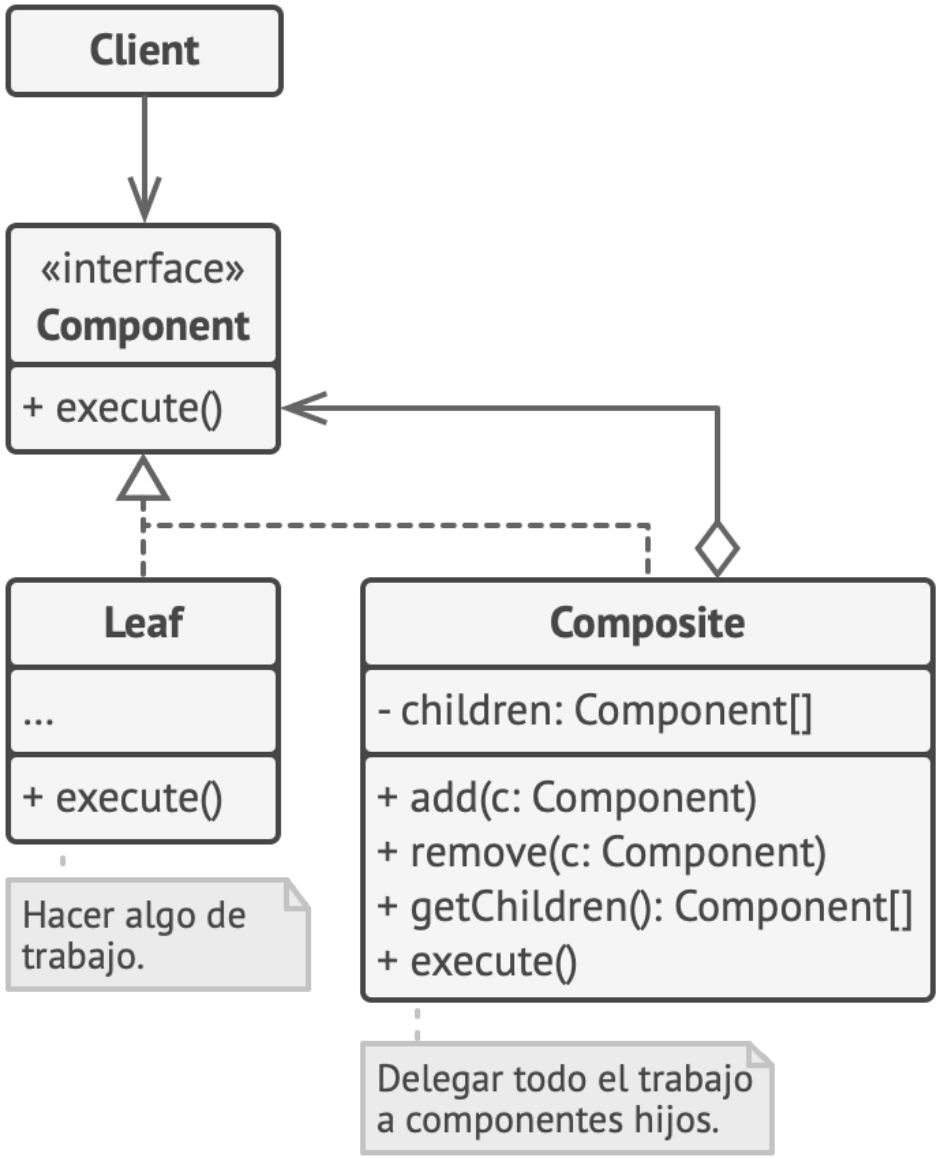
Object adapter:

- Un adapter puede funcionar con varios objetos *Service* o *Adaptee*
- Es más complicado heredar el comportamiento del objeto adaptado

Composite



Composite: Estructura



Composite

- Permite construir objetos complejos componiendo de forma recursiva objetos similares en una estructura de **árbol**.
- Permite manipular **uniformemente** todos los objetos contenidos en el árbol, ya que todos ellos poseen una interfaz común definida en la clase raíz.

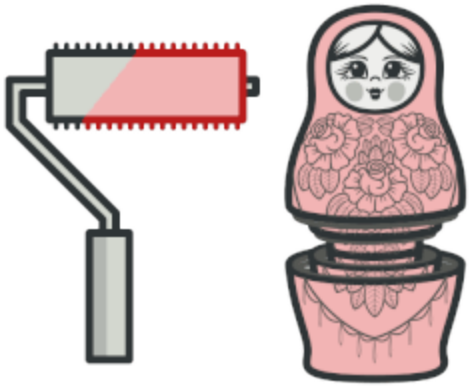
Ventajas:

- El cliente trata a todos los objetos de la misma forma
- La inclusión de nuevos tipos de hojas o compuestos no afecta a la estructura anterior

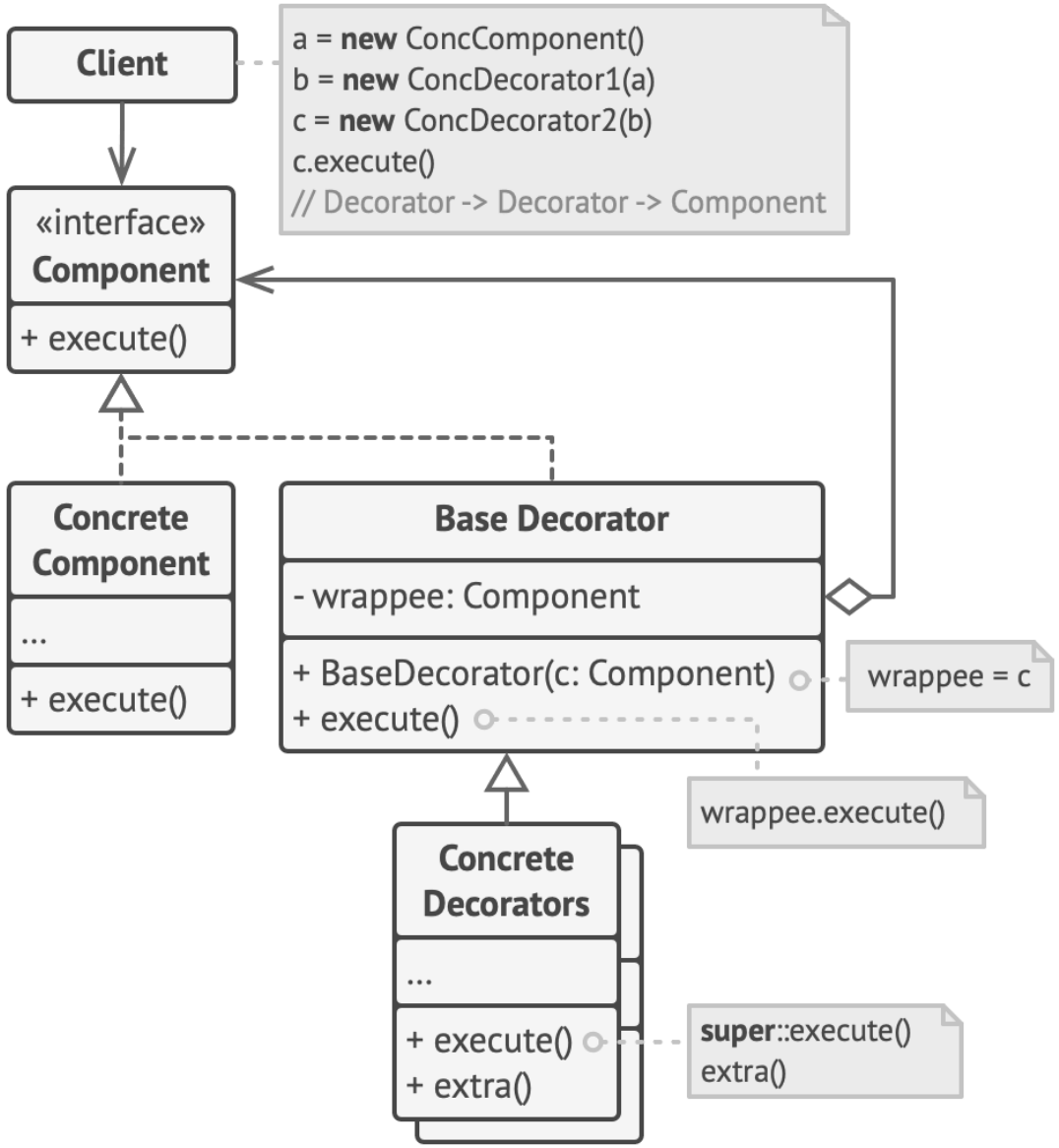
Desventajas:

- Si se desea restringir el tipo de objetos que pueden formar parte de otros $\Rightarrow$ Necesidad de comprobaciones dinámicas

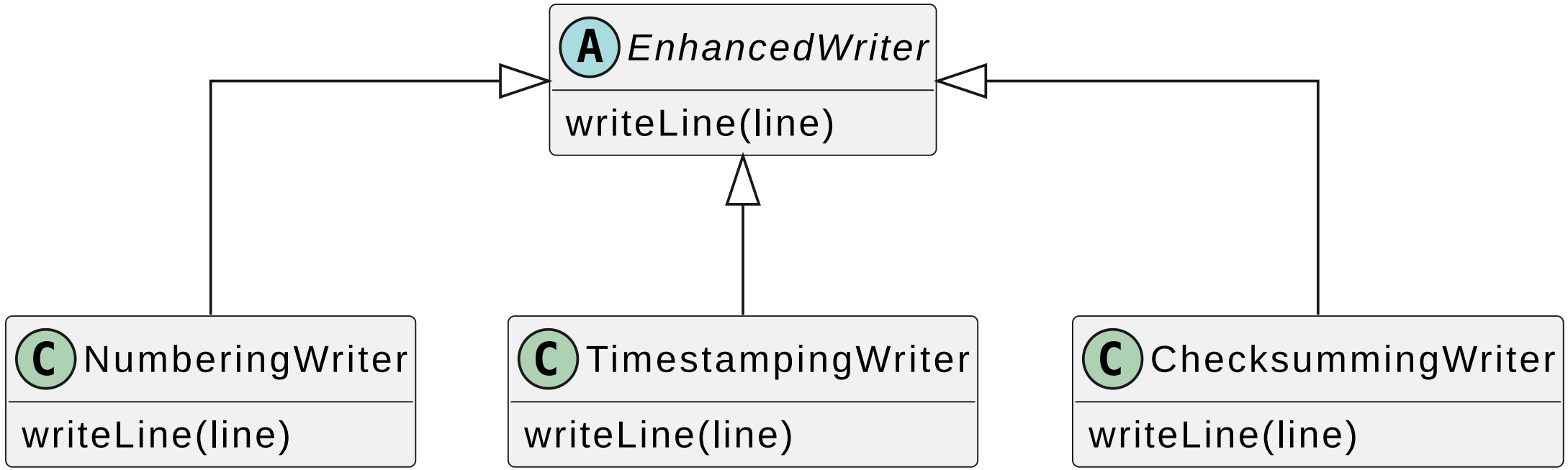
Decorator



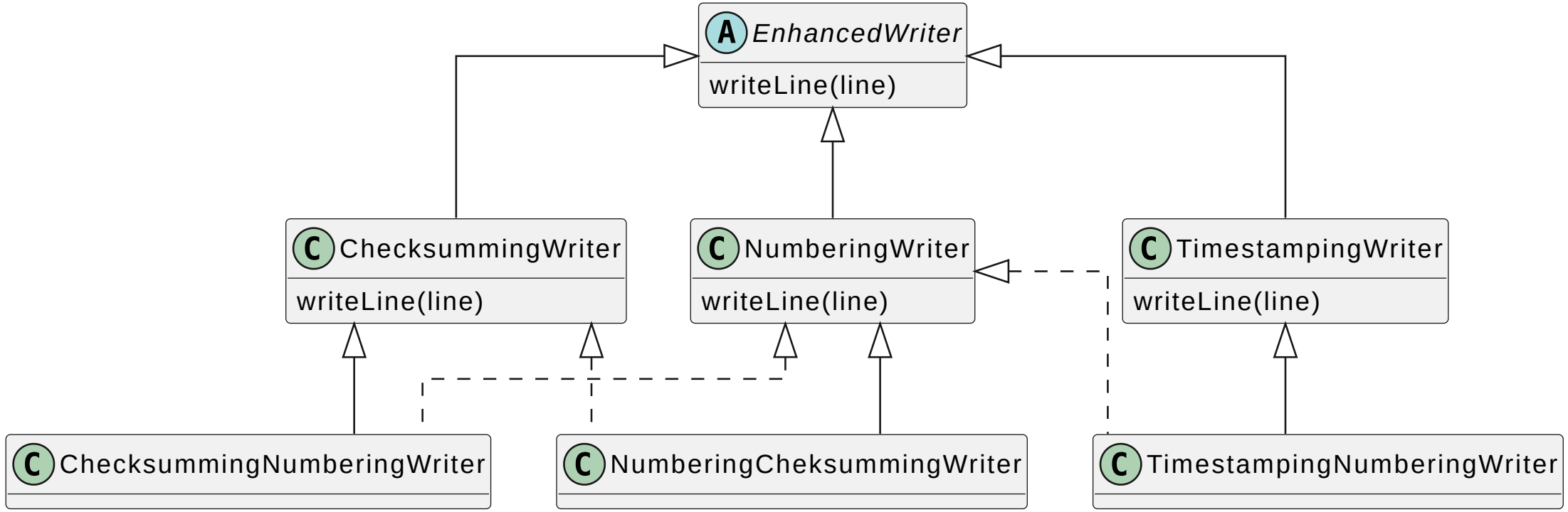
Decorator: Estructura



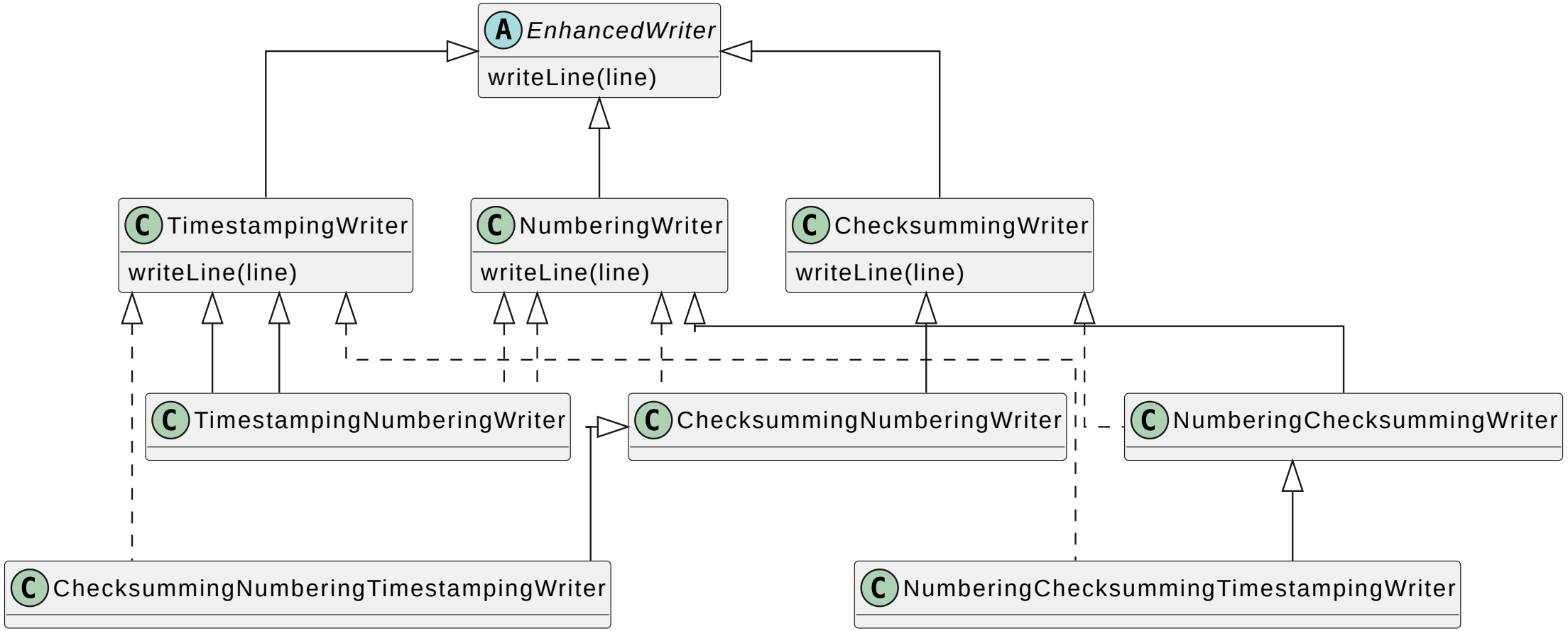
Ejemplo: `EnhancedWriter` original



Ejemplo: `EnhancedWriter` ampliado – herencia fuera de control



Ejemplo: `EnhancedWriter` ampliado – herencia fuera de control



Decorator

- El patrón decorator permite añadir responsabilidades a objetos concretos de forma **dinámica**.
- Los decoradores ofrecen una **alternativa** más flexible que la herencia para extender funcionalidades.

Ventajas:

- Permite añadir o quitar responsabilidades a los objetos sin afectar a otros objetos

Desventajas:

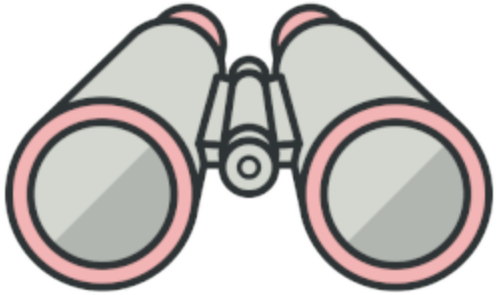
- Rompe la identidad de objetos: un componente y su decorador no son el mismo objeto
- Provoca la creación de muchos objetos pequeños y complica la depuración

¿Diferencia entre Strategy y Decorator?

Diferencia entre Strategy y Decorator

El *decorator* cambia la piel, el *strategy* cambia las tripas

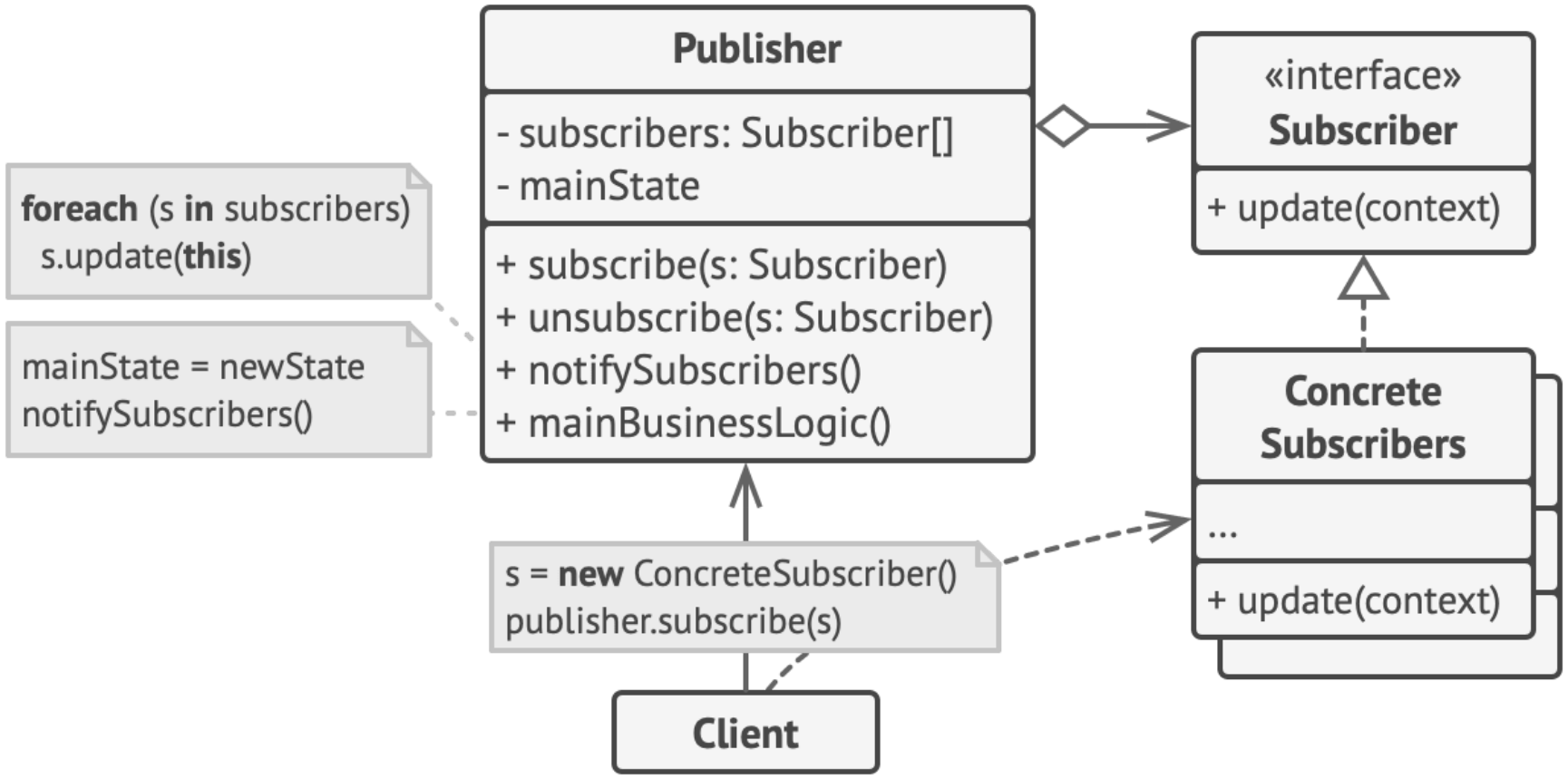
Observer



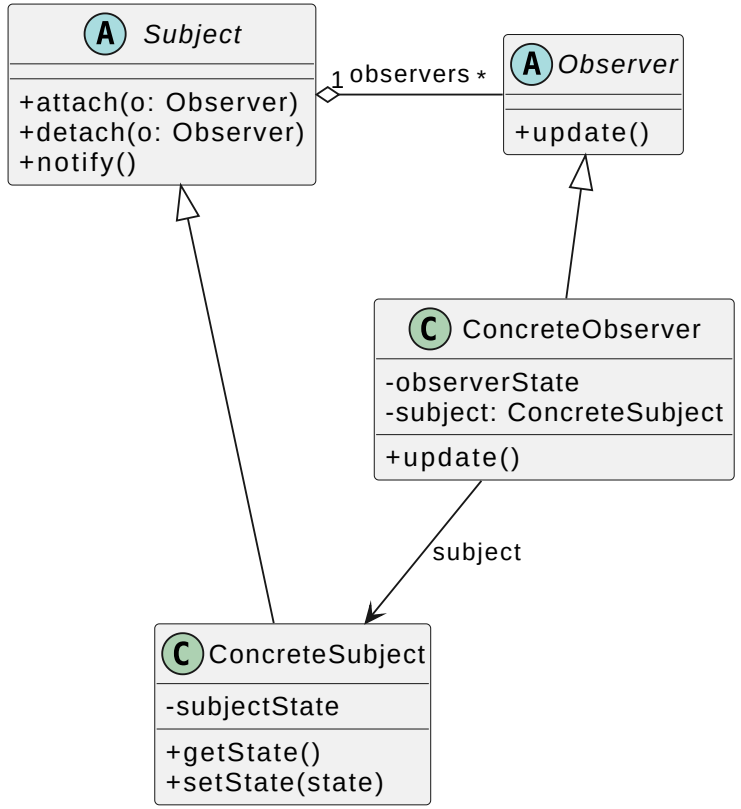
Observer

- Define una dependencia 1:N entre objetos de modo que cuando el estado de un objeto cambia, se les notifica el cambio a todos los que de él dependen y estos se actualizan de forma automática.

Observer: Estructura



Observer: Estructura (según GoF)



Observer: roles

Sin distinguir entre `Observable` y `Subject`:

- `Publisher` = `Observable` = `Subject`
- `ConcreteSubscriber` `--→` `Subject`

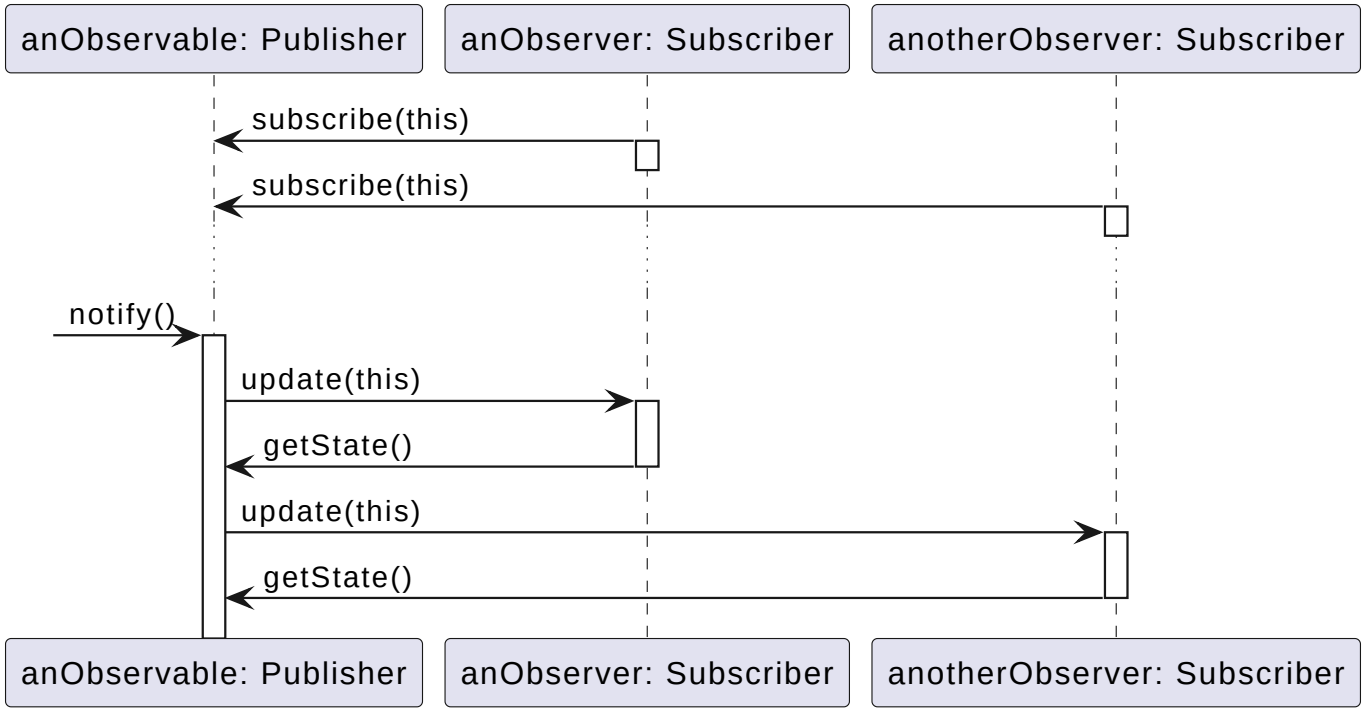
Con `Subject` separado:

- `Subscriber` = `Observer`
- Distinguir entre `Observable` y `Subject`
- Definir `Observer.update()`
  - `ConcreteObserver` mantiene referencia a `ConcreteSubject`
  - El estado se recupera desde `ConcreteSubject.getState()` (pull)

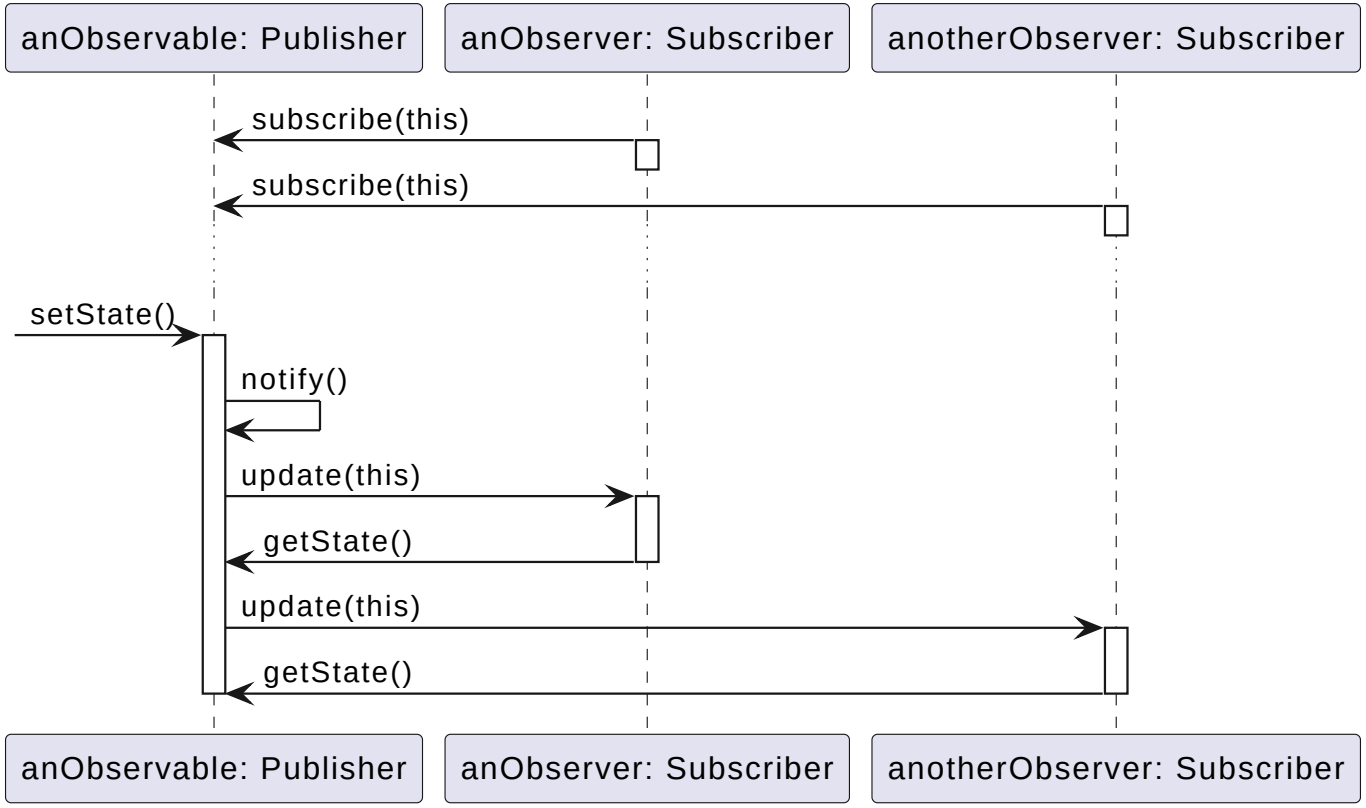
Observer: Detalles de implementación

- ¿Quién dispara la actualización?
  - El publicador, tras cambiar de estado: menos eficiente si hay muchas notificaciones
  - El cliente, tras una serie de cambios de estado: si se olvida puede provocar inconsistencias

Observer: Comportamiento (síncrono) – disparo externo



Observer: Comportamiento (síncrono) – autodisparo



Observer: Detalles de implementación

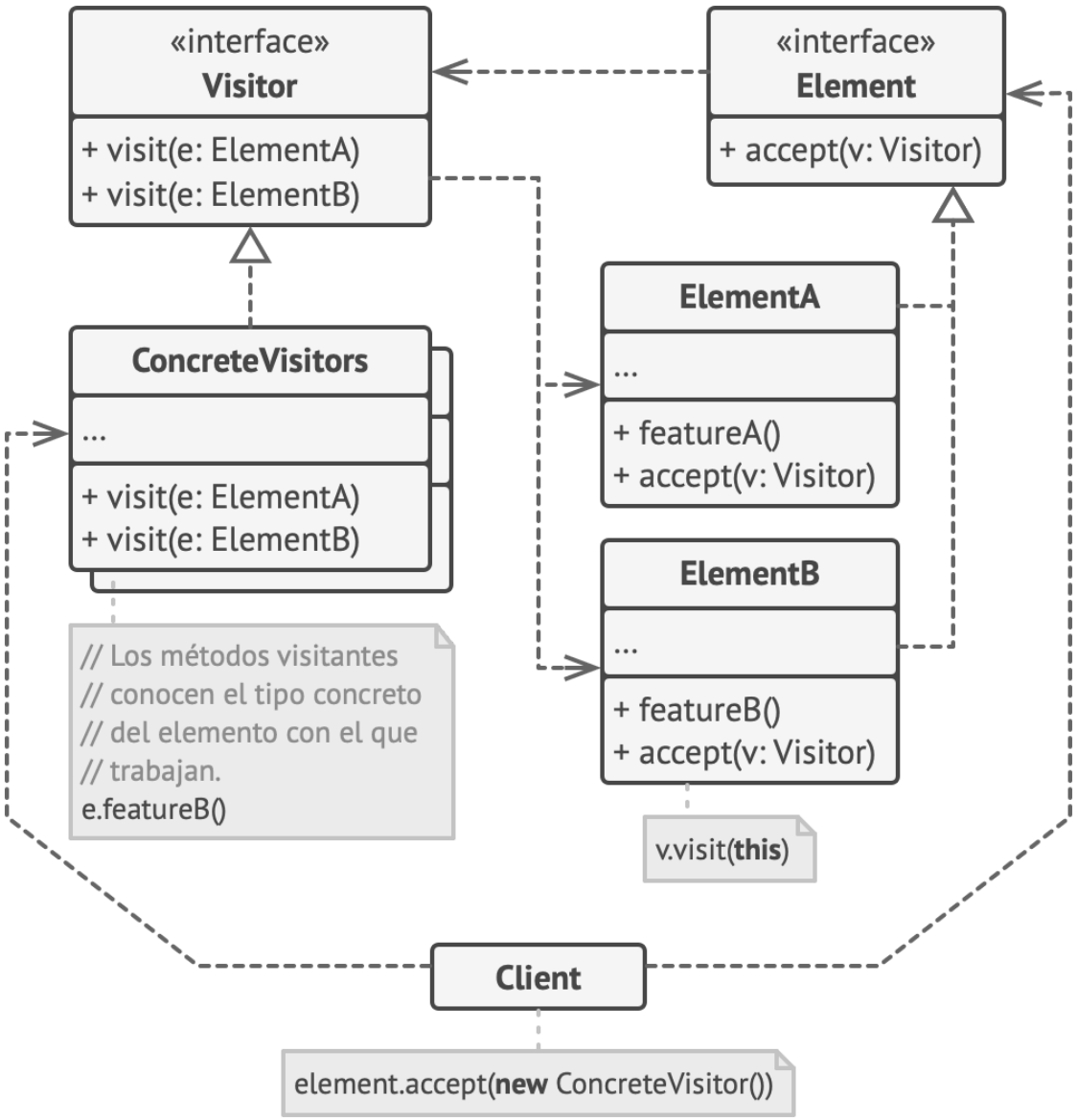
- Los suscriptores necesitan información para hacer la actualización:
  - `update(context)` para pasar la información necesaria al suscriptor
  - `update(this)` para que el suscriptor extraiga la información necesaria pidiéndosela al publicador
  - `ConcreteSubscriber.setPublisher()` para vincularlos permanentemente (opción menos flexible)

Visitor

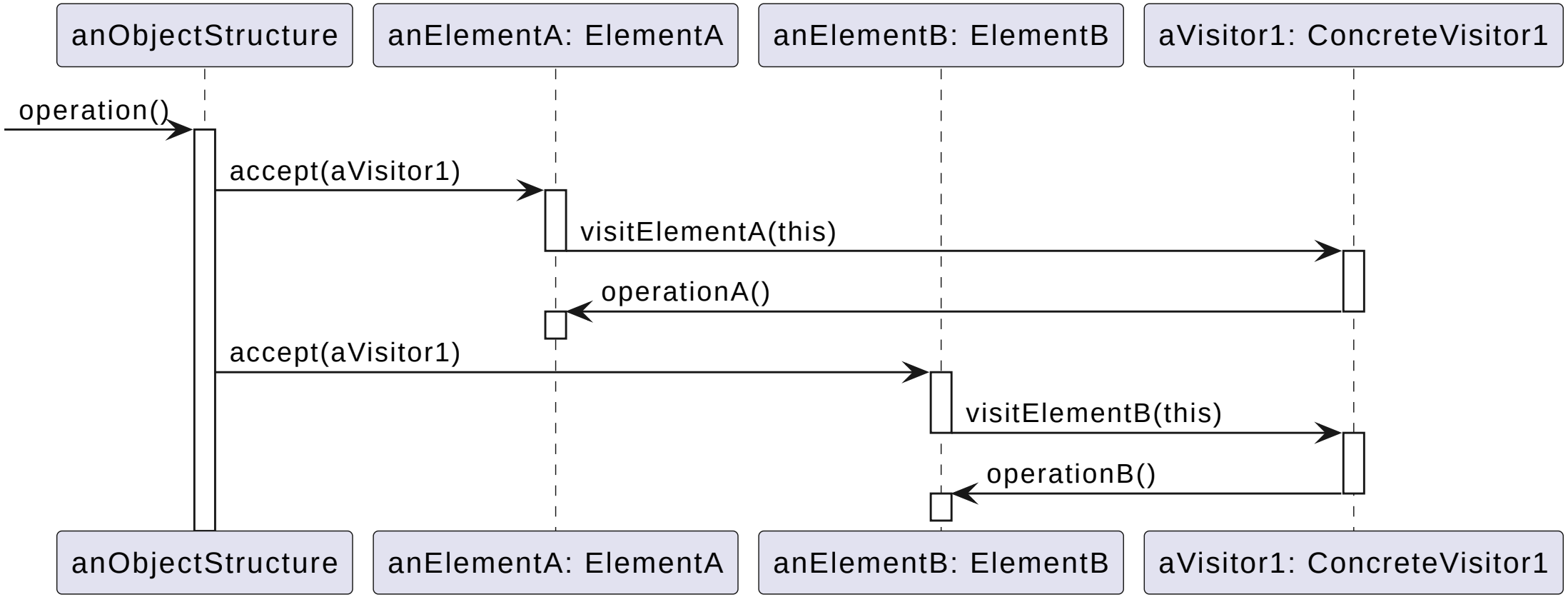
Visitor

- Representa una **operación** que se lleva a cabo sobre los elementos de una **estructura** de objetos
- Permite **definir nuevas** operaciones sin modificar las clases de los **elementos** sobre las que opera.

Visitor: Estructura



Visitor: Comportamiento



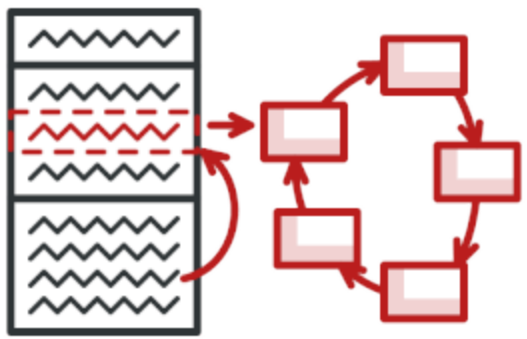
Ventajas:

- Permite implementar el *double dispatch*: la operación que se ejecuta tras el `accept()` depende del tipo de `Visitor` y del tipo de `Element`
- Separa los datos y las operaciones de los elementos visitados, facilitando la inclusión de nuevas operaciones sin tener que cambiar las clases
- Permite acumular el estado de una operación global sobre una estructura

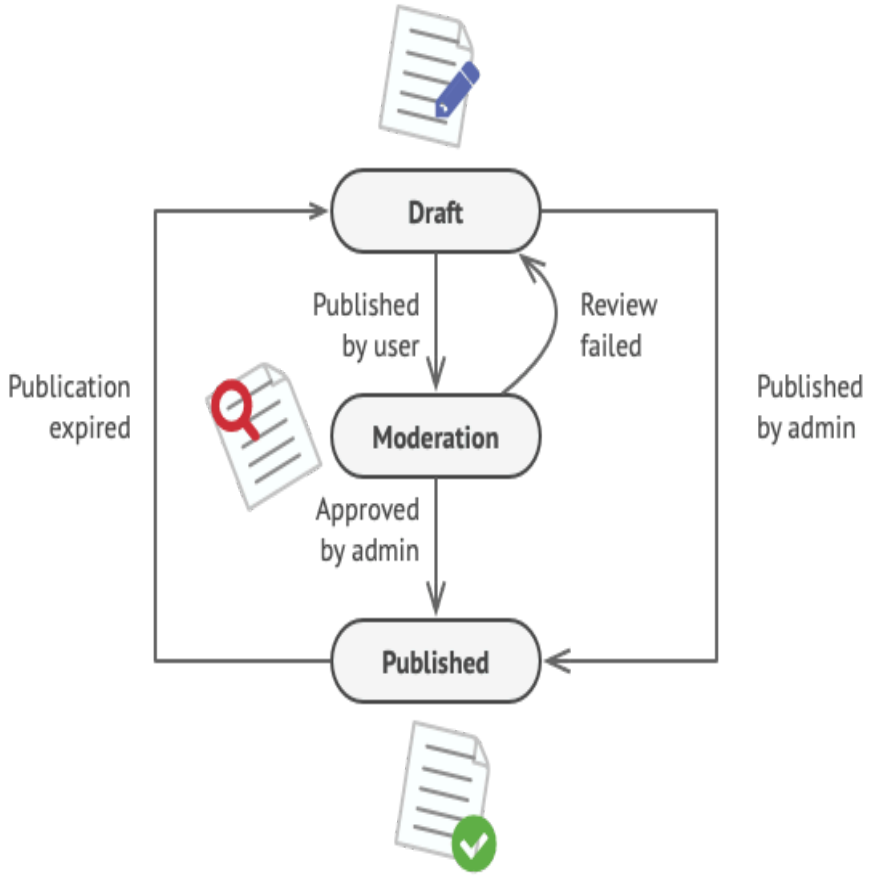
Desventajas:

- Rompe la encapsulación (?)
- Los tipos de `Element` visitados deben ser estables

State



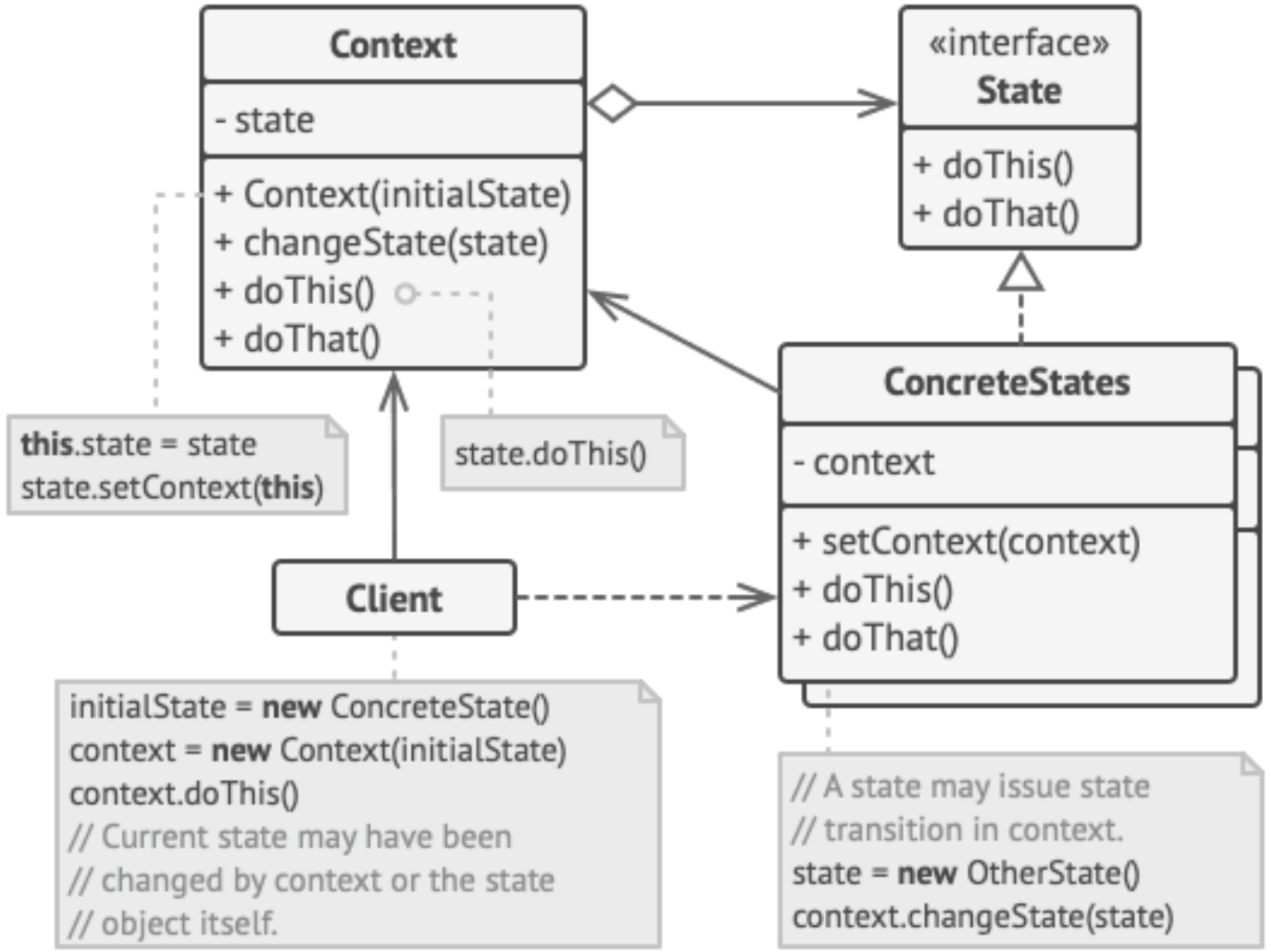
State: Ejemplo




```
class Document {
  var state: String = _
  var expirationDate: Date = _
  // ...

  def publish(currentUser: User): Unit = state match {
    case "draft" =>
      if (currentUser.role == "admin") {
        state = "published"
      } else {
        state = "moderation"
      }
    case "moderation" =>
      if (currentUser.role == "admin") {
        state = "published"
      }
    case "published" =>
      if (new Date().after(expirationDate)) {
        state = "draft"
      }
    case _ =>
      // Handle any unexpected state.
      throw new IllegalStateException(s"Invalid document state: $state")
  }
}
```

State: Estructura



Diferencia de State con Strategy

- Cada estado puede ser consciente de la existencia de otros estados e iniciar transiciones de estado
- Cada estrategia desconoce a las otras

Otros patrones específicos

Patrón *Active Record*

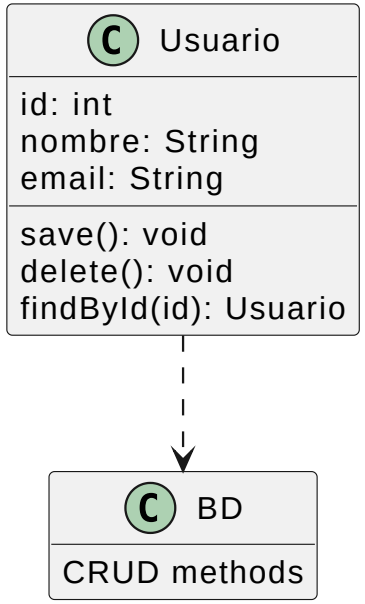
El objeto de dominio mantiene el **estado** e incorpora la **lógica de persistencia** sobre ese estado.

- La entidad representa los datos del dominio y expone operaciones CRUD (`save()`, `delete()`, etc.)
- Lógica de acceso a datos acoplada al modelo persistente

Desventajas:

- **Mezcla de responsabilidades** en la misma clase
- **Acoplamiento** al ORM o a la base de datos
- **Pruebas más difíciles**: cuesta el *mocking*
- **Escala peor** cuando aparecen reglas complejas

Estructura



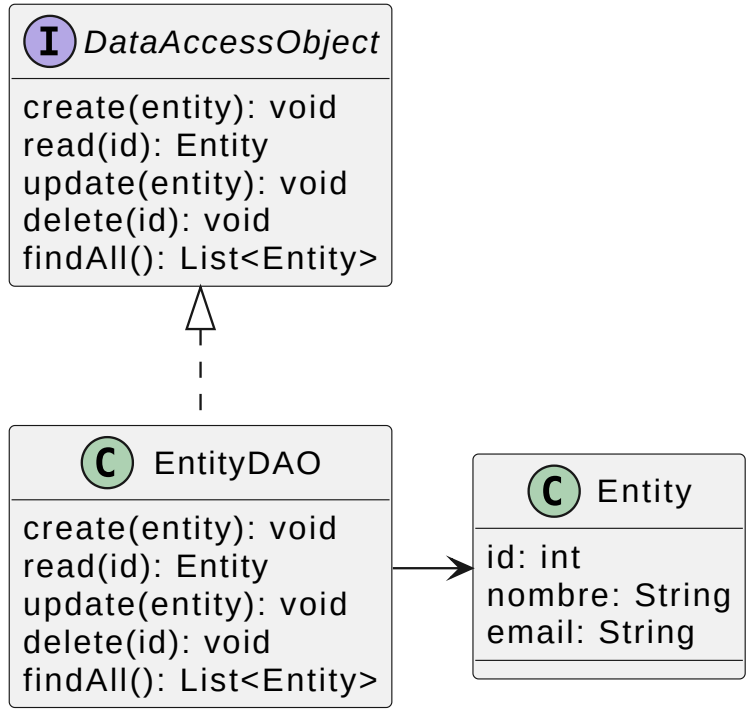
Patrón *Data Access Object* (DAO)

El patrón **Data Access Object** se usa para abstraer y encapsular los accesos a las fuentes de datos, proporcionando una **capa de persistencia** con independencia del soporte concreto de almacenamiento (BD relacional, NoSQL, ficheros, etc.).

Problemas (sin DAO):

- Lógica de acceso a datos dispersa por la aplicación
- Acoplamiento dependencias concretas (JDBC, Hibernate, etc.) en el código de negocio
- Difícil cambiar la implementación de persistencia sin modificar todo el código

DAO: Estructura



Ejemplo: Usuario DAO (enfoque clásico)

```java
// Interfaz DAO - define el contrato
public interface UsuarioDAO {
  void create(Usuario usuario);
  Usuario read(int id);
  void update(Usuario usuario);
  void delete(int id);
  List<Usuario> findAll();
}

// Implementación JDBC
public class UsuarioDAOImpl implements UsuarioDAO {
  private Connection connection;

  public UsuarioDAOImpl(Connection connection) {
    this.connection = connection;
  }

  @Override
  public void create(Usuario usuario) {
    String sql =
      "INSERT INTO usuarios (nombre, email) VALUES (?, ?)";
    try (PreparedStatement stmt =
            connection.prepareStatement(sql)) {
      stmt.setString(1, usuario.getNombre());
      stmt.setString(2, usuario.getEmail());
      stmt.executeUpdate();
    } catch (SQLException e) {
      throw new PersistenceException(e);
    }
  }
  ...
```

```java
  ...
  @Override
  public Usuario read(int id) {
    String sql =
      "SELECT * FROM usuarios WHERE id = ?";
    try (PreparedStatement stmt =
            connection.prepareStatement(sql)) {
      stmt.setInt(1, id);
      try (ResultSet rs = stmt.executeQuery()) {
        if (rs.next()) {
          return new Usuario(
            rs.getInt("id"),
            rs.getString("nombre"),
            rs.getString("email")
          );
        }
      }
    } catch (SQLException e) {
      throw new PersistenceException(e);
    }
    return null;
  }

  // ... otros métodos (update, delete, findAll)
}
```

DAO: Ventajas y Desventajas

Ventajas:

- **Separación de responsabilidades**: lógica de negocio desacoplada del acceso a datos
- **Cambios de persistencia**: cambiar BD sin modificar la lógica de negocio
- **Testabilidad**: fácil crear mocks del DAO para pruebas unitarias
- **Centralización**: código de SQL/queries concentrado en una única ubicación

Desventajas:

- **Boilerplate**: mucho código repetitivo (CRUD methods en cada DAO)
- **Mantenimiento**: cambios en la entidad requieren actualizar el DAO
- **Inflexibilidad**: las consultas complejas requieren nuevos métodos en la interfaz

Patrón *Repository*

Evolución moderna del DAO que surge con la popularidad de frameworks como *Spring Data JPA*. Ofrece una abstracción de más alto nivel que el DAO tradicional, proporcionando operaciones CRUD genéricas sin necesidad de implementación manual.

Relación Repository $\leftrightarrow$ DAO:

- Repository = DAO de alto nivel + convenciones + derivación de consultas
- Ambos abstraen la persistencia, pero Repository reduce boilerplate

Repository vs DAO

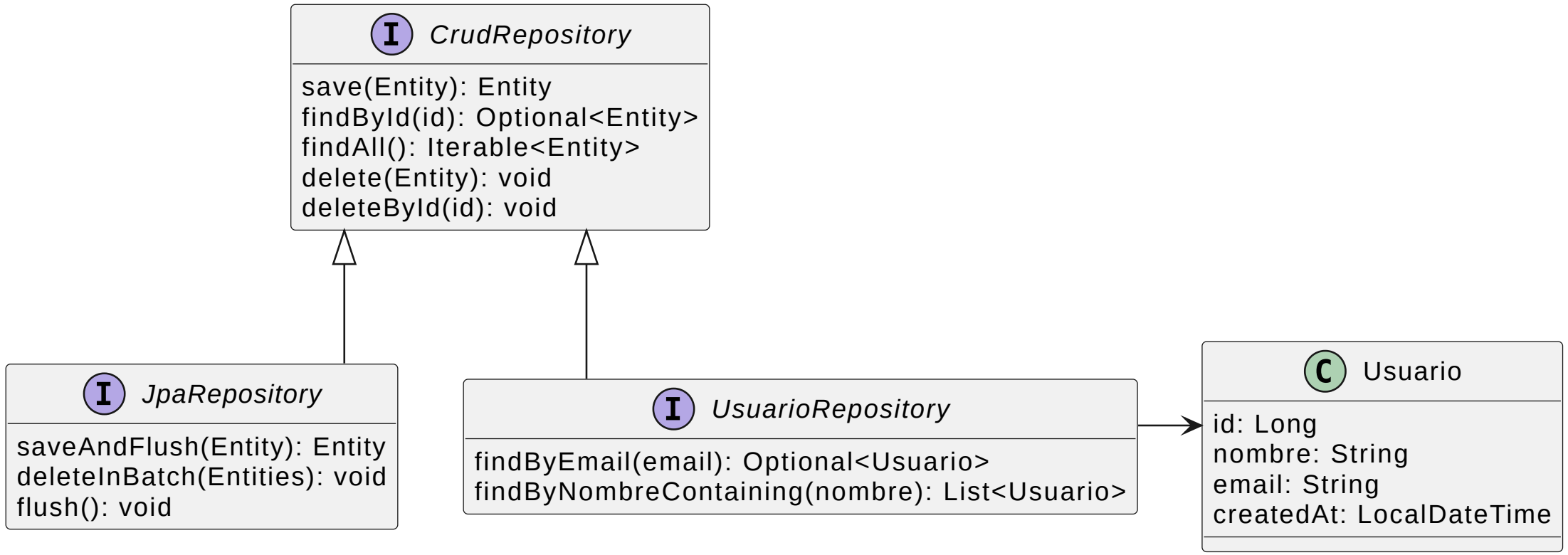
DAO (clásico)

- Interfaz + Implementación explícita
- Métodos CRUD manuales
- Control total sobre SQL
- Verboso, bajo nivel

Repository (moderno)

- Hereda de interfaz genérica
- Métodos CRUD automáticos
- Queries derivadas del método
- Conciso, alto nivel

Repository: Estructura (Spring Data JPA)



Ejemplo: Usuario Repository (Spring Data JPA)

```java
// Interfaz Repository - hereda CRUD automático
@Repository
public interface UsuarioRepository extends JpaRepository<Usuario, Long> {
  // Métodos derivados (queries generadas automáticamente)
  Optional<Usuario> findByEmail(String email);
  List<Usuario> findByNombreContainingIgnoreCase(String nombre);
  List<Usuario> findByCreatedAtAfter(LocalDateTime fecha);
  boolean existsByEmail(String email);
}

// Entity con JPA
@Entity
@Table(name = "usuarios")
@Data
@NoArgsConstructor
public class Usuario {
  @Id
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  private Long id;

  @Column(nullable = false)
  private String nombre;

  @Column(unique = true, nullable = false)
  private String email;

  @CreationTimestamp
  private LocalDateTime createdAt;
}
```

```java
// Usando el Repository en un Servicio
@Service
public class UsuarioService {
  private final UsuarioRepository usuarioRepository;

  @Autowired
  public UsuarioService(UsuarioRepository usuarioRepository) {
    this.usuarioRepository = usuarioRepository;
  }

  public Usuario crearUsuario(Usuario usuario) {
    if (usuarioRepository.existsByEmail(usuario.getEmail())) {
      throw new EmailYaExisteException();
    }
    return usuarioRepository.save(usuario);
  }

  public Usuario obtenerPorId(Long id) {
    return usuarioRepository.findById(id)
      .orElseThrow(() -> new UsuarioNoEncontradoException());
  }

  public List<Usuario> buscarPorNombre(String nombre) {
    return usuarioRepository.
            findByNombreContainingIgnoreCase(nombre);
  }
}
```

Repository: Ventajas respecto a DAO

- **Menos boilerplate**: Métodos CRUD heredados automáticamente de `CrudRepository`
- **Query derivadas**: Métodos generados a partir del nombre (método query method)
- **Transacciones automáticas**: Spring maneja `@Transactional` por defecto
- **Testabilidad**: Fácil crear mocks con Mockito o usar `@DataJpaTest`
- **Integración Spring**: Inyección de dependencias, AOP, etc.
- **Convenciones**: Desarrollo más rápido siguiendo estándares

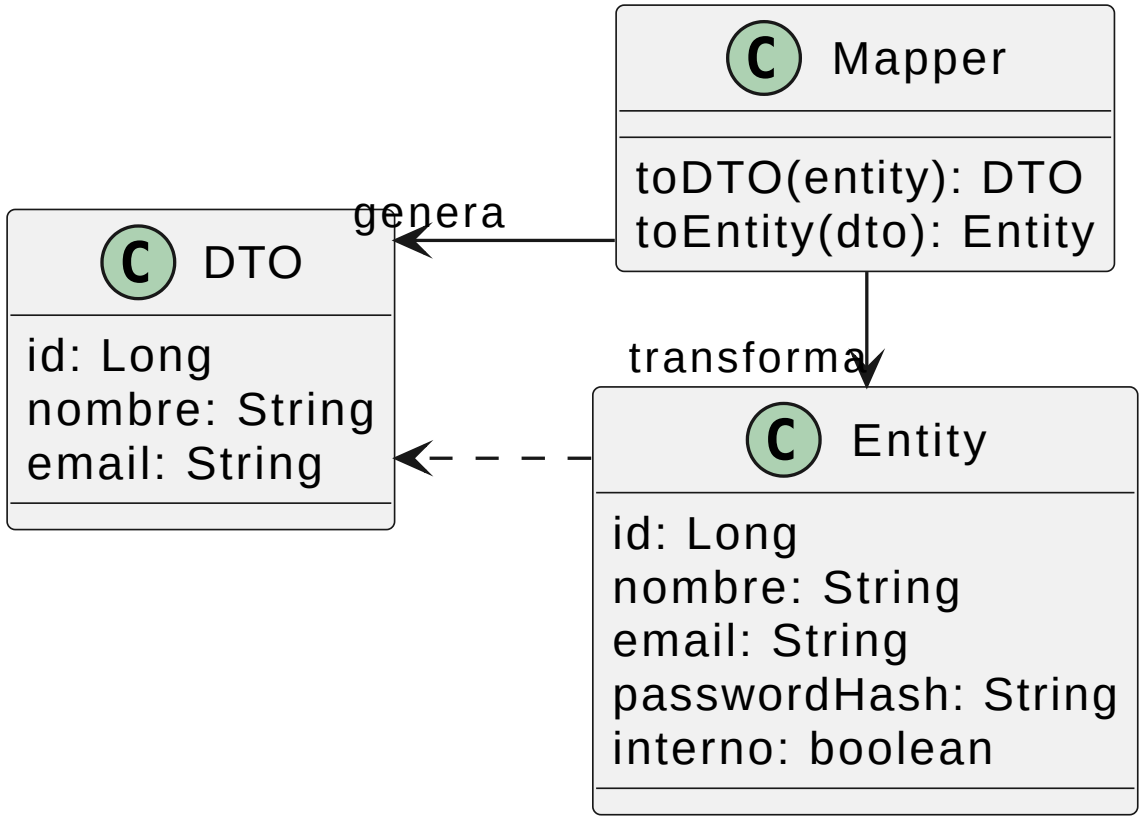
Data Transfer Object (DTO)

Sirve para crear objetos planos o *Plain Old Java Objects* (POJO) que se envían entre aplicaciones, capas, o servidores remotos. Un DTO no tiene comportamiento de negocio, solo almacena y entrega datos (*value object*).

Problema a resolver:

- Exponer entidades de BD directamente en APIs REST crea acoplamiento
- Cambios en la BD obligan a cambios en clientes API
- Puede exponerse información sensible (contraseñas, datos internos)
- Diferentes vistas de datos requieren múltiples selecciones/proyecciones

DTO: Estructura



Ejemplo: Usuario DTO (con ModelMapper)

```java
// Entity JPA (contiene datos + lógica de negocio)
@Entity
@Table(name = "usuarios")
@Data
@NoArgsConstructor
public class Usuario {
  @Id
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  private Long id;

  @Column(nullable = false)
  private String nombre;

  @Column(unique = true, nullable = false)
  private String email;

  @JsonIgnore // No serializar en JSON
  private String passwordHash;

  @CreationTimestamp
  private LocalDateTime createdAt;

  // Métodos de negocio
  public boolean validarPassword(String rawPassword) {
    return BCrypt.checkpw(rawPassword, this.passwordHash);
  }
}
```

```java
// DTO - solo datos públicos (sin contraseña)
@Data
@NoArgsConstructor
@AllArgsConstructor
public class UsuarioDTO {
  private Long id;
  private String nombre;
  private String email;
  private LocalDateTime createdAt;
}

// Mapper usando ModelMapper (spring-boot-starter-modelmapper)
@Component
public class UsuarioMapper {
  private final ModelMapper modelMapper;

  @Autowired
  public UsuarioMapper(ModelMapper modelMapper) {
    this.modelMapper = modelMapper;
  }

  public UsuarioDTO toDTO(Usuario usuario) {
    return modelMapper.map(usuario, UsuarioDTO.class);
  }

  public Usuario toEntity(UsuarioDTO usuarioDTO) {
    return modelMapper.map(usuarioDTO, Usuario.class);
  }

  public List<UsuarioDTO> toDTOList(List<Usuario> usuarios) {
    return usuarios.stream()
      .map(this::toDTO)
      .collect(Collectors.toList());
  }
}
```

```java
// Controlador REST usando DTO
@RestController
@RequestMapping("/api/usuarios")
public class UsuarioController {
  private final UsuarioService usuarioService;
  private final UsuarioMapper usuarioMapper;

  @GetMapping("/{id}")
  public ResponseEntity<UsuarioDTO>
          obtenerUsuario(@PathVariable Long id) {
    Usuario usuario = usuarioService.obtenerPorId(id);
    return ResponseEntity.ok(usuarioMapper.toDTO(usuario));
  }

  @GetMapping
  public ResponseEntity<List<UsuarioDTO>>
          listarUsuarios() {
    List<Usuario> usuarios = usuarioService.listarTodos();
    return ResponseEntity.ok(usuarioMapper.toDTOList(usuarios));
  }

  @PostMapping
  public ResponseEntity<UsuarioDTO>
          crearUsuario(@RequestBody UsuarioDTO usuarioDTO) {
    Usuario usuario = usuarioMapper.toEntity(usuarioDTO);
    Usuario creado = usuarioService.crearUsuario(usuario);
    return ResponseEntity.created(URI.create("/api/usuarios/" +
                                  creado.getId()))
      .body(usuarioMapper.toDTO(creado));
  }
}
```

DTO: Ventajas y desventajas

Ventajas:

- **Separación de responsabilidades**: Entidad y DTO pueden evolucionar independientemente
- **Seguridad**: control sobre qué datos se exponen (ej: no exponer contraseñas)
- **Versionado API**: diferentes versiones de DTO para diferentes clientes
- **Performance**: seleccionar solo datos necesarios (proyecciones)
- **Contrato de API**: DTOs actúan como contrato entre cliente-servidor

Desventajas:

- **Duplicación**: mantener Entity y DTO con mapeos entre ambos
- **Boilerplate**: código repetitivo si hay muchos DTOs
- **Overhead**: conversión Entity $\leftrightarrow$ DTO tiene coste de CPU/memoria

Repository + DTO + Servicio integrados

```java
@Service
public class UsuarioService {
    private final UsuarioRepository usuarioRepository;
    private final UsuarioMapper usuarioMapper;

    @Autowired
    public UsuarioService(UsuarioRepository usuarioRepository,
                          UsuarioMapper usuarioMapper) {
        this.usuarioRepository = usuarioRepository;
        this.usuarioMapper = usuarioMapper;
    }

    public UsuarioDTO obtenerPorId(Long id) {
        Usuario usuario = usuarioRepository.findById(id)
            .orElseThrow(() -> new UsuarioNoEncontradoException());
        return usuarioMapper.toDTO(usuario);  // Entity → DTO
    }

    public List<UsuarioDTO> buscarPorNombre(String nombre) {
        List<Usuario> usuarios = usuarioRepository
            .findByNombreContainingIgnoreCase(nombre);
        return usuarioMapper.toDTOList(usuarios);  // List<Entity> → List<DTO>
    }

    public UsuarioDTO crearUsuario(UsuarioDTO usuarioDTO) {
        Usuario usuario = usuarioMapper.toEntity(usuarioDTO);  // DTO → Entity
        Usuario creado = usuarioRepository.save(usuario);
        return usuarioMapper.toDTO(creado);  // Entity → DTO
    }
}
```

Para profundizar sobre patrones

- Martin Fowler – Patterns in Enterprise Software: Catálogos de patrones a distintos niveles
  - Martin Fowler – Patterns of Enterprise Application Architecture (EAA)
  - Hohpe y Woolf – Enterprise Integration Patterns (EIP)
  - Buschmann y otros – Pattern-Oriented Software Architecture (POSA) Volume 1: A system of patterns
- Peter Norvig – Design Patterns in Dynamic Programming: Implementaciones más simples para los patrones de diseño del GoF en lenguajes dinámicos

Para profundizar sobre patrones

- David Arno – Are design patterns compatible with modern software techniques?
- Implementaciones de los patrones de diseño del GoF en diversos lenguajes de programación:
  - Kamran Ahmed – Design Patterns for Humans!: Explicación de los patrones de diseño del GoF implementados en PHP
  - Márk Török – Design Patterns in TypeScript
  - Bogdab Vliv - Design Patterns in Ruby
- Lewis y Fowler – Microservicios
- Chris Richardson - Microservices patterns
- Spring Data JPA – Query Methods Documentation
- ModelMapper – Documentation and Examples