SEMINARIO

6

Repaso

Diseño del Software

Grado en Ingeniería Informática del Software

Curso 2017-2018

Escuela de Ingeniería Informática
Universidad de Oviedo

César Acebal

Patrones de creación

Cuestiones

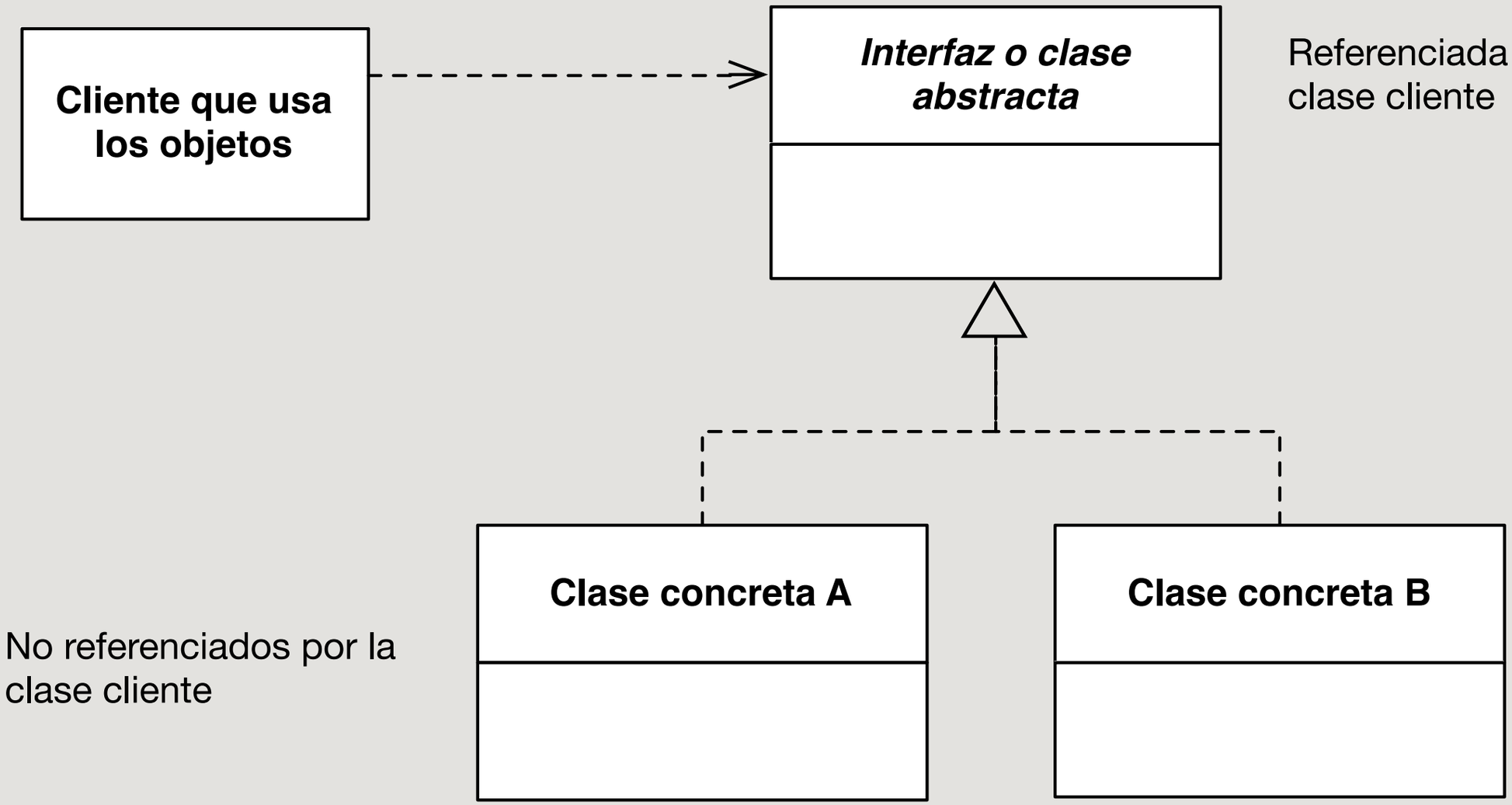
- **Define «factoría»**
  - Una factoría es cualquier método u objeto que se usa para crear otros objetos
- **Nombra los patrones de diseño que hemos visto que son factorías**
  - *Singleton*, *Factory Method*, *Abstract Factory* y *Prototype*

Cuestiones

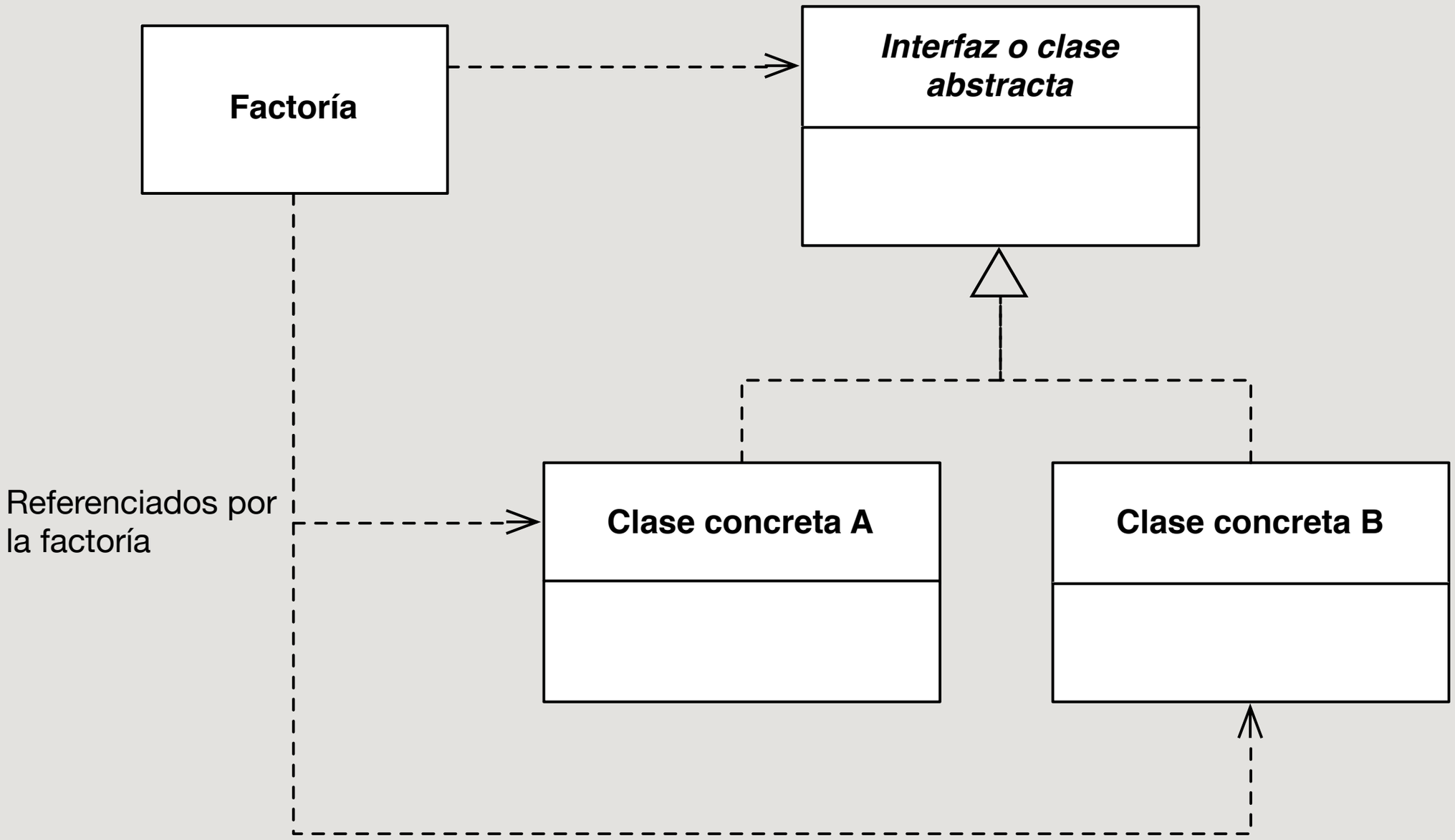
- **¿Cuál sería una regla general a la hora de gestionar la creación de objetos?**
  - Una clase debería encargarse de crear y gestionar otros objetos o simplemente utilizarlos, pero, por lo general, no ambas cosas
  - Menor acoplamiento: hay una clara división de tareas
    - (Principio de responsabilidad única)

Los objetos que usan otros objetos, están desacoplados de ellos: no saben con qué objetos están tratando (o cuáles podrían aparecer en un futuro). Ese trabajo corresponde a las factorías (que, a su vez, sólo saben qué objetos están creando o gestionando, sin importarle cómo se usan).

El cliente que los usa



La fábrica que los crea



Patrones de creación

- **Hemos visto los siguientes:**
  - Singleton
  - Factory Method
  - Abstract Factory
  - Prototype

Patrones de crea

- Singleton
- Factory Method
- Abstract Factory
- Prototype

- **En términos generales, ¿qué persiguen todos ellos?**

Abstraer a los clientes de las clases concretas de objetos a crear

El «new» nos obliga a especificar la clase del objeto a crear

En ocasiones... ¡ésta ni siquiera será conocida en tiempo de compilación!

Vendrían a cumplir así el principio de «programar para una interfaz, no para una implementación» (bueno, quizás con la excepción del Singleton)

Singleton

- ¿Para qué se usaba?
- ¿Qué problemas presentaba?
- Que una clase sea un «Singleton»... ¿implica necesariamente que sus clientes tengan que saberlo?

¿Qué permitía?

¿A qué decíamos que se parecía?, ¿por qué era «peligroso»?

¿Siempre usaremos el método «getInstance»?

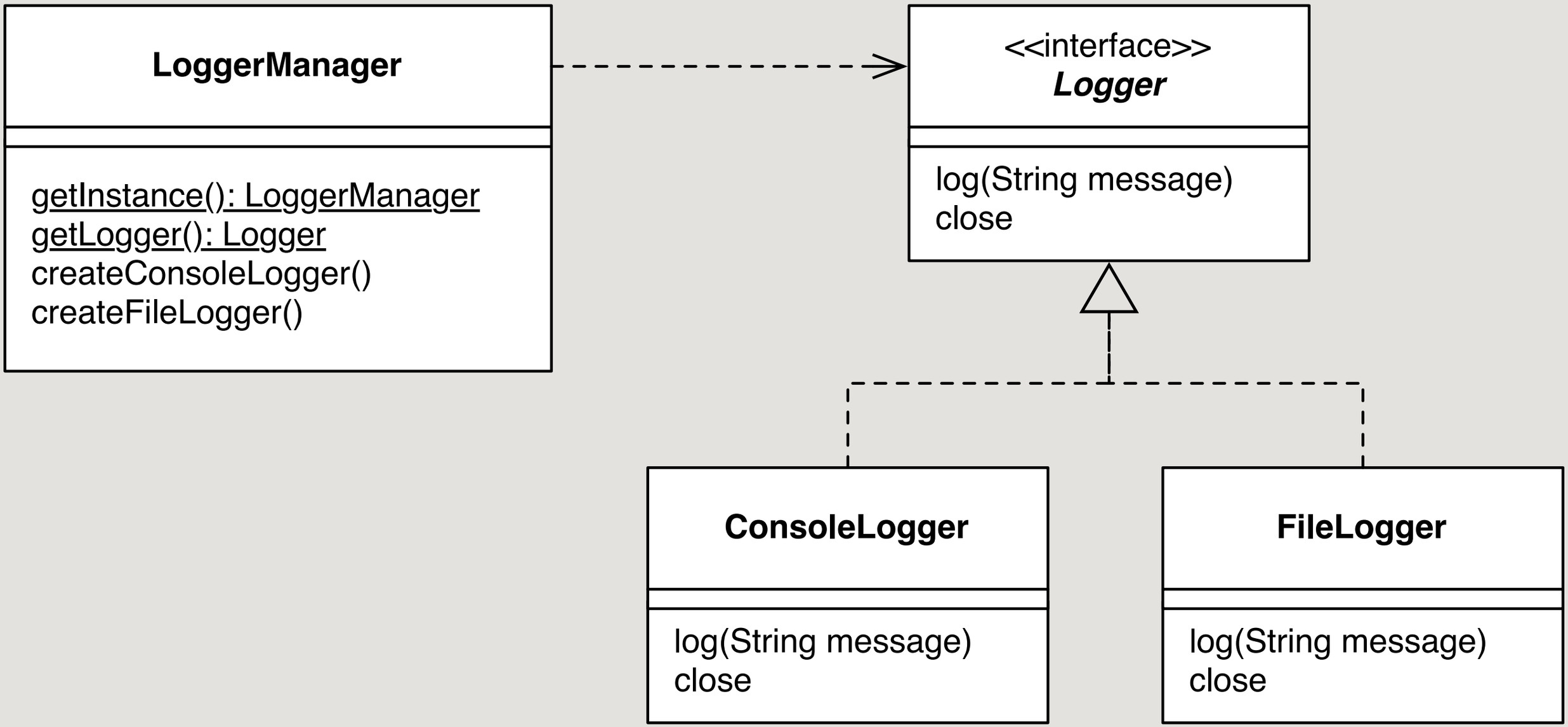
Factorías

- **Diferencias entre Factory Method y Abstract Factory**

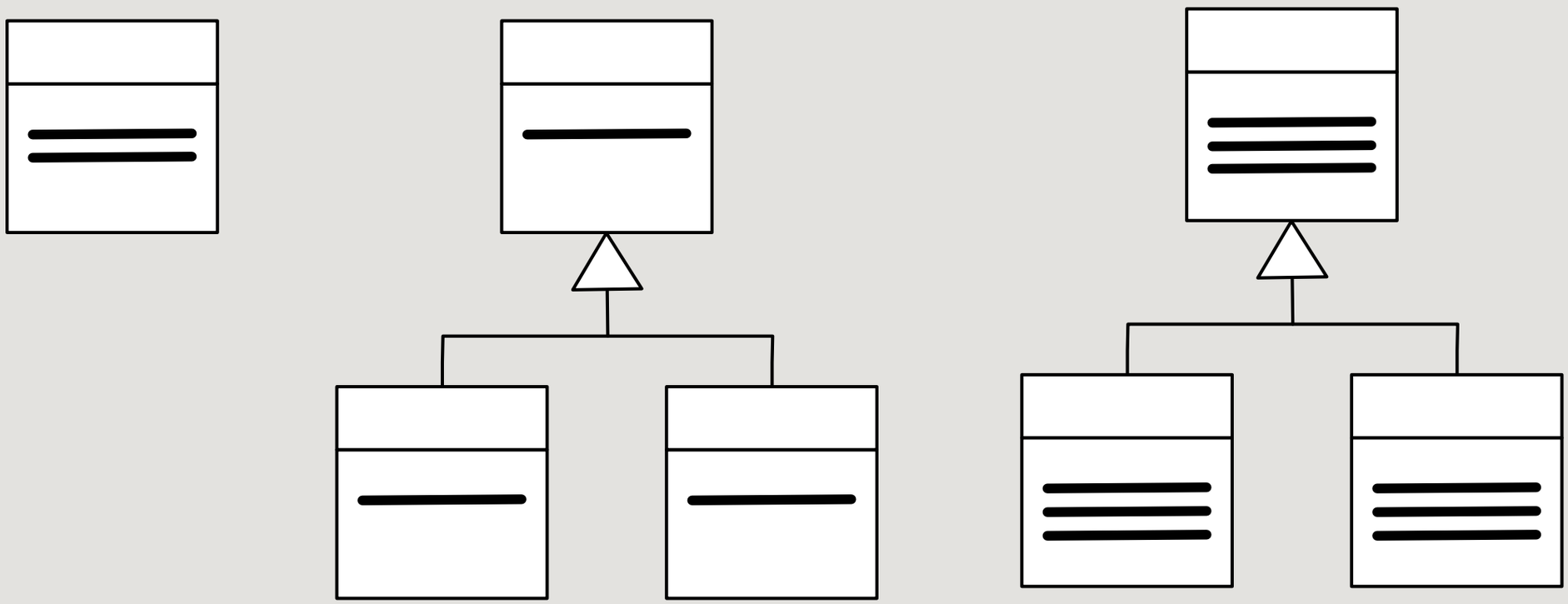
  ¿Realmente se parecen?

- **¿A qué llamábamos «factorías», a secas?**

Ejemplo

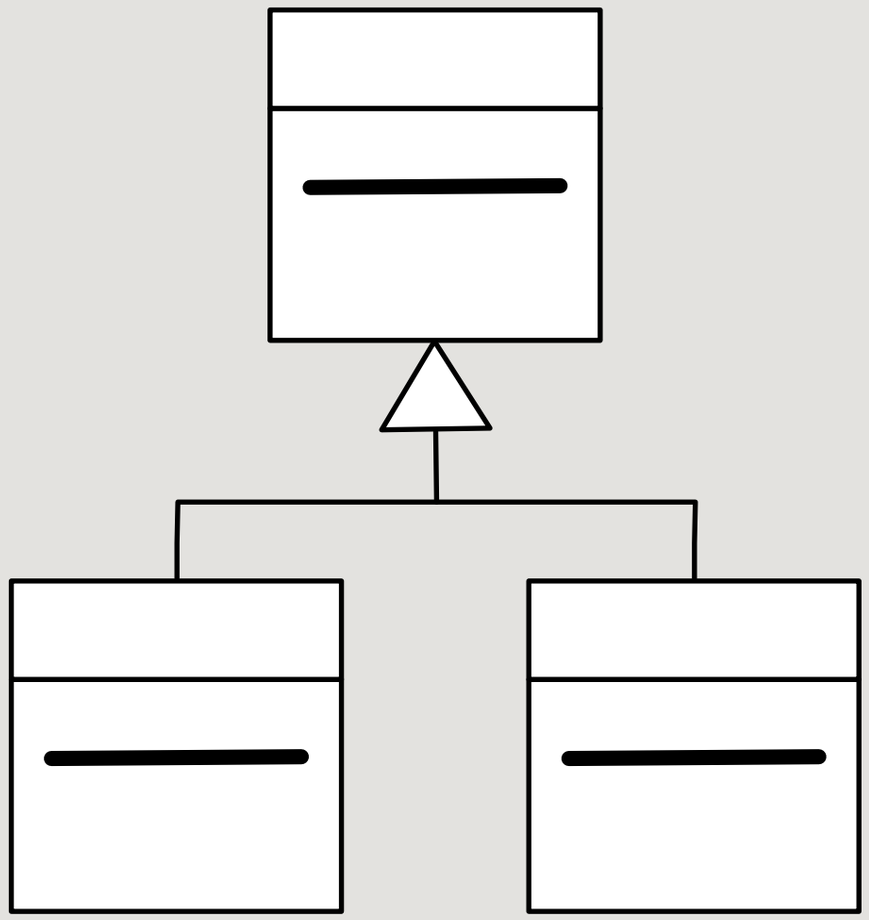

LoggerManager
getInstance(): LoggerManager
getLogger(): Logger
createConsoleLogger()
createFileLogger()
<<interface>>
Logger
log(String message)
close
ConsoleLogger
log(String message)
close
FileLogger
log(String message)
close

Diferencias



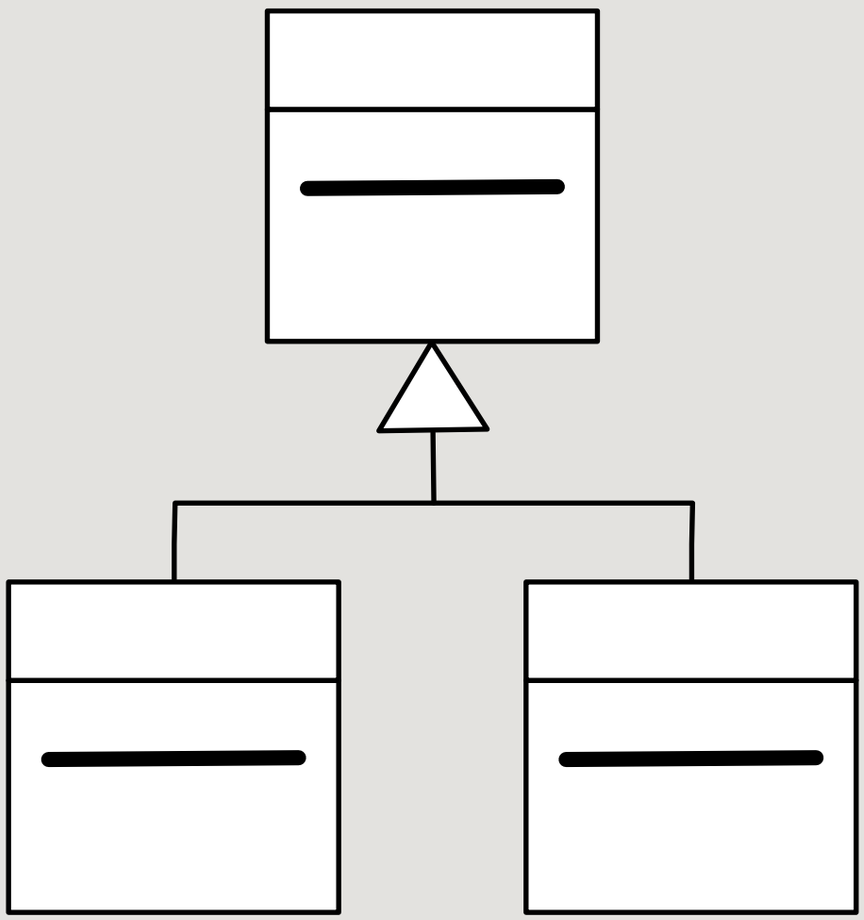
Las líneas en negrita representan métodos que crean objetos. Esta figura, debida a Kerievsky («Refactoring to Patterns», 2005), ilustra de ese modo, muy esquemáticamente, las diferencias más significativas entre una simple clase de creación y los patrones de diseño *Factory Method* y *Abstract Factory*.

¿Y el Prototype?

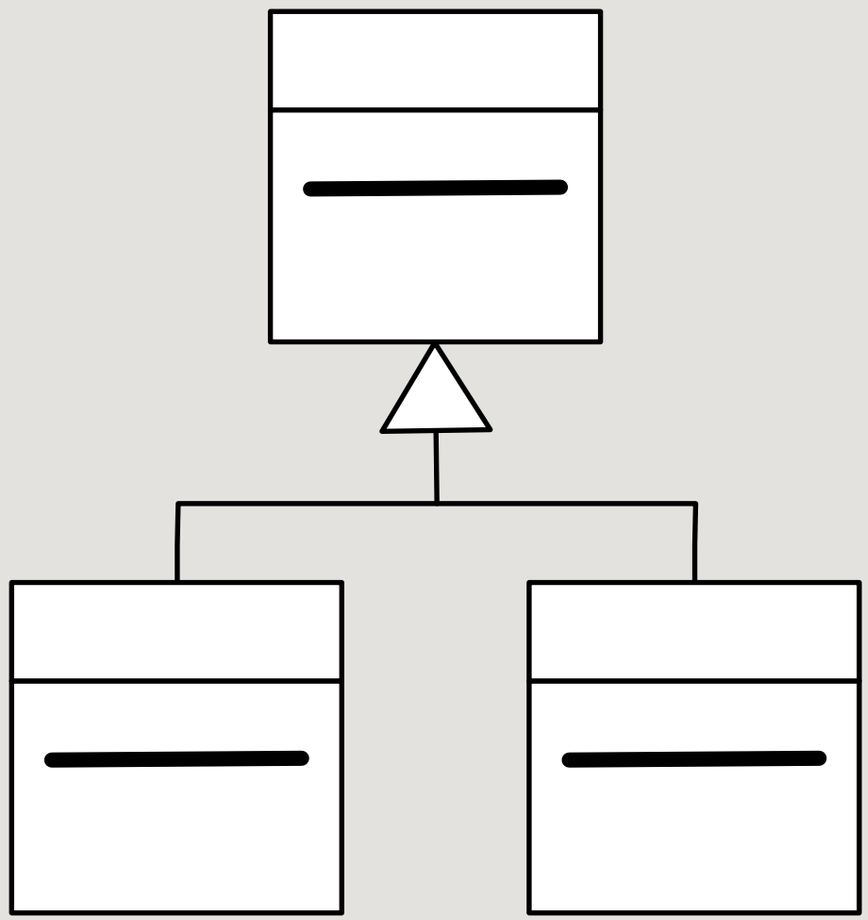


Prototype

¿Y el Prototype?



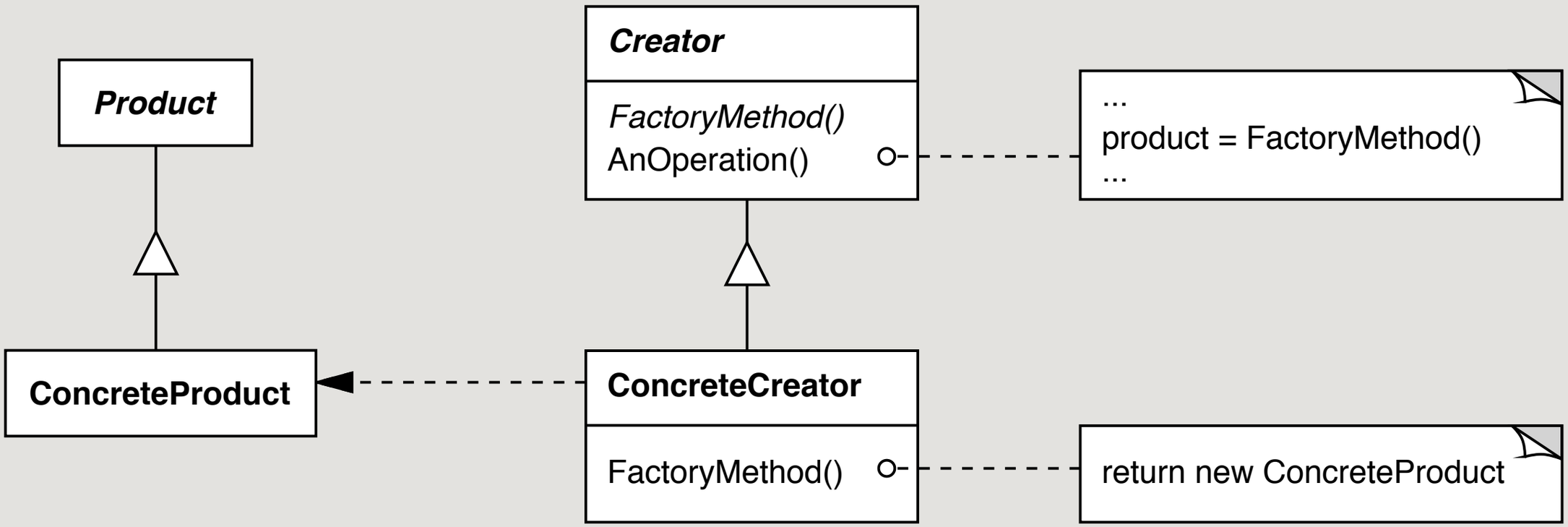
Prototype



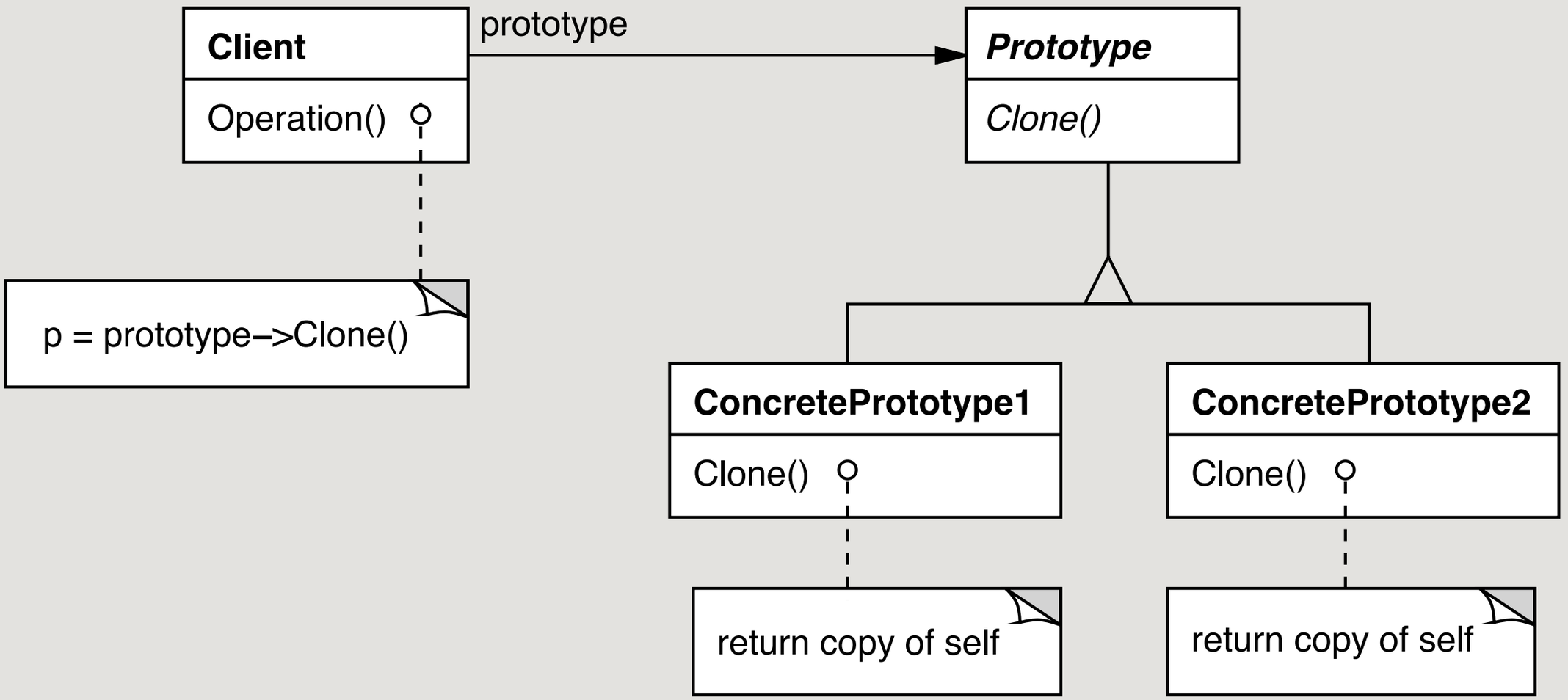
Factory Method

¿En qué se diferencia del Factory Method?

Factory Method



Prototype



El prototipo crea una copia de sí mismo. El Factory Method crea objetos de otras clases.

Otros patrones estudiados

Veamos algunos ejemplos de dónde los hemos usado en prácticas (o en seminarios)

Repaso

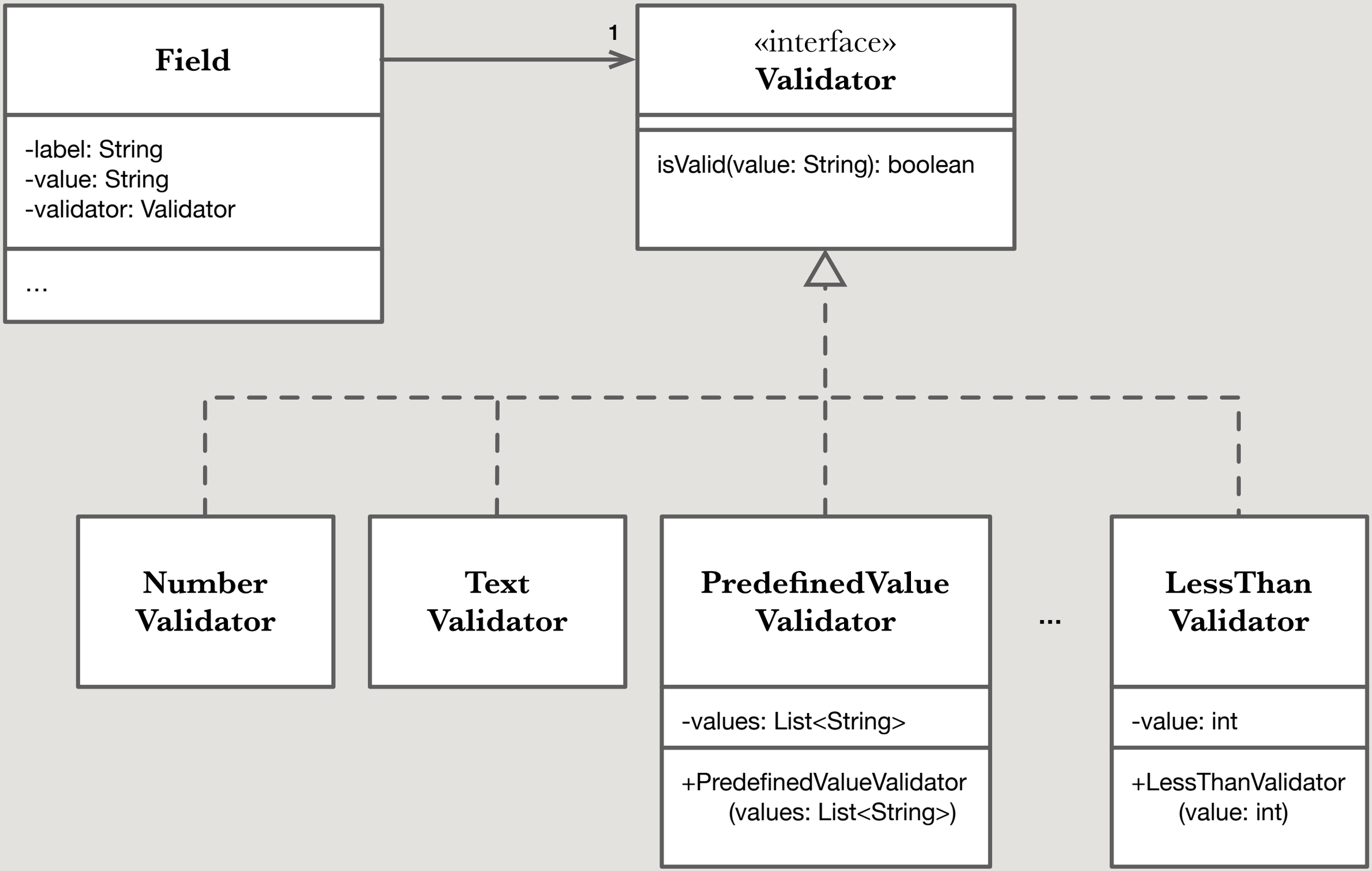
- **Otros patrones estudiados:**
  - Strategy
  - Composite
  - Command
  - Visitor
  - State
  - Template Method
  - Observer
  - Adapter
  - Decorator

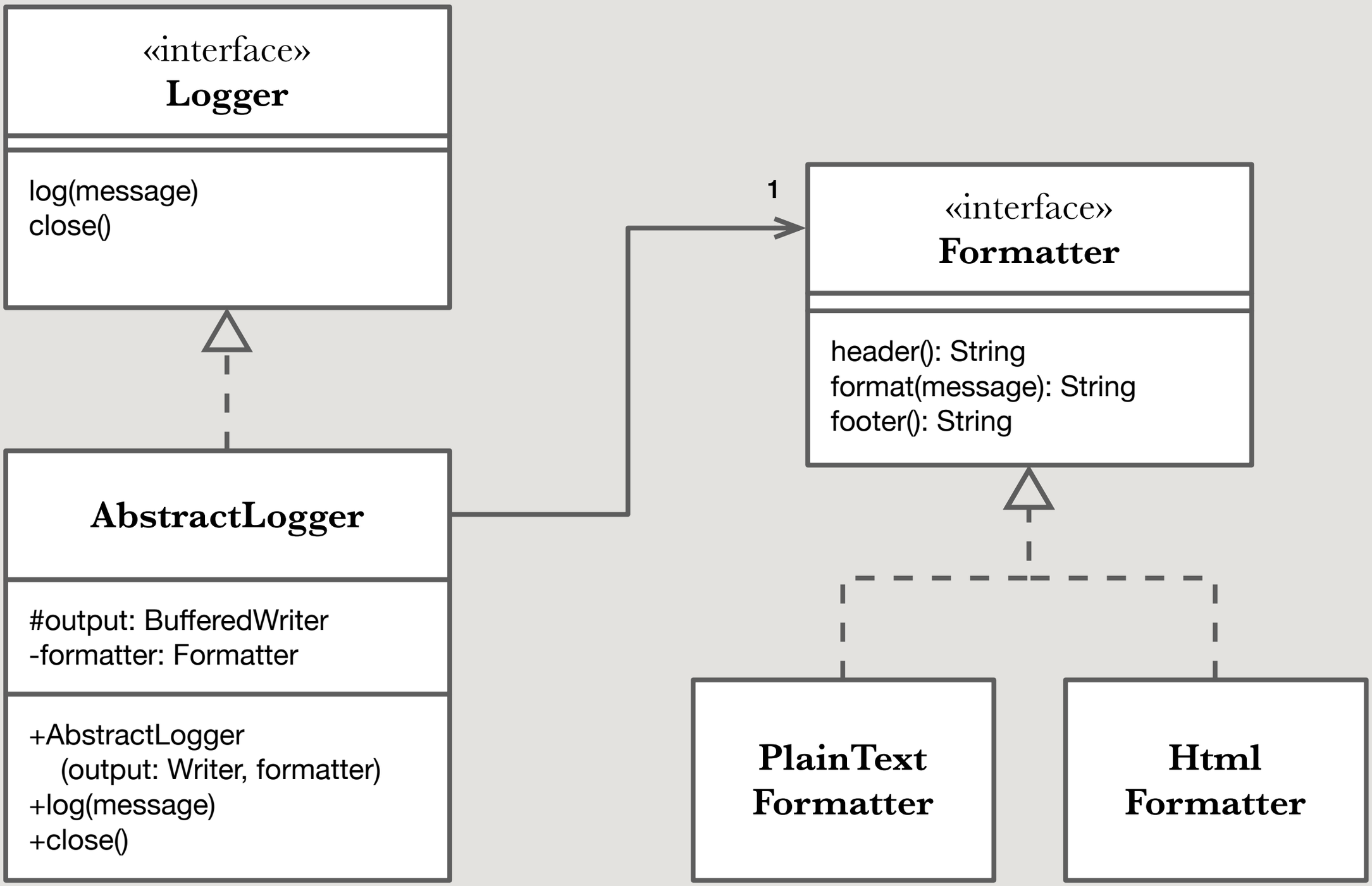
- **Patrones:**
  - Strategy
  - Composite
  - Command
  - Visitor
  - State
  - Template Method
  - Observer
  - Adapter
  - Decorator

- **Prácticas**
  - Videoclub
  - Intérprete de pila
  - Editor gráfico
  - Formulario
  - Editor con deshacer/ repetir
  - Intérprete (AST)
  - Mapas
  - Sistema de ficheros
  - Ball Game
  - Encuestas
  - Logger (seminarios)

Strategy

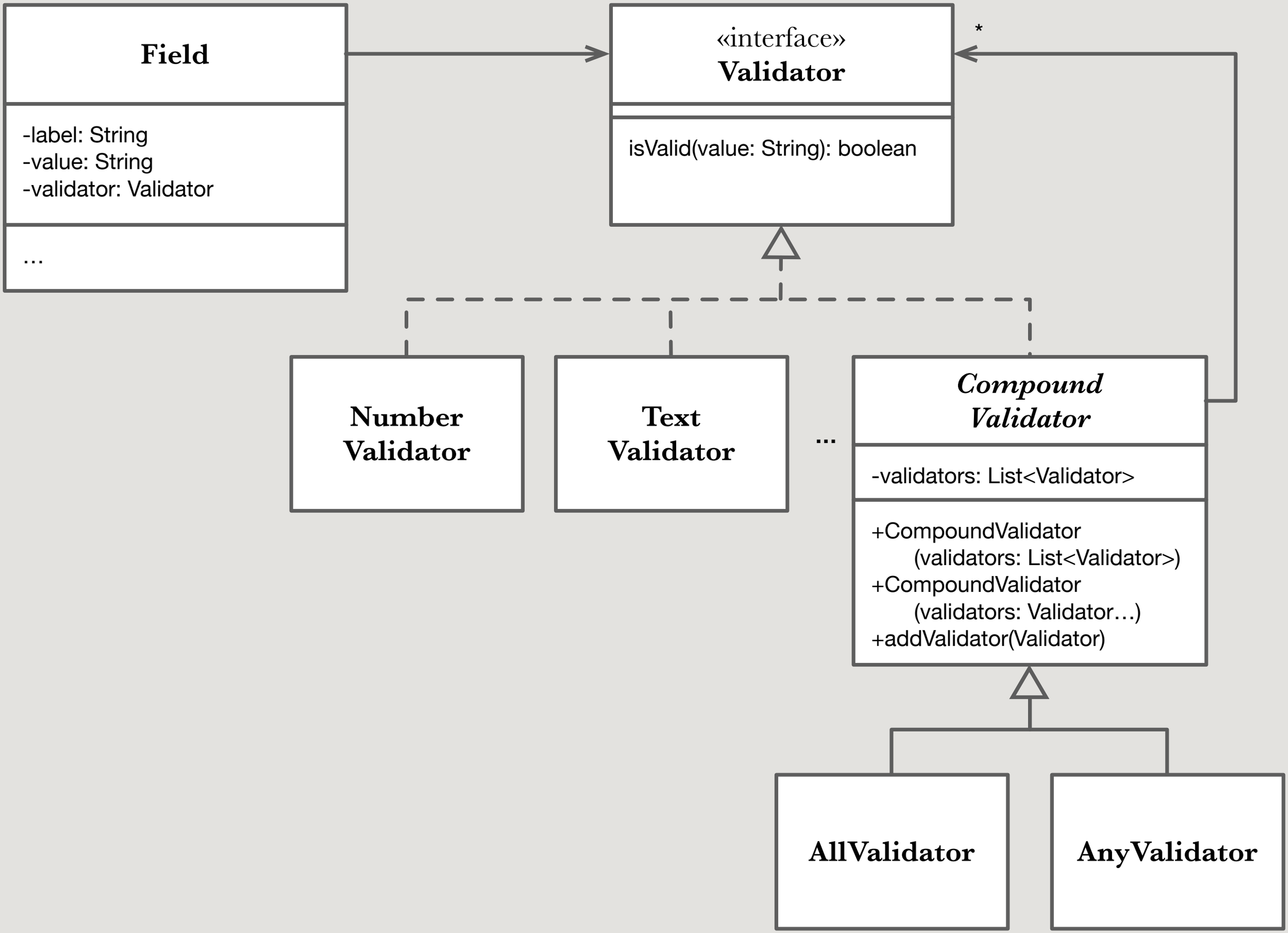
Algunos ejemplos...

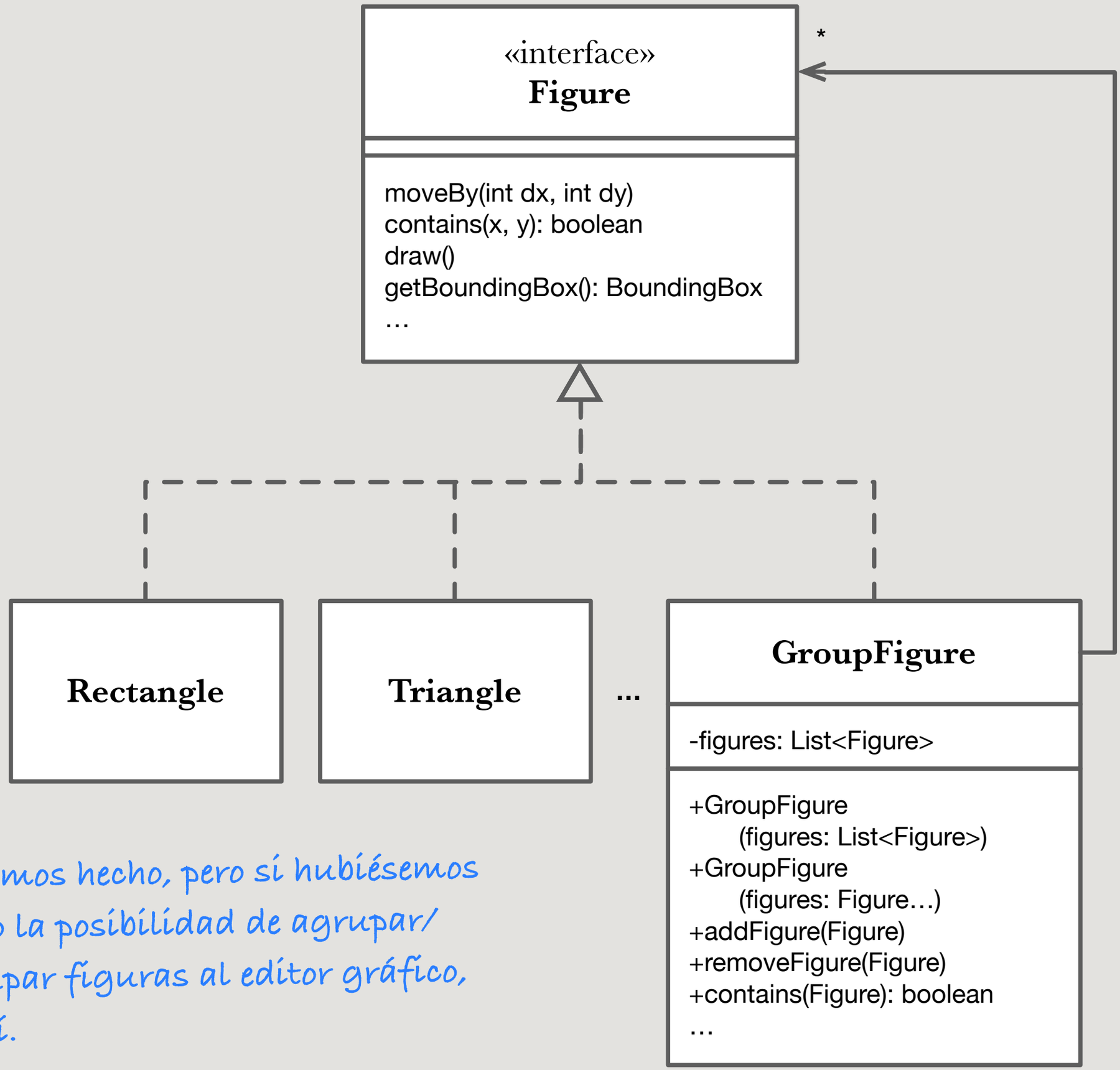

Field
-label: String
-value: String
-validator: Validator
...
1
«interface»
Validator
isValid(value: String): boolean
Number Validator
Text Validator
PredefinedValue Validator
-values: List<String>
+PredefinedValueValidator (values: List<String>)
...
LessThan Validator
-value: int
+LessThanValidator (value: int)


«interface»
Logger
log(message)
close()
1
«interface»
Formatter
header(): String
format(message): String
footer(): String
AbstractLogger
#output: BufferedWriter
-formatter: Formatter
+AbstractLogger
(output: Writer, formatter)
+log(message)
+close()
PlainText
Formatter
Html
Formatter

Composite

Algunos ejemplos...

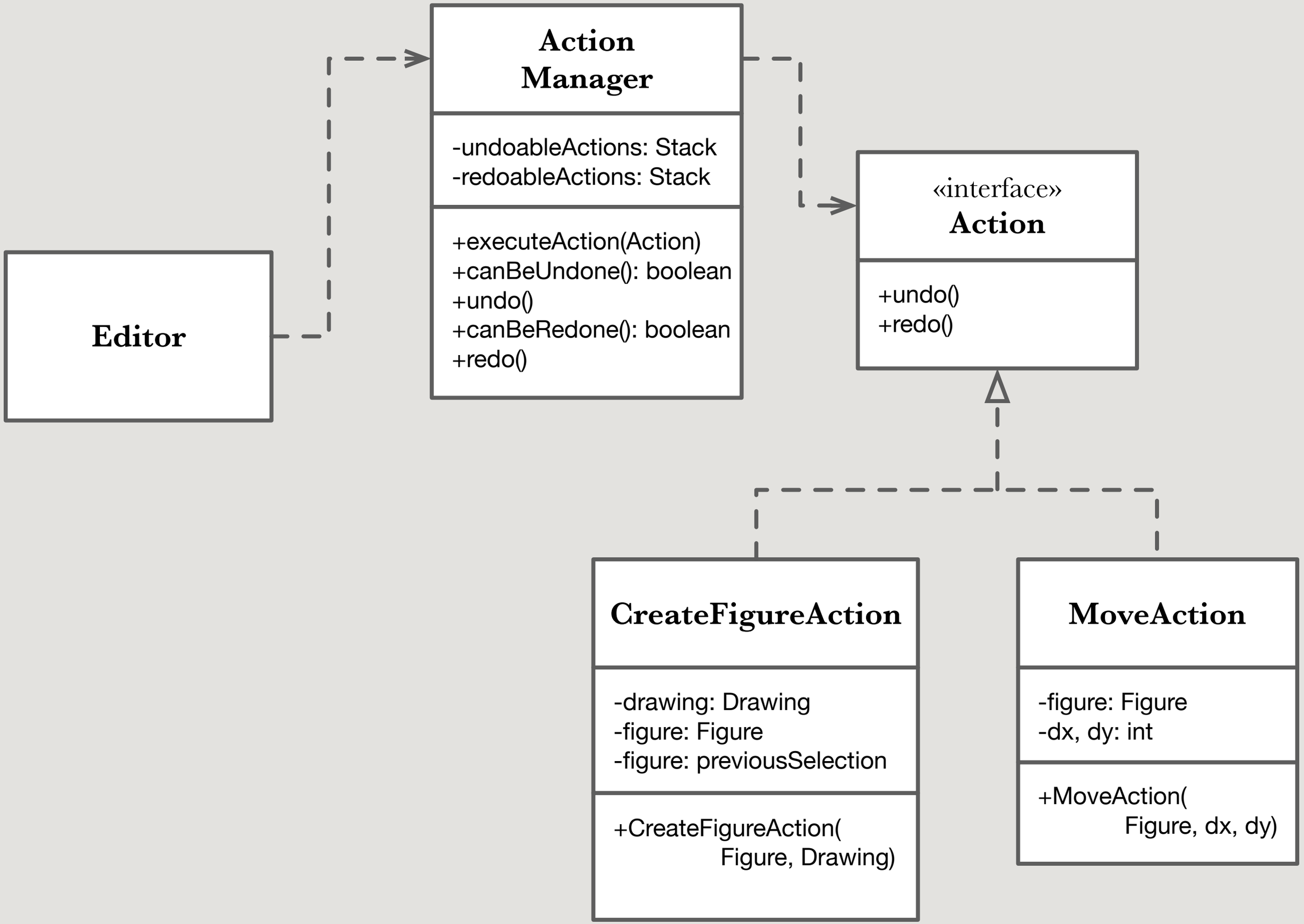

Field
-label: String
-value: String
-validator: Validator
...
«interface»
Validator
isValid(value: String): boolean
*
Number Validator
Text Validator
...
Compound Validator
-validators: List<Validator>
+CompoundValidator (validators: List<Validator>)
+CompoundValidator (validators: Validator…)
+addValidator(Validator)
AllValidator
AnyValidator

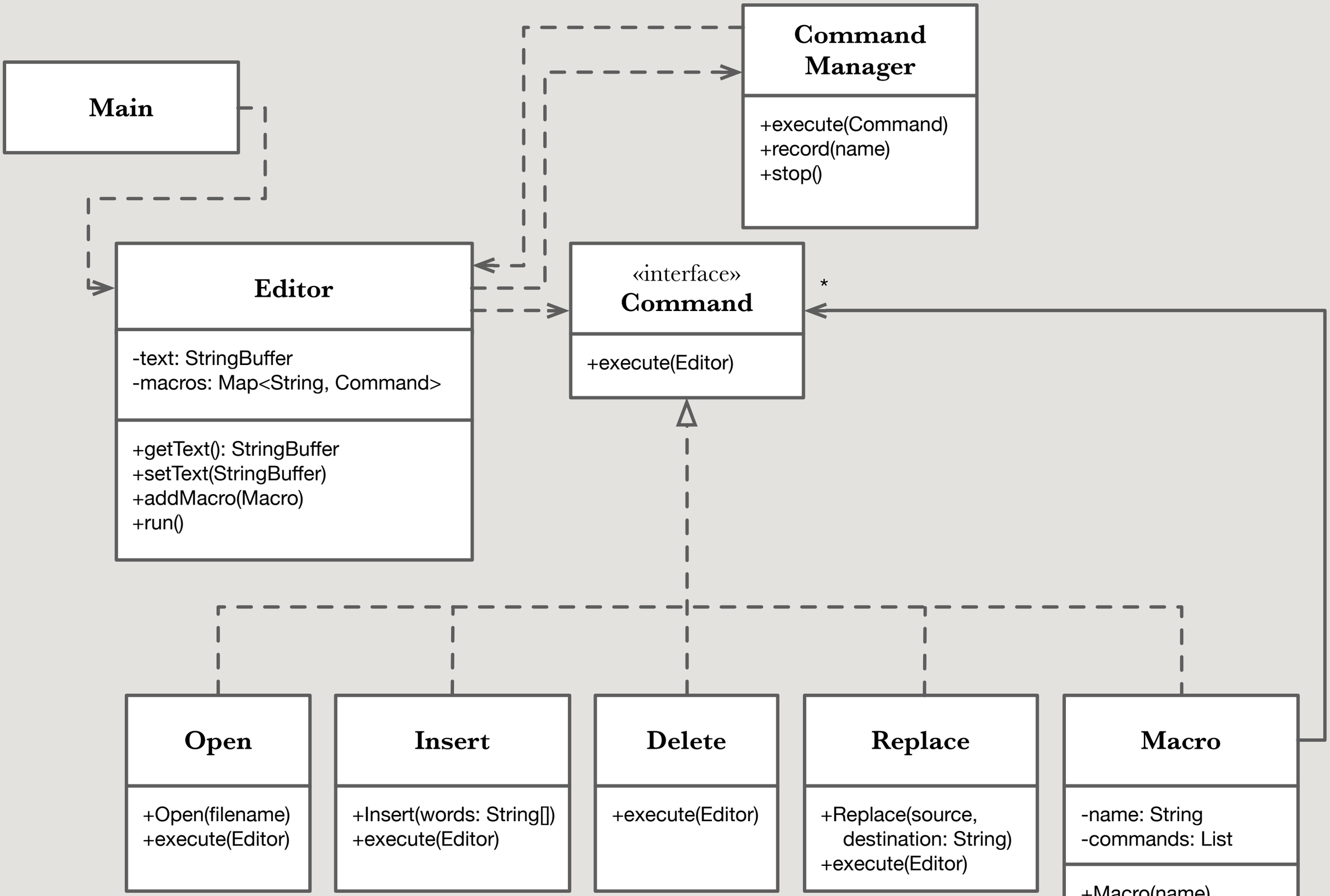


No la hemos hecho, pero si hubiésemos añadido la posibilidad de agrupar/desagrupar figuras al editor gráfico, sería así.

Command

Algunos ejemplos...


Action Manager
-undoableActions: Stack
-redoableActions: Stack
+executeAction(Action)
+canBeUndone(): boolean
+undo()
+canBeRedone(): boolean
+redo()
Editor
«interface»
Action
+undo()
+redo()
CreateFigureAction
-drawing: Drawing
-figure: Figure
-figure: previousSelection
+CreateFigureAction(
Figure, Drawing)
MoveAction
-figure: Figure
-dx, dy: int
+MoveAction(
Figure, dx, dy)

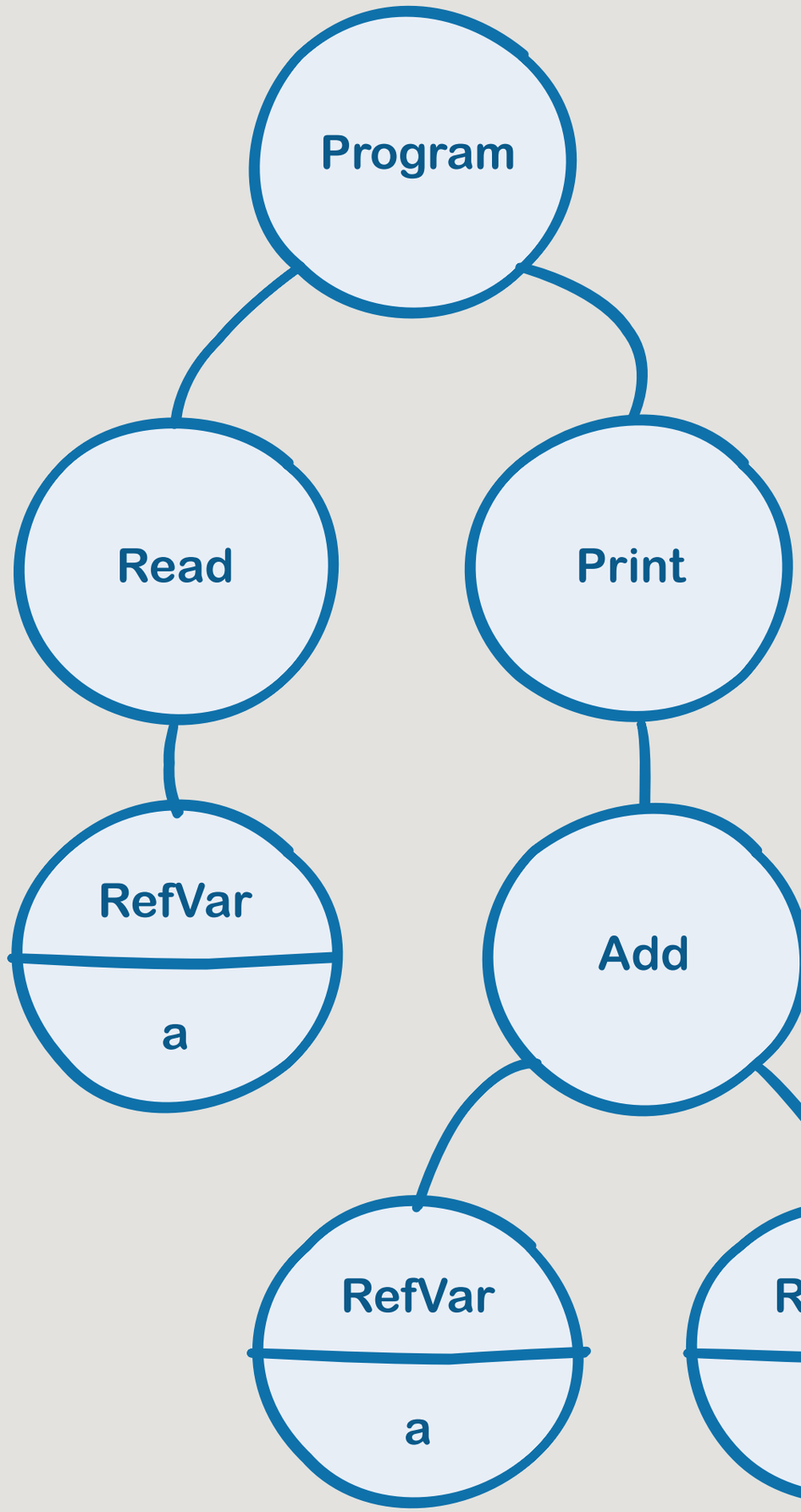


Aquí tenemos además otro ejemplo de Composite (las «macros»)

Visitor

Operaciones sobre un AST

```
read a;
print a + b;
```

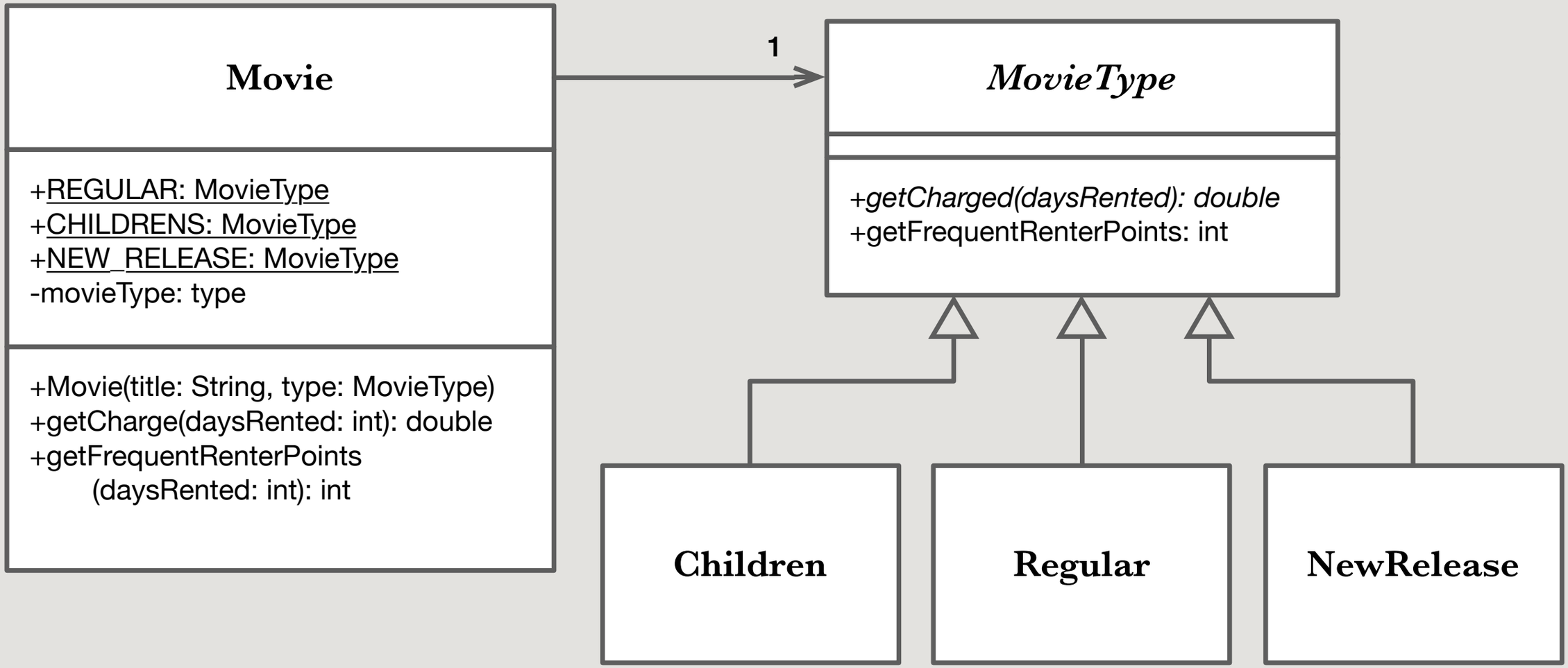


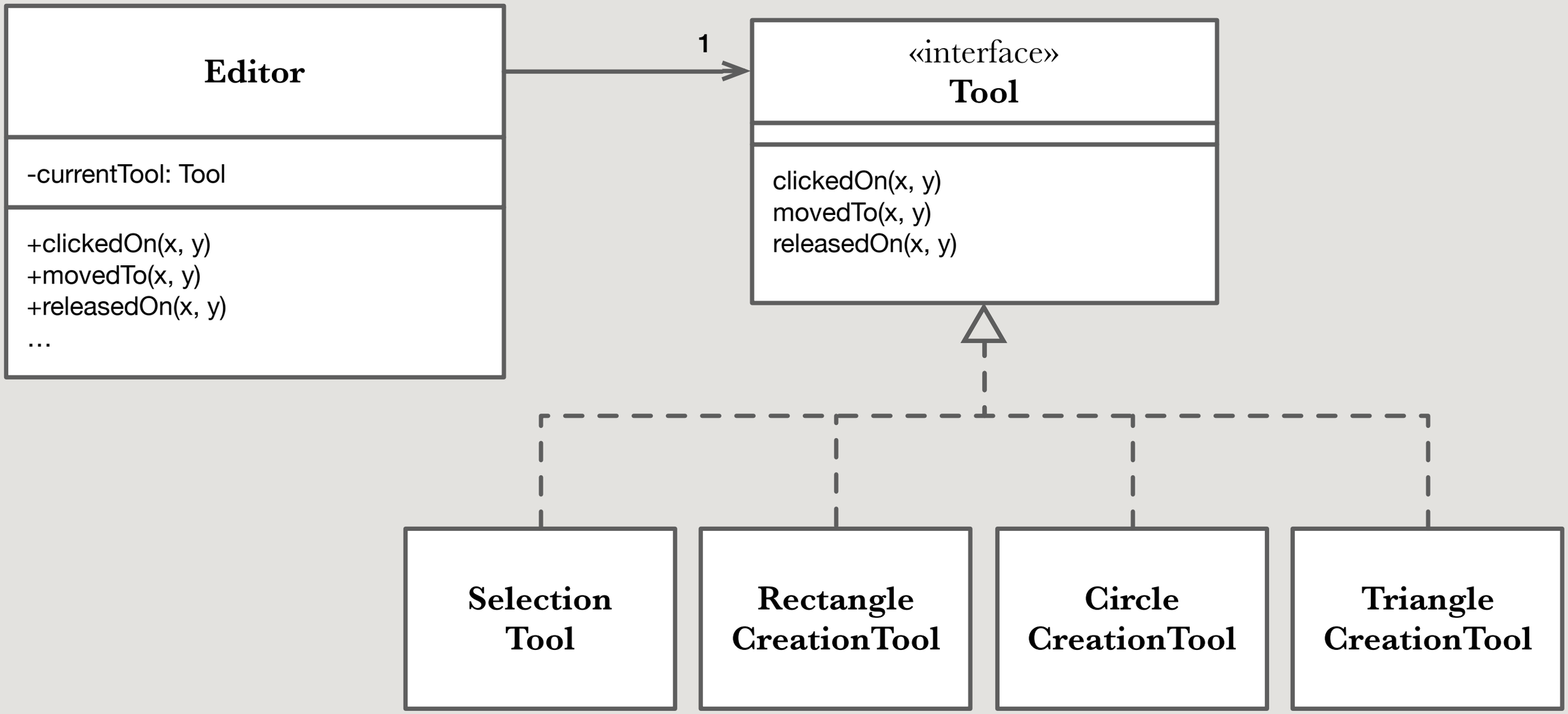
```
List<Statement> statements = new ArrayList<Statement>();

...

Program program = new Program(statements);

Visitor printVisitor = new PrintVisitor();
System.out.println(program.accept(printVisitor));

Visitor interpreterVisitor = new InterpreterVisitor();
program.accept(interpreterVisitor);
```

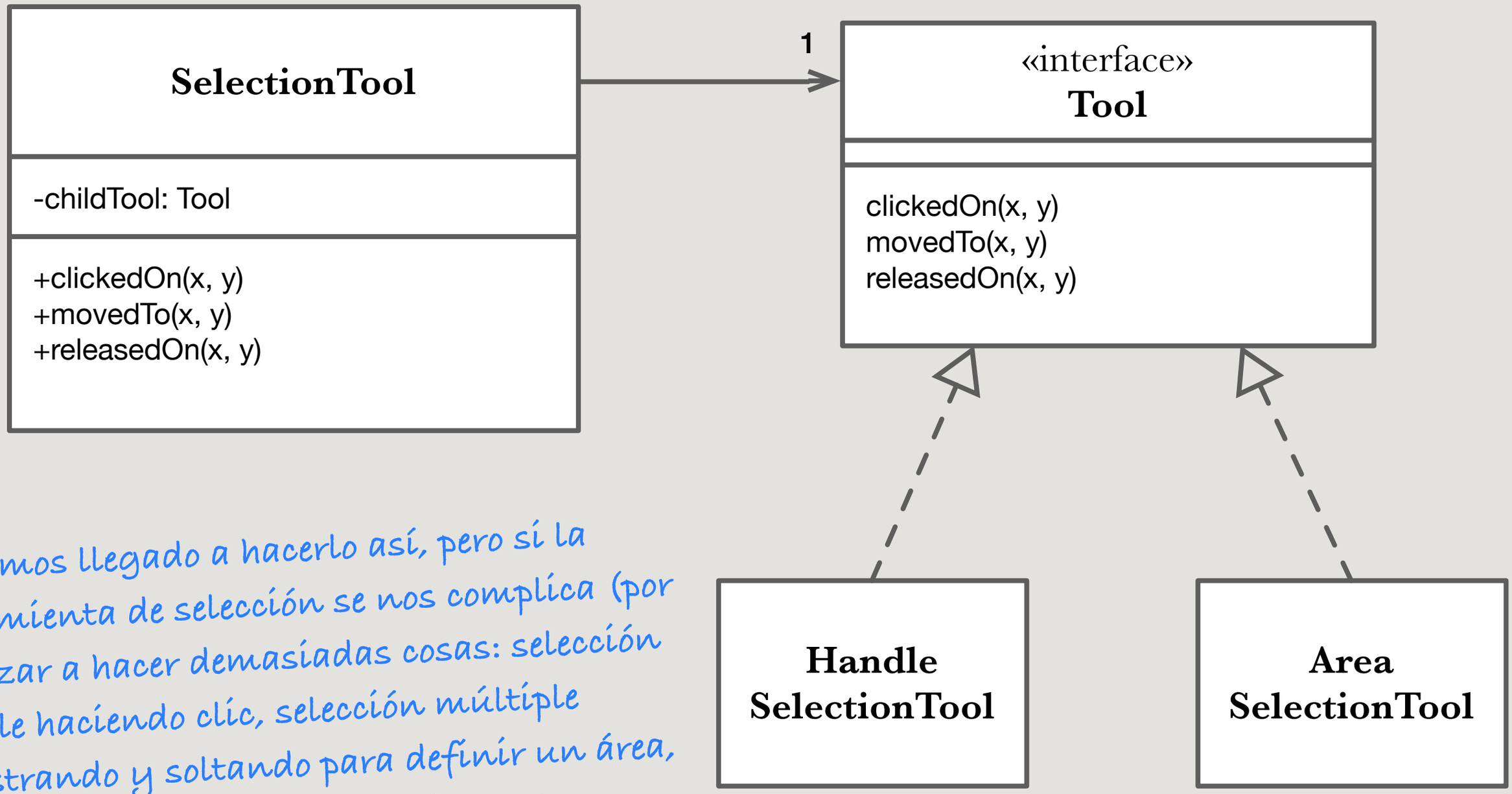
¿Qué es lo que permitía el Visitor? ¿Cuándo tiene sentido usarlo y cuándo no es recomendable?

State

Algunos ejemplos…

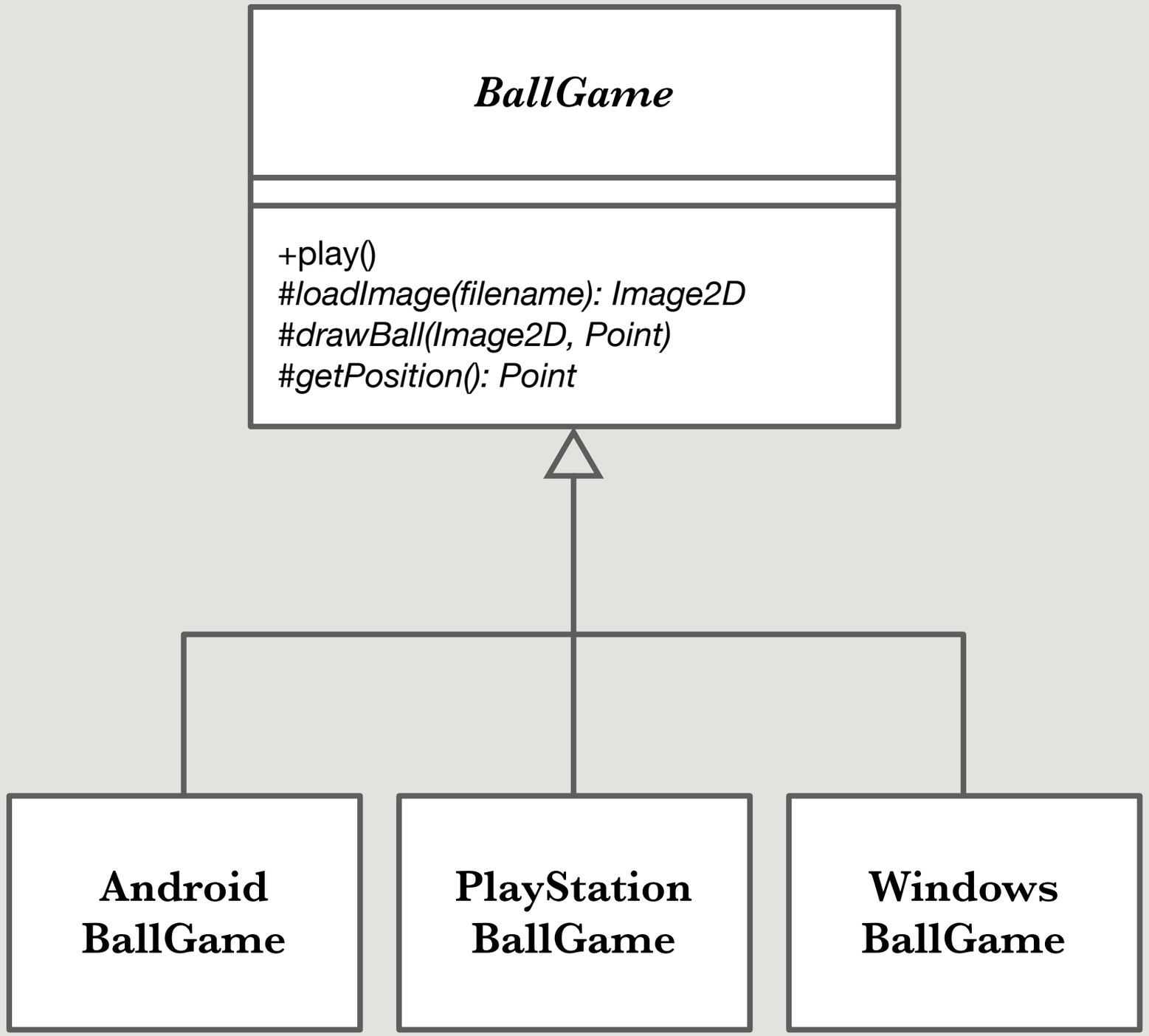

Movie
+REGULAR: MovieType
+CHILDRENS: MovieType
+NEW_RELEASE: MovieType
-movieType: type
+Movie(title: String, type: MovieType)
+getCharge(daysRented: int): double
+getFrequentRenterPoints
(daysRented: int): int
1
MovieType
+getCharged(daysRented): double
+getFrequentRenterPoints: int
Children
Regular
NewRelease


Editor
-currentTool: Tool
+clickedOn(x, y)
+movedTo(x, y)
+releasedOn(x, y)
…
1
«interface»
Tool
clickedOn(x, y)
movedTo(x, y)
releasedOn(x, y)
Selection Tool
Rectangle CreationTool
Circle CreationTool
Triangle CreationTool



No hemos llegado a hacerlo así, pero si la herramienta de selección se nos complica (por empezar a hacer demasiadas cosas: selección simple haciendo clic, selección múltiple arrastrando y soltando para definir un área, mover la figura o figuras previamente seleccionadas...) podríamos volver a aplicar en ella el patrón State y considerar que tiene distintos estados (una especie de «subherramientas» de esta herramienta)

Template Method


BallGame
+play()
#loadImage(filename): Image2D
#drawBall(Image2D, Point)
#getPosition(): Point
Android BallGame
PlayStation BallGame
Windows BallGame

```
public abstract class BallGame
{
    public void play()
    {
        Image2D image = loadImage("Bola.jpg");

        // lógica principal del juego
        for (int i = 0; i < 10; i++) {
            Point point = getPosition();
            drawBall(image, point);
        }
    }

    // Partes variables del algoritmo
    protected abstract Image2D loadImage(String file);
    protected abstract void drawBall(Image2D image, Point point);
    protected abstract Point getPosition();

}
```

Pero, además de ésa en que lo vimos
explícitamente, si no en todas... casi

```
public abstract class AbstractSentence implements Sentence
{
    @Override
    public void execute(Context context)
                            throws ProgramException
    {
        doExecute(context);
        context.incrementIp();
    }

    protected abstract void doExecute(Context context)
                                throws ProgramException;
}
```

```
public abstract class AbstractCreationTool implements Tool
{
    ...

    @Override
    public void release()
    {
        Figure newFigure = createFigure(boundingBox);
        editor.addFigure(newFigure);
        editor.trace("Se creó la figura: " + newFigure);
        editor.toolDone();
    }

    protected abstract Figure createFigure(
                                        BoundingBox boundingBox);
}
```

Etcétera