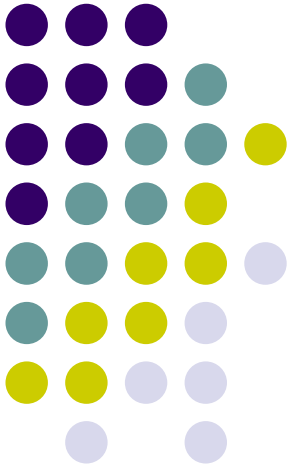
Patrones de Diseño

Patrón de comportamiento *Strategy*



Strategy
Propósito

- Definir una familia de algoritmos, encapsularlos y hacerlos intercambiables
- Permite que el algoritmo cambie sin que afecte a los clientes que lo usan
- También conocido como *policy* (política)

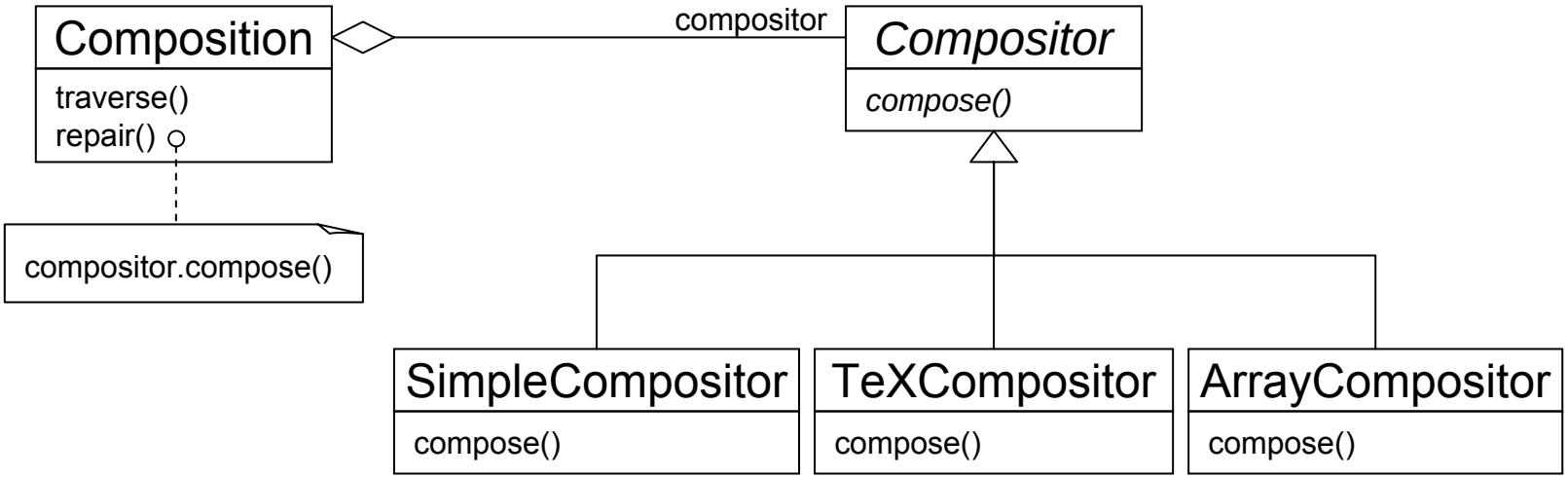
Strategy
Motivación

- Ej: existen muchos algoritmos para dividir un flujo de texto en líneas

- Codificar los algoritmos en las clases que los necesitan no es deseable por lo siguiente:
  - Los clientes se hacen más complejos
  - Distintos algoritmos serán apropiados en distintos momentos
  - Es difícil añadir nuevos algoritmos y modificar los existentes
  - No hay reutilización

- Solución:
  - definir clases que encapsulen los distintos algoritmos

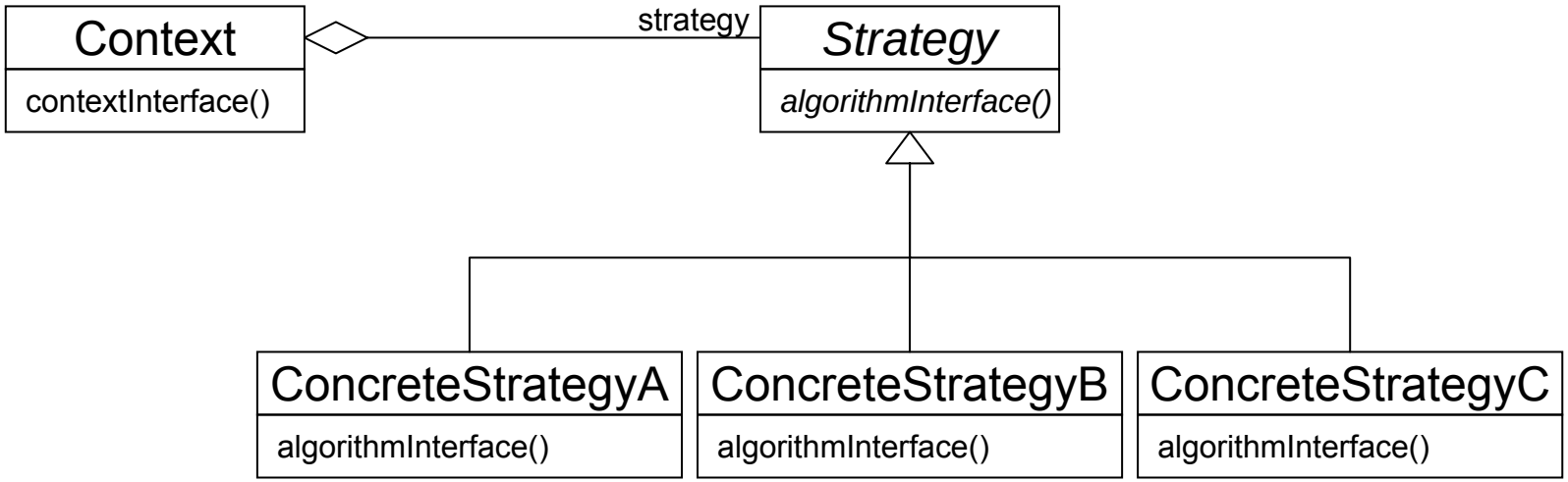
Strategy
Motivación



Strategy
Aplicabilidad

- Usa el patrón *Strategy* cuando:
  - Varias clases relacionadas sólo difieren en su comportamiento. *Strategy* permite configurar a una clase con uno de entre varios comportamientos
  - Se necesiten variantes del mismo algoritmo, que se implementan como una jerarquía de clases
  - Un algoritmo usa datos que los clientes no tienen por qué conocer (ej. estructuras de datos específicas del algoritmo)
  - Una clase define muchos comportamientos que aparecen en sentencias condicionales → mover los relacionados a un *strategy*

Strategy
Estructura



Strategy
Participantes

- **Strategy** (*Compositor*): define una interfaz común a los algoritmos que soporta.
- **ConcreteStrategy** (*SimpleCompositor*, *TeXCompositor*, *ArrayCompositor*): implementa un algoritmo usando la interfaz *Strategy*
- **Context** (*Composition*)*:*
  - Está configurado con un objeto *ConcreteStrategy*
  - Mantiene una referencia al objeto *Strategy*
  - Puede definir una interfaz que le permita a *Strategy* acceder a sus datos

Strategy
Consecuencias

- Ayuda a factorizar funcionalidad común de los algoritmos
- Alternativa a herencia estática (subclasificar *Context*) que facilita la comprensión, mantenimiento y extensión. Además permite cambiar el algoritmo dinámicamente.
- Ayuda a eliminar sentencias condicionales

```
switch (_breakingStrategy) {
  case SimpleStrategy:
     composeWithSimpleCompositor();
     break;
  case TeXStrategy:
     composeWithTeXCompositor();
     break;
  // ...
}
```

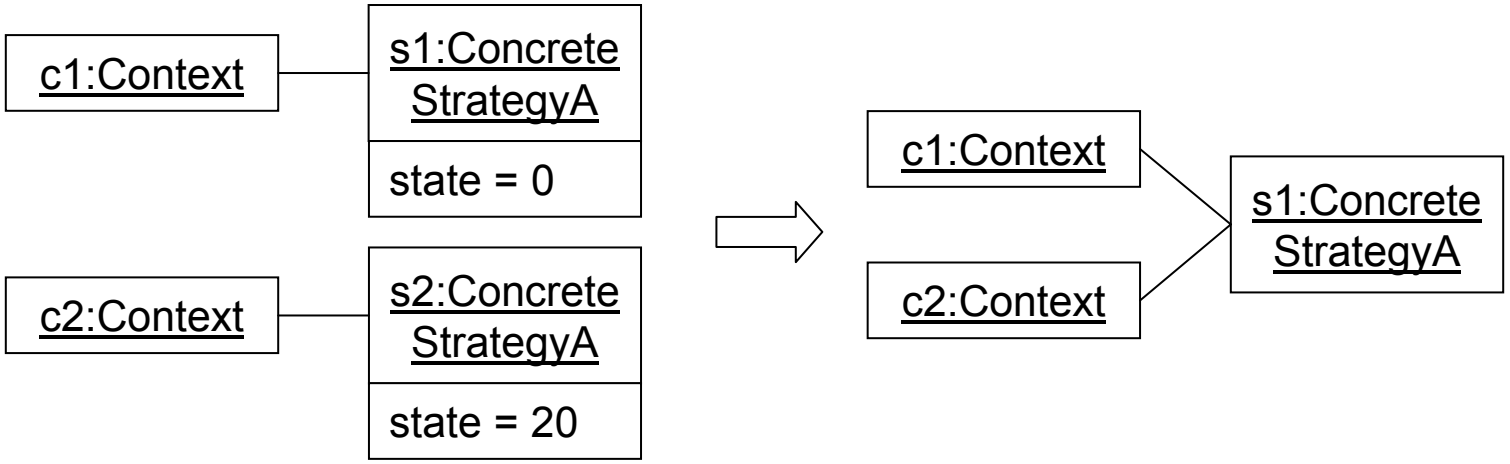
```
_compositor.compose();
```

Strategy
Consecuencias

- Diferentes implementaciones del mismo comportamiento
- El cliente debe entender en qué difieren las implementaciones de una estrategia → usar el patrón sólo si esas diferencias son relevantes
- Estimación “por lo alto” de la comunicación entre el *Context* y el *Strategy* para tratar implementaciones más y menos complejas
  - implementación 1 usa como datos de entrada `(int dato1, String dato2)`
  - implementación 2 usa como datos de entrada `(int dato1)`
  - ¿qué pongo en la interfaz *Strategy*?

Strategy
Consecuencias

- Incremento del número de objetos. Puede reducirse implementando estrategias sin estado compartidas




```
// ejecutar doStep1 modifica el
// atributo state de la estrategia

_strategy.doStep1(...);
_strategy.doStep2(...);
```

```
// el estado se pasa en las llamadas
// sucesivas a la estrategia

int state = _strategy.doStep1(...);
_strategy.doStep2(..., state);
```

Strategy
Código de ejemplo


```
public interface Strategy {
   public void execute();
}

public class ConcreteStrategyA implements Strategy {
   public void execute() { ... }
}

public class ConcreteStrategyB implements Strategy {
   public void execute() { ... }
}

public class Context {
   private Strategy _strategy;
   public Context (Strategy s) { _strategy = s; }
   public Context () { _strategy = new ConcreteStrategyA(); }
   public void execute() { _strategy.execute(); }
}

public class Client {
   public static void main (String args[]) {
      Context context = new Context(new ConcreteStrategyA());
      context.execute();
   }
}
```

Strategy
Implementación

- Definición de la comunicación entre *Context* y *Strategy*
  - El contexto pasa sus datos como argumentos de las operaciones de la estrategia: bajo acoplamiento, posible paso de parámetros innecesarios
  - El contexto se pasa a sí mismo como argumento: alto acoplamiento
- Configurar el contexto con una estrategia (tipos genéricos)
  - Si la estrategia se puede usar en tiempo de compilación
  - Si la estrategia no va a cambiar en tiempo de ejecución
- Comportamiento por defecto en el contexto sin estrategia
  - Creo y utilizo un objeto estrategia sólo si es necesario

Strategy
Ejercicio

- **Se quiere construir una herramienta CASE para el modelado con UML**
- **Su interfaz debe permitir redistribuir los elementos de un diagrama UML según diversos *layouts*: *CircleLayout*, *FlowLayout*, *BorderLayout*...**
- **Realizar el diseño de la aplicación**

Strategy
Solución