Analysis, Design, and Software Architecture (BDSA)

*Paolo Tell*

# Architectural and Object Oriented Design

IT University
of Copenhagen

# Outline

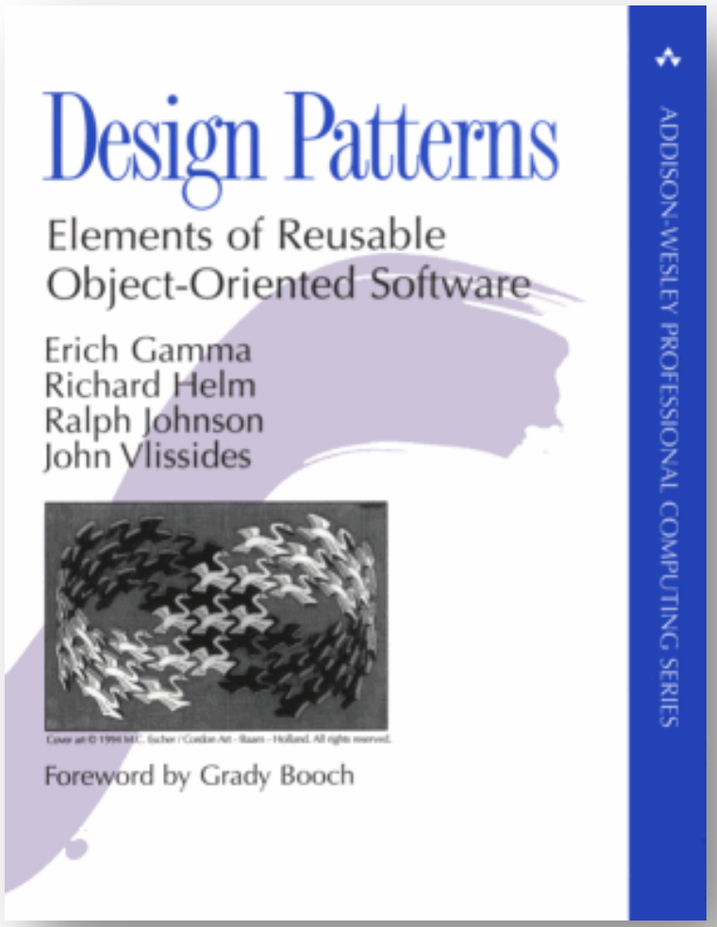
- Literature
  - [OOSE] ch. 7-8

- Topics covered:
  - Design Patterns (II)
  - SDD

Patterns are ways to describe best practices, good design, and capture experience in a way that is possible for others to reuse this experience.

- stylized, abstract description of good practice;
- tried and tested in different environments;
- system organization that has proved to be successful;
- described when to use it – and when not;
- strengths and weaknesses.

# Patterns




Originally proposed by the "Gang-of-Four" (GoF)

Erich Gamma, Richard Helm,
Ralph Johnson & John Vlissides

# Design patterns

# Overview

| Structural | Behavioural | Creational |
|---|---|---|
| **Adapter** | Strategy | Builder |
| **Façade** | State | Prototype |
| **Composite** | **Command** | Factory method |
| **Decorator** | Observer | Abstract factory |
| **Bridge** | Memento | |
| Singleton | Interpreter | |
| **Proxy** | Iterator | |
| Flyweight | **Visitor** | |
| | Mediator | |
| | Template method | |
| | **Chain of responsibility** | |

# Adapter



- Purpose
  - Adapt the interface of an already existing class to the interface that a client is expecting.

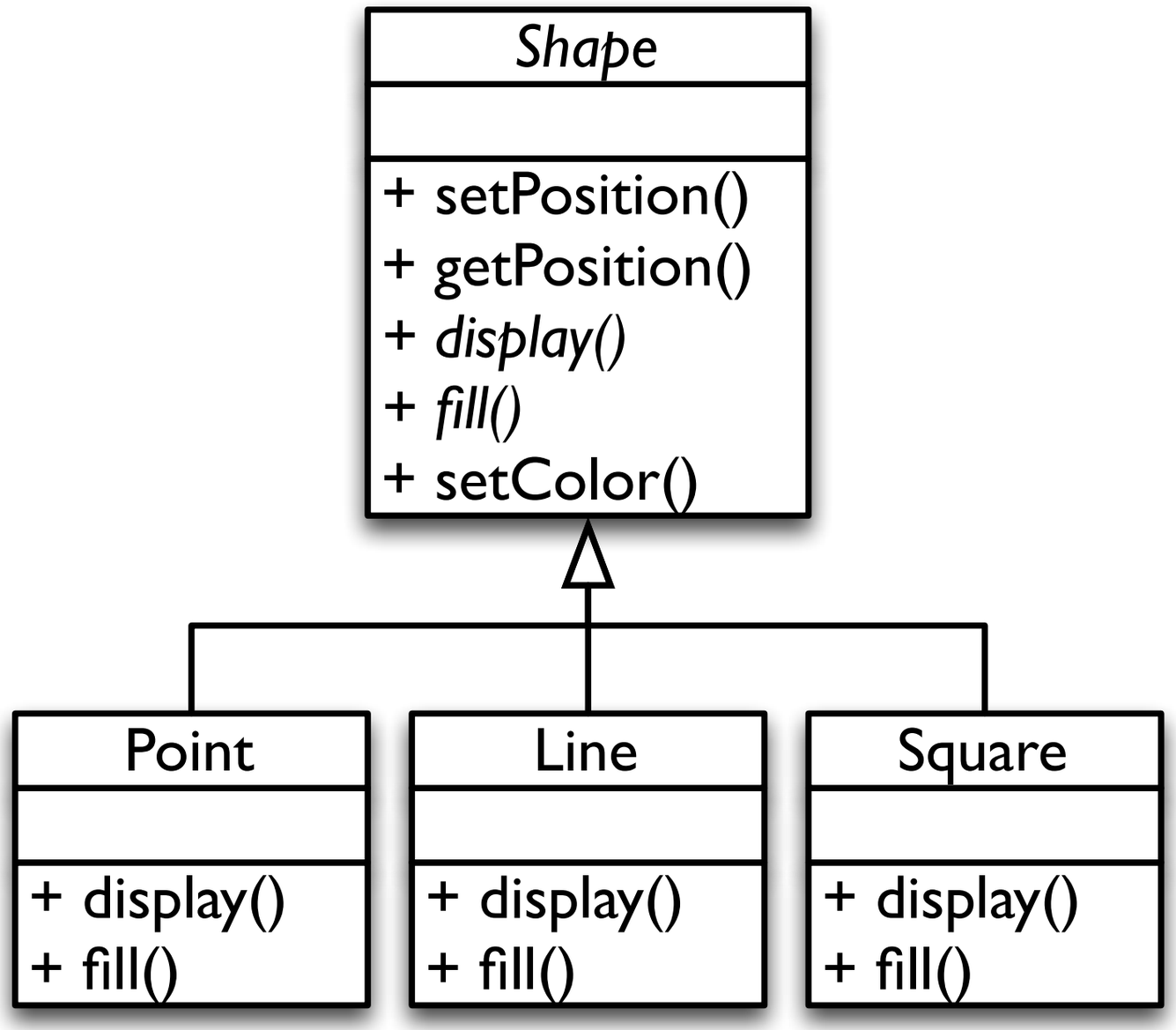
- (Imagine a power socket adapter. You have a schuko plug and you are in Denmark, how do you fix it? You place an adapter!)

# Purpose



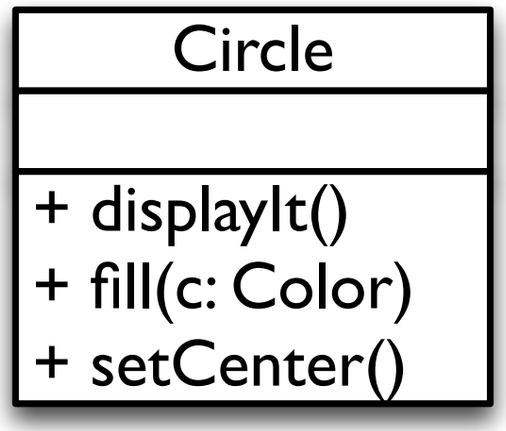
- Data
  - A client is expecting:
    - a class/object that provides some services
    - with a given interface (name of the class, methods, and their signatures)
  - A class that
    - would provide those services
    - but has an interface different from the expected one
- The adapter adapts the different interface to the one expected

# Example (1/3)


Shape
+ setPosition()
+ getPosition()
+ display()
+ fill()
+ setColor()
Point
+ display()
+ fill()
Line
+ display()
+ fill()
Square
+ display()
+ fill()

# Example (2/3)

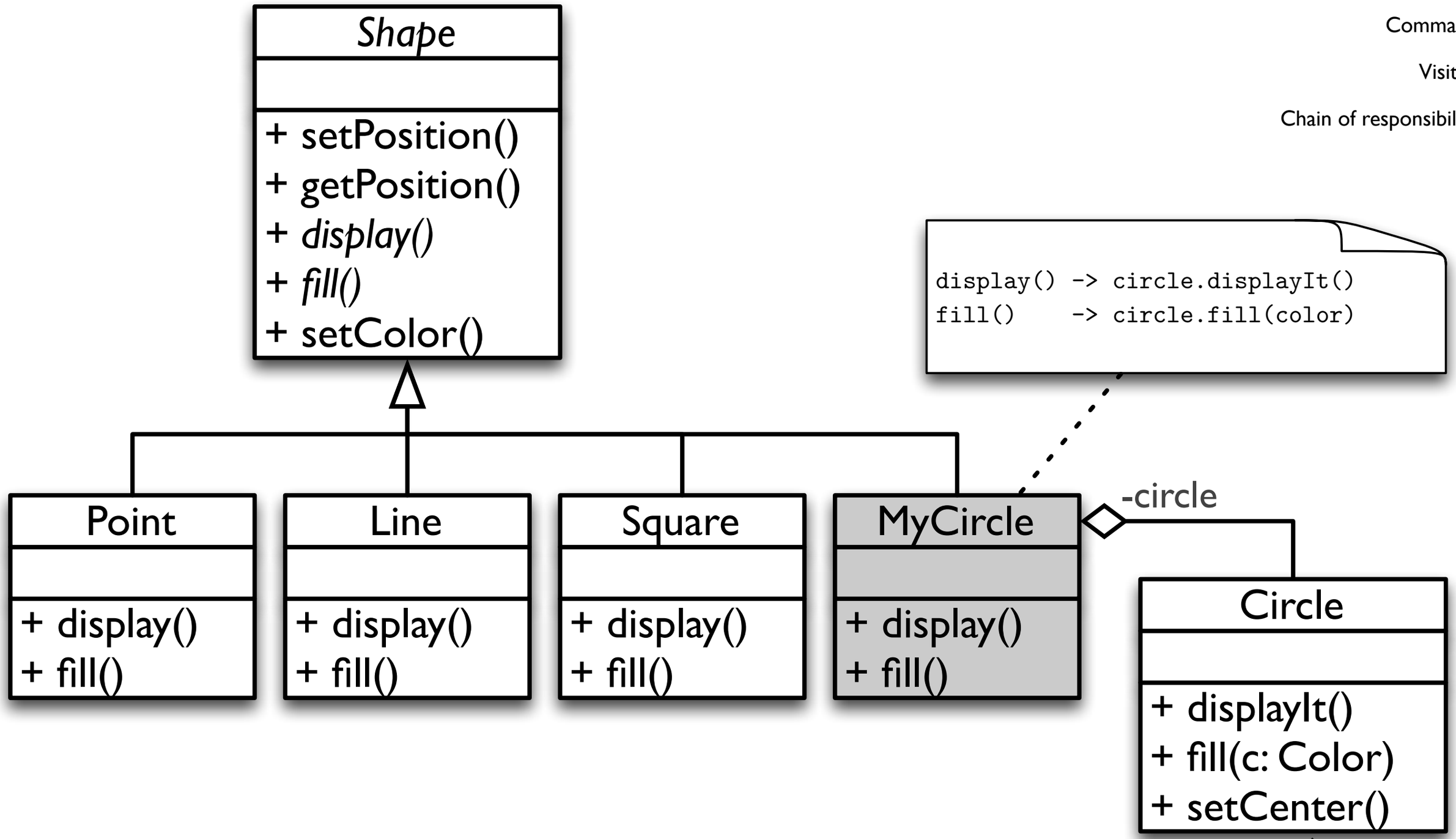
- We want to now add a `Circle`
  - Shall we implement it from scratch?
    - Reuse!!!
  - Shall we use the class `Circle`?
    - But it has a different interface :(




- Let's create an adapter

- (Note! It is important to highlight that we cannot modify `Circle`)
  - We do not have the source code
  - It is already used as is
  - …

# Example (2/3)


Shape
+ setPosition()
+ getPosition()
+ display()
+ fill()
+ setColor()
display() -> circle.displayIt()
fill() -> circle.fill(color)
Point
+ display()
+ fill()
Line
+ display()
+ fill()
Square
+ display()
+ fill()
MyCircle
+ display()
+ fill()
-circle
Circle
+ displayIt()
+ fill(c: Color)
+ setCenter()

# Considerations



- `Circle` cannot be a subclass of `Shape`
- `Circle` must have a specific interface (in this case "imposed" by the hierarchy descending from `Figure`)
- `MyCircle` adapts the interface of `Circle` to the expected one

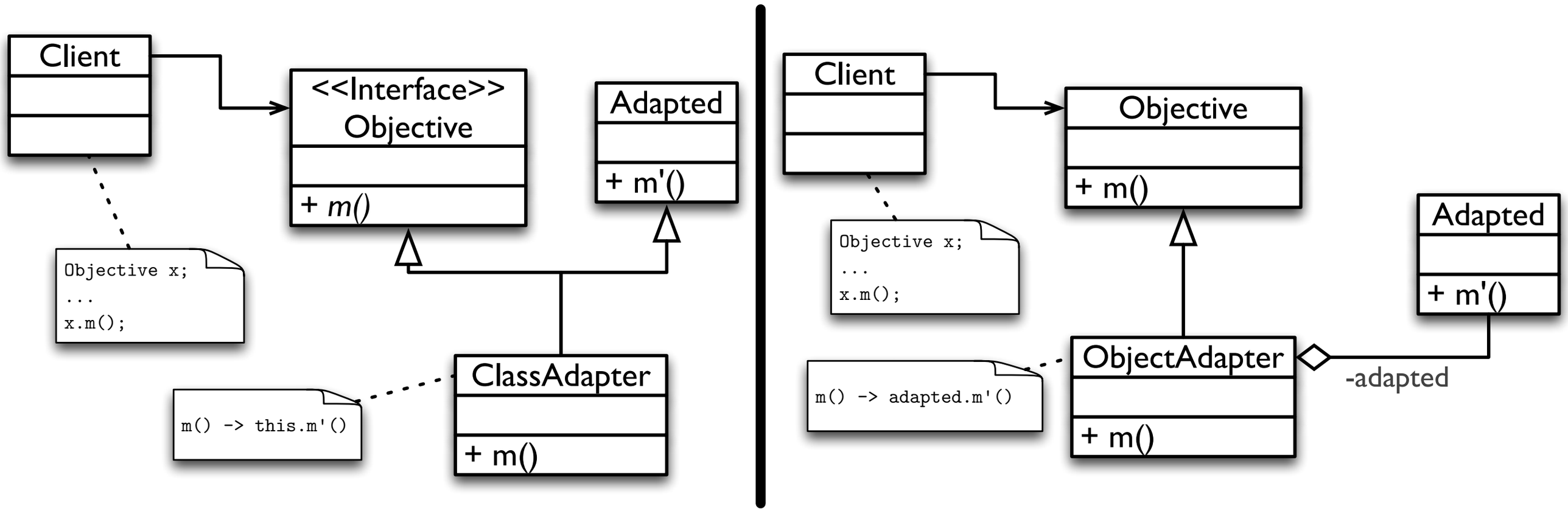
# 2 types of adapters

- Object adapter
  - Based on delegation/composition

- Class adapter
  - Based on inheritance
  - The adapter inherits both from the expected interface as well as from the adapted class
  - No multiple inheritance … the expected interface must be an interface, not a class

- (Are these 2 different patterns?)

# Class adapter versus object adapter



# Adapter: summary and comments

- Adapter is used when a class needs to be reused, that maybe was developed also with reuse in mind, but with the "wrong" interface (different from what we need)

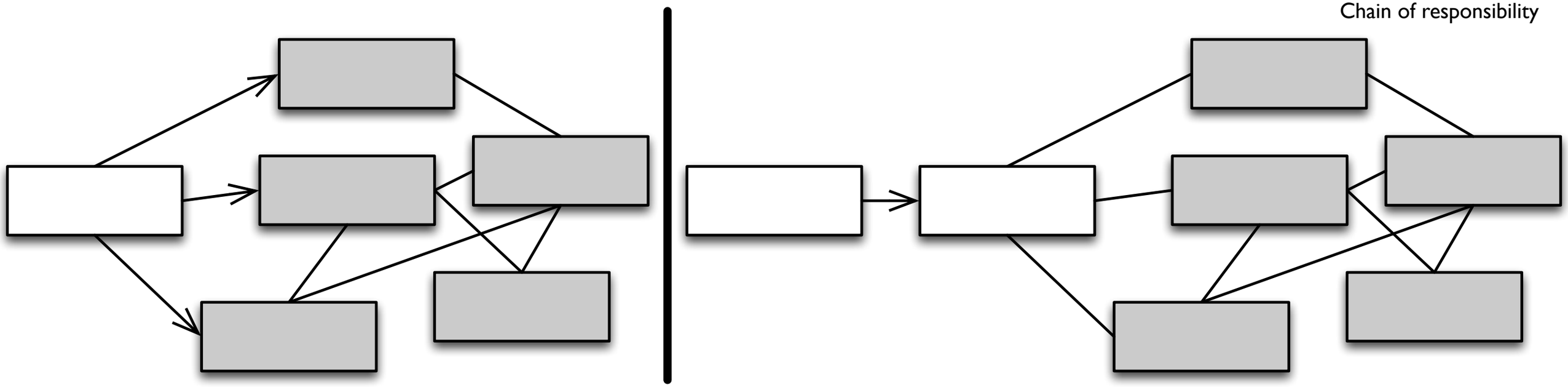
- Therefore, from now on, we can ignore if a class has an interface that does not match our needs: we just put an adapter in front!

- The Adapter could also slightly modify the behaviour of the `Adapted` (but for "heavy" modifications there are other patterns)

- Also an object adapter could adapt several classes …

# Façade





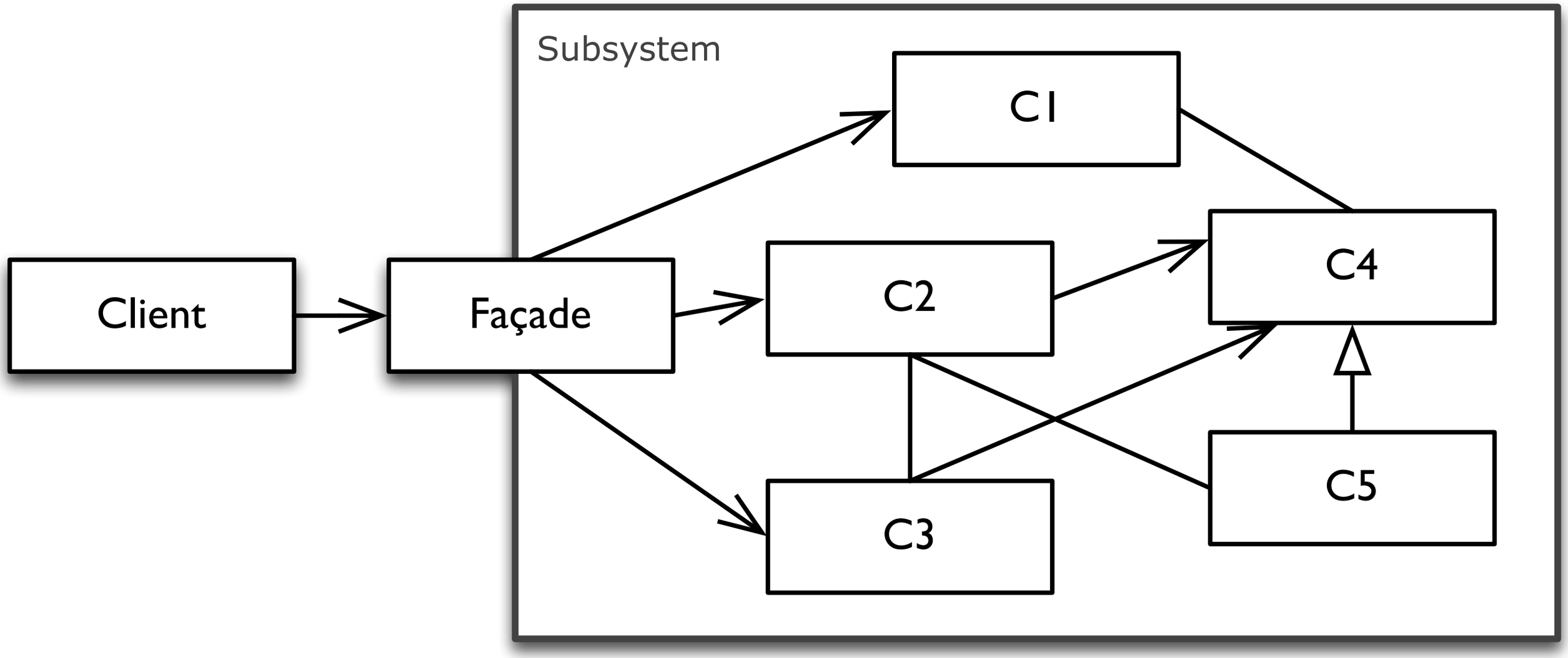
- Purpose
  - Simplify the use of a system
  - Provide a unified interfaces for a group of "dispersed" functionalities from a multitude of interfaces/classes

# Examples

- Classes for 3D graphics to draw 2D images
- Compiler
  - Classes like `Parser`, `Scanner`, `Token`, `SyntacticTree`, `CodeGenerator`, … versus
  - A class `Compiler` with a method `compile()`
- Quick installation manual of the printer
  - Often it works
  - If it does not, there is always the complete manual

# Façade diagram





# Advantages and Comments

- It is an easy pattern
- Often it is used with static methods
- Promote loose coupling between the `Client` and the `Subsystem`
- Hide from the `Client` the components of the `Subsystem`
- The `Client` can still, if necessary, can still use directly the classes of the `Subsystem`

# Façade versus Adapter



- Both of them are wrappers

- Both are based on an interface, but:
  - Façade simplifies it
  - Adapter converts it

# Composite



- Purpose
  - Compose objects in recursive tree structures to represent whole/part hierarchies
  - and allow clients to treat both complex as well as single/simple objects in the same uniform way
- It is a nice pattern

# Example (1/5)



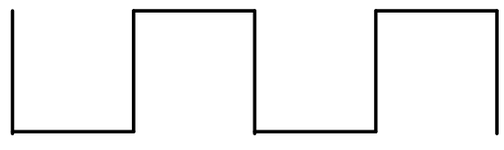
- Drawing program

- Based on these requirements: the system shall support
  - the creation of simple shapes (Point, Line, Circle)
  - the dynamic grouping of simple shapes into complex ones
  - treat the complex shapes as they were simple ones

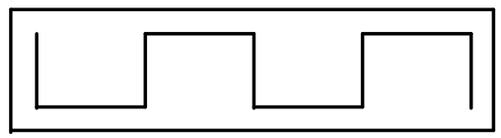
# Example (2/5)



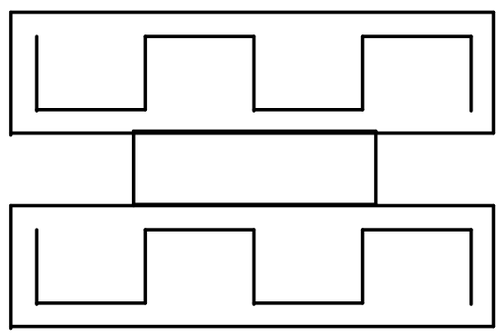
- Group 4 lines into a rectangle



- Group N lines into a coil

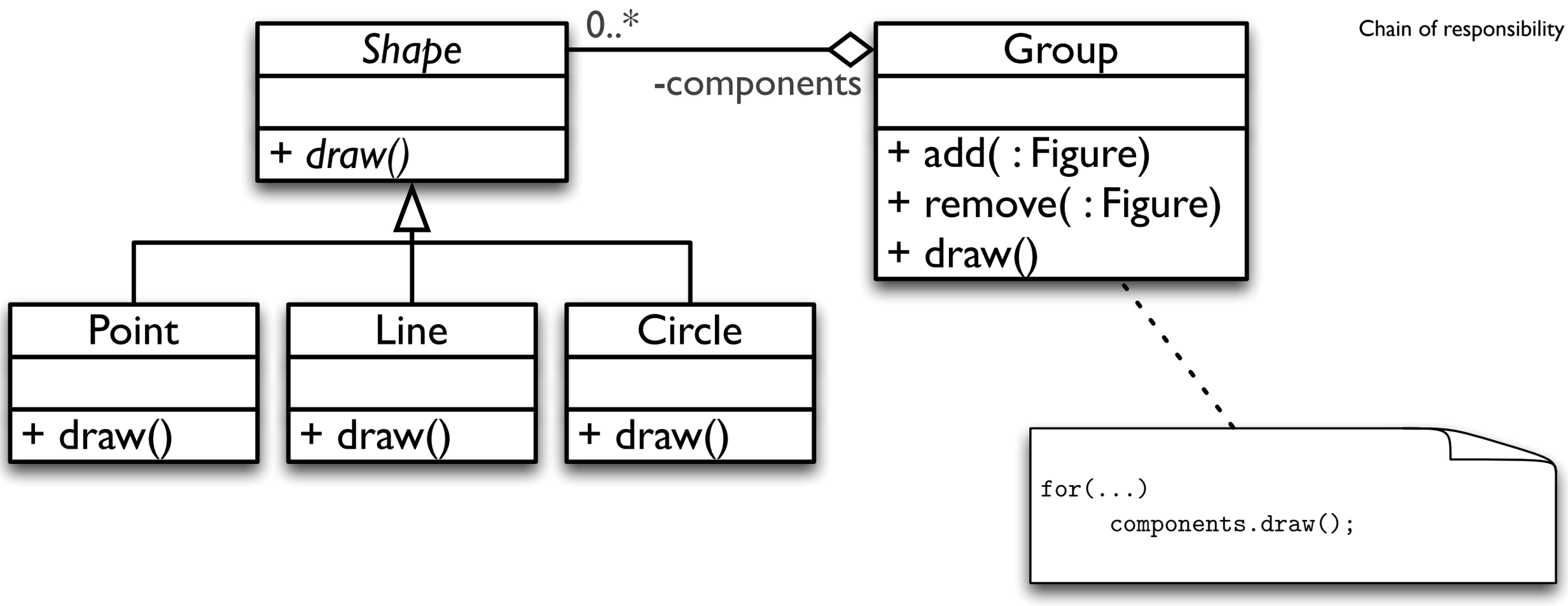


- Group a coil and a rectangle



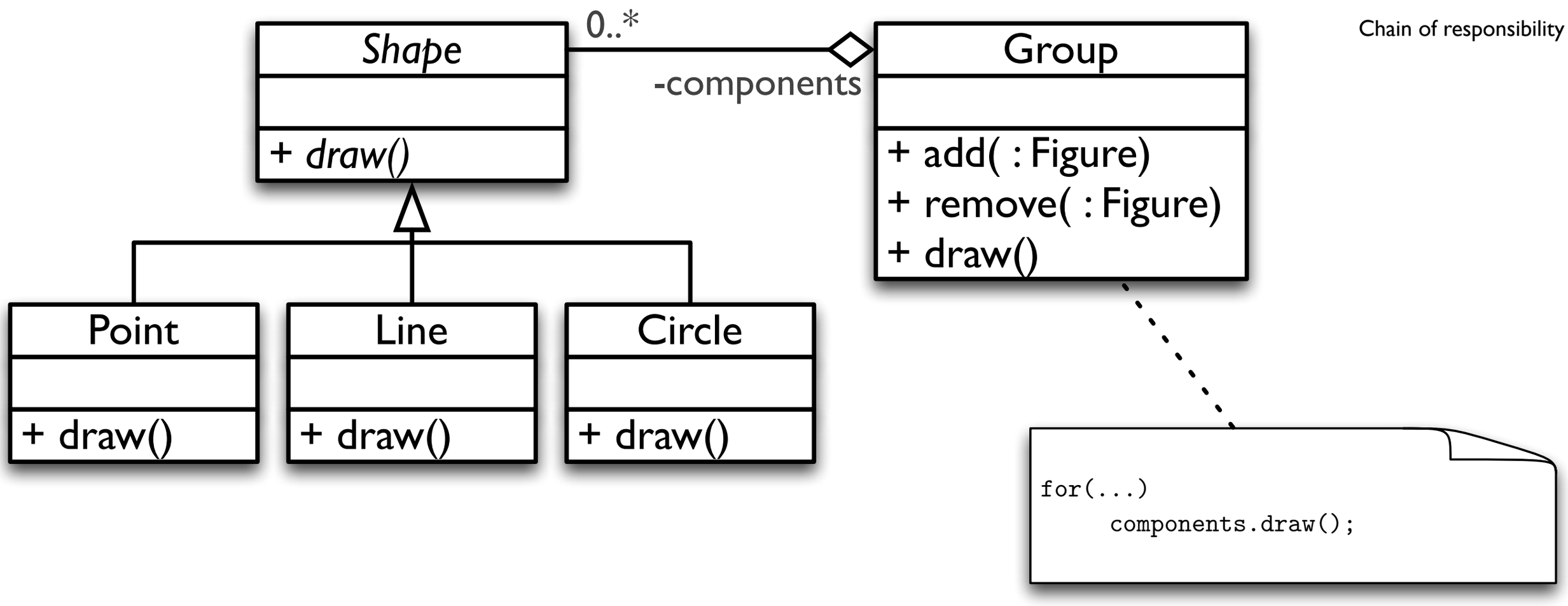
- …
- … let's try to model this with UML

# Example (3/5)




- 1st attempt. What is the problem?

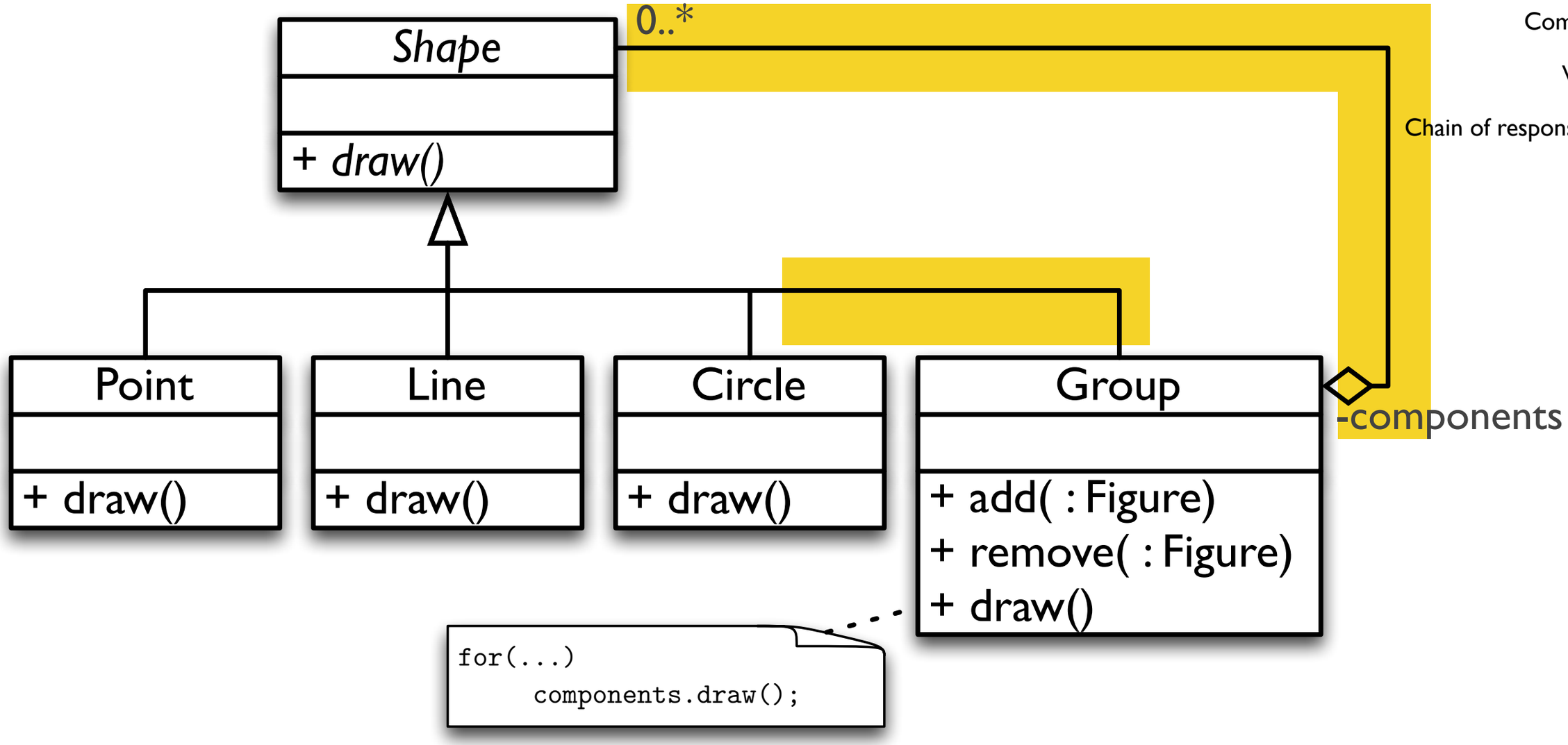
# Example (3/5)




- Complex shapes (`Group`) and simple shapes (`Shape`) are handled in different ways

# Example (4/5)






- 2nd attempt. `Group` subclassing `Shape`!
- This is the base idea behind the composite

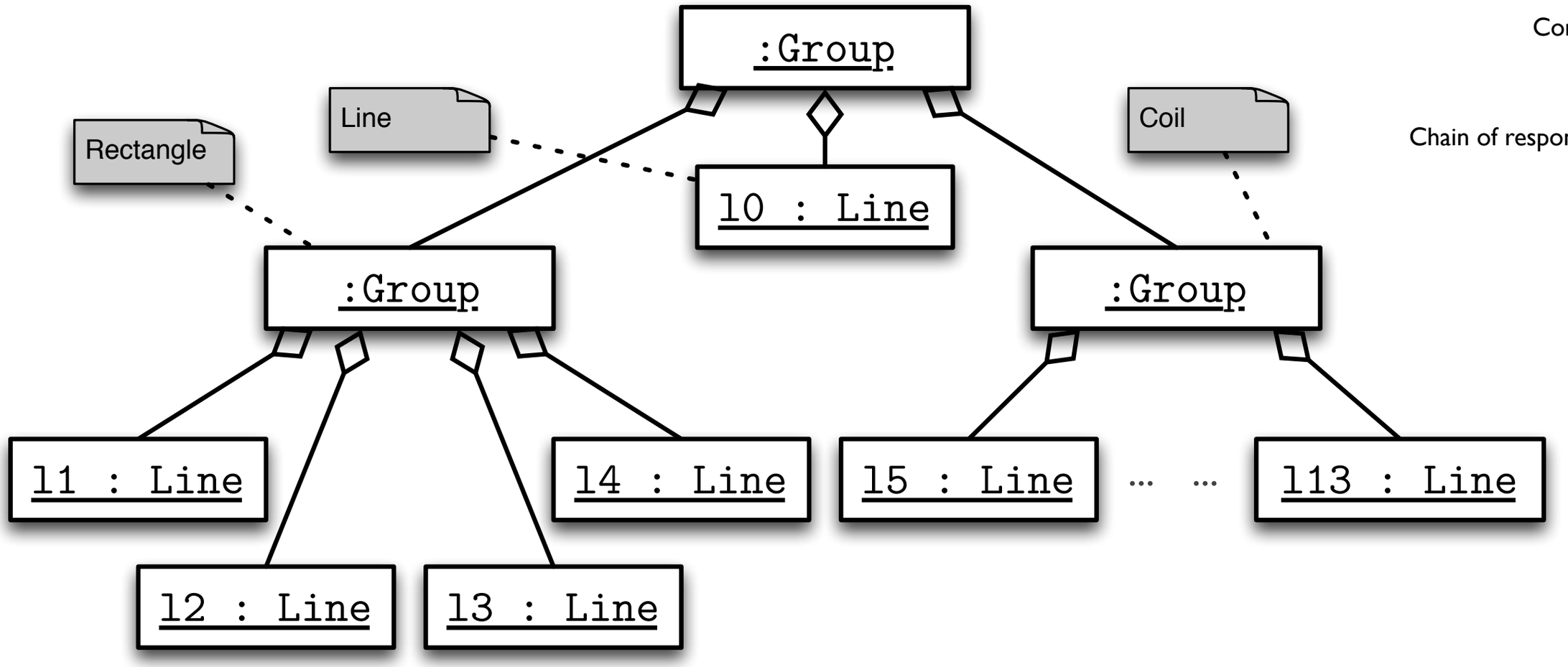
# Example (5/5)



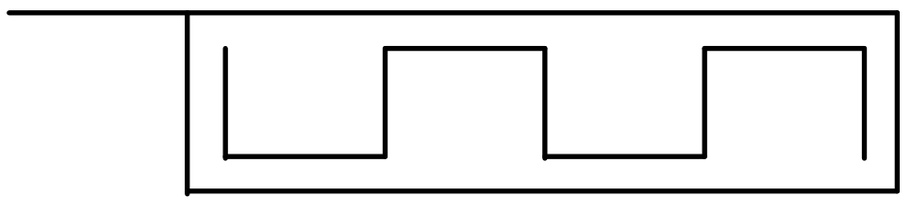
- A point/line/circle is a figure …
- … handle complex objects as they were simple objects …

- A group is a figure!
- The recursion is:
  - `Group` contains `Figure`,
  - and because a `Figure` can be a `Group` (a `Group` is a subclass of `Figure`)
  - a group can contain groups …

# Object diagram: tree!

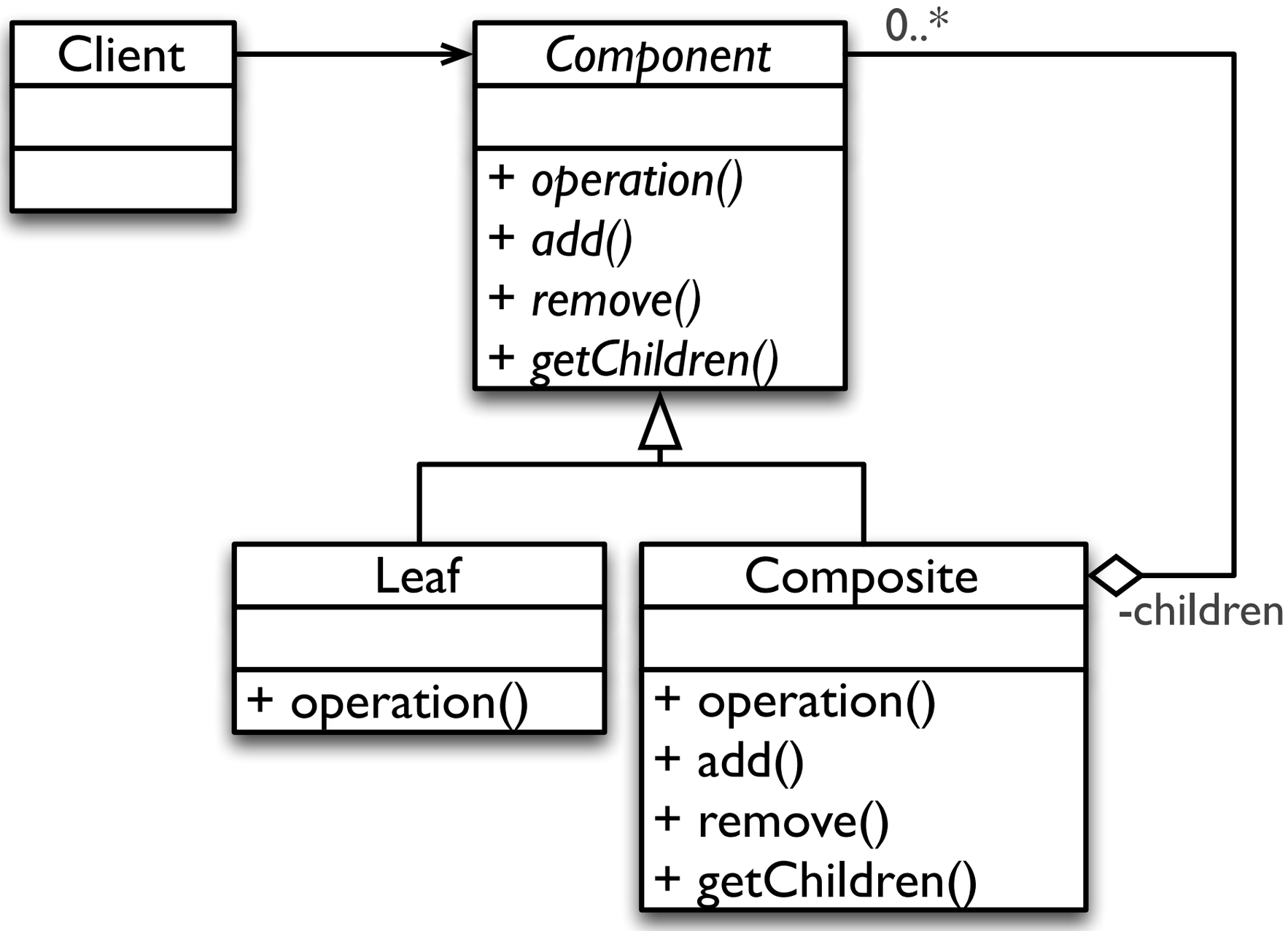



- To represent this figure, we need:
  - 3 instances of `Group`
  - 14 instances of `Line`



# Composite diagram





# Comments



- It simplifies the client, which handles complex and simple objects in the same manner
  - In all places in which a client is expecting a simple object, we can pass him a complex one

- `Component` interface
  - Maximizing for flexibility: the `Client` cannot distinguish between atomic objects and complex ones. However, what will be the implementation of `add` on a leaf?
  - Minimizing for security: no problem for the `add` on `Leaf`, but the interfaces are different.

- Leaf and Composite could:
  - be abstract
  - be extended

- Cycles can generate infinite recursion when visiting the structure
  - track the visited nodes

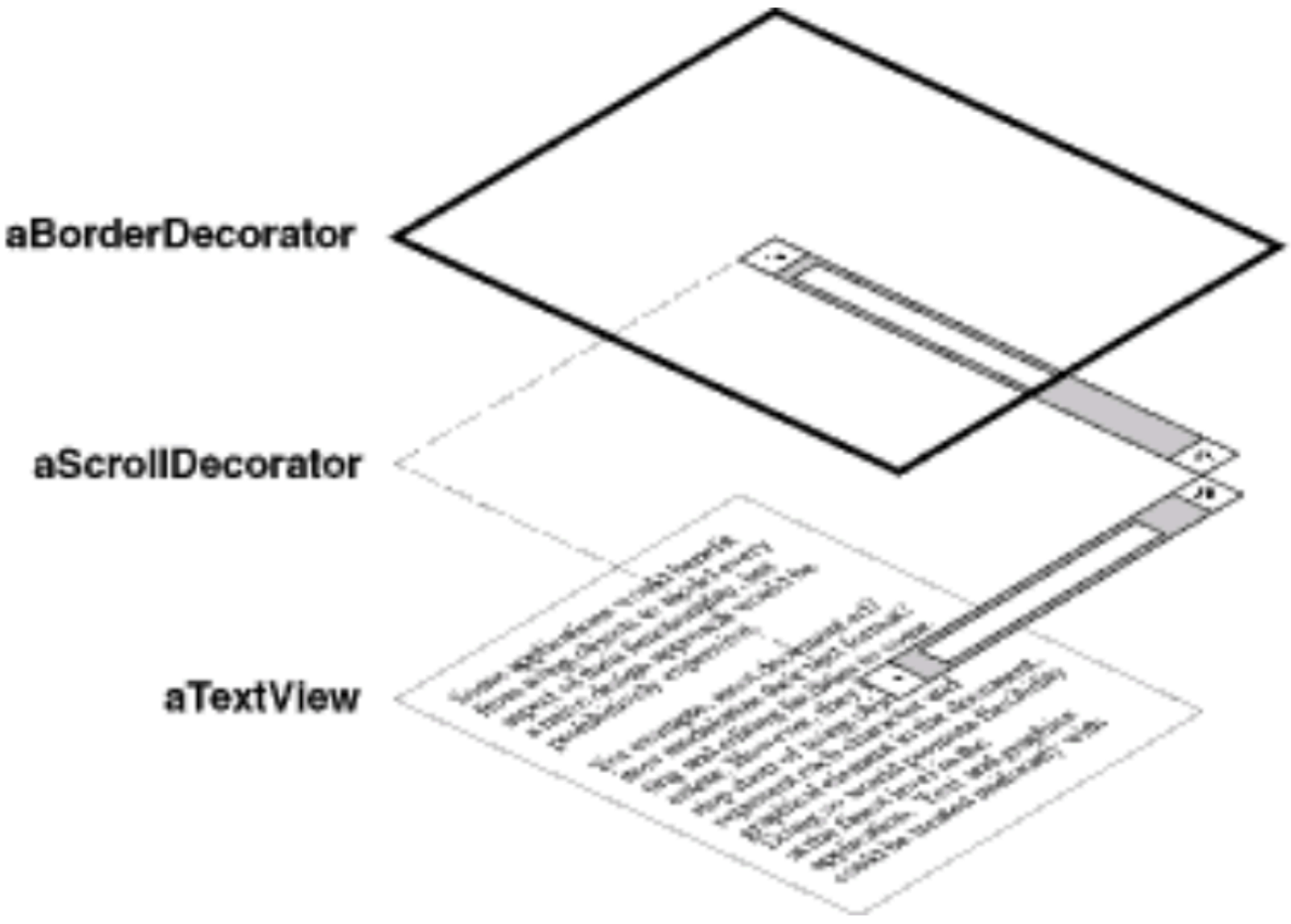
# Decorator

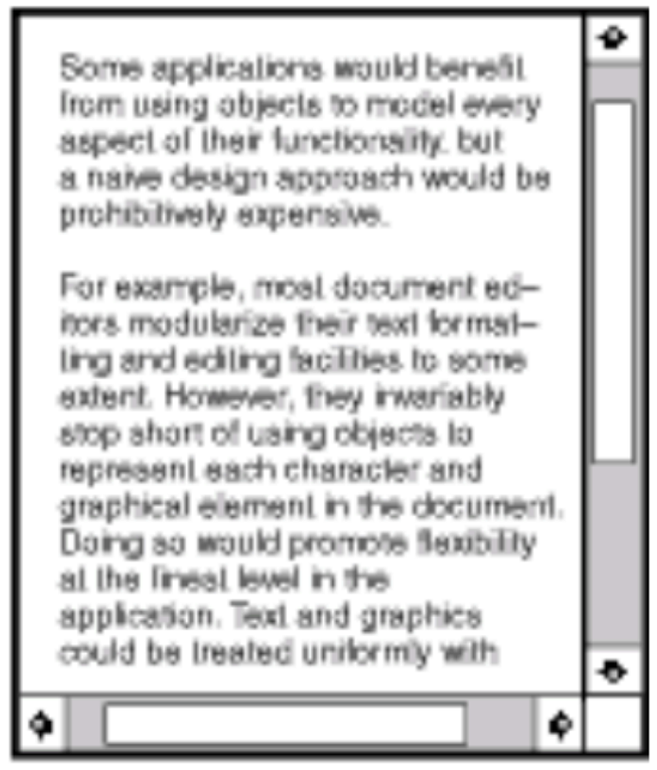


- Purpose
  - Dynamically add to an object some functionalities/responsibilities/behaviour
- (More flexible) Alternative to the creation of subclasses aimed at extending functionalities
  - also specialization through inheritance can add functionalities/responsibilities/ behaviour

# Example



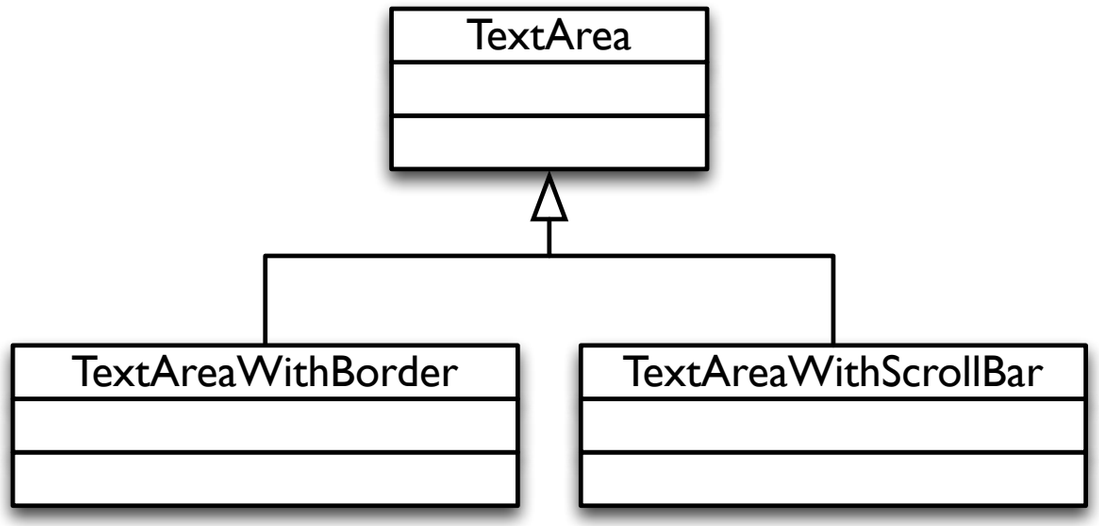



- How can we obtain this?

# Statically add responsibility?






- Inheritance
  - Create the instance of the needed class
  - Cons
    - `TextAreaWithBorderAndScrollBar`?
      `TextAreaWithScrollBarAndBorder`?
      - Other 2 subclasses?
    - Static: I cannot add functionalities after the creation of the instance
    - Other components? (`Window`) Duplication!

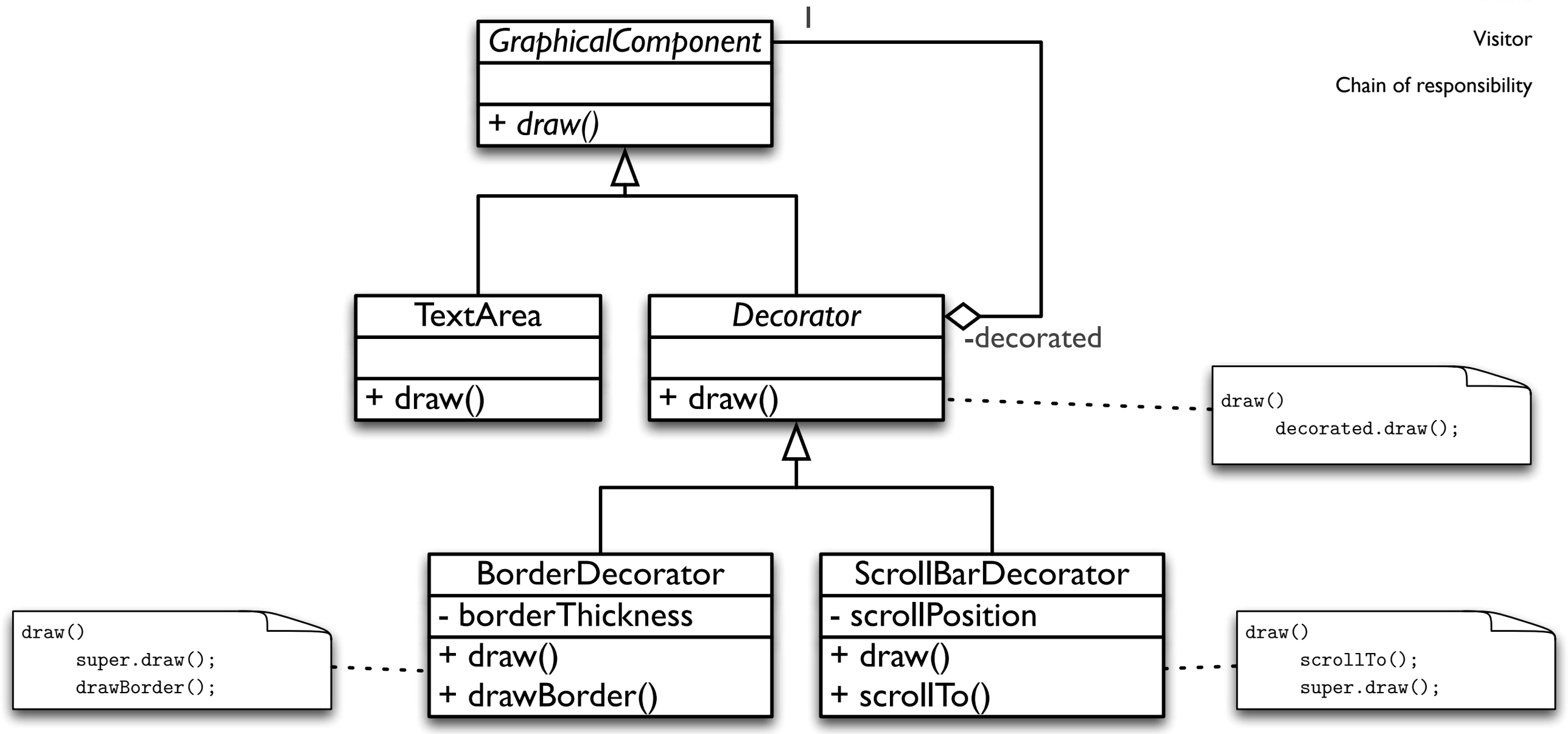
# Dynamically add responsibility



- Encapsulate the object to "decorate" (the text area) in another object responsible to handle the "decoration" (the border, the scroll bar): the "decorator"
- The decorator transfers/delegates and does something extra (before or after)
- The decorator has the interface conforming to the decorated object, hence, it is transparent to the client

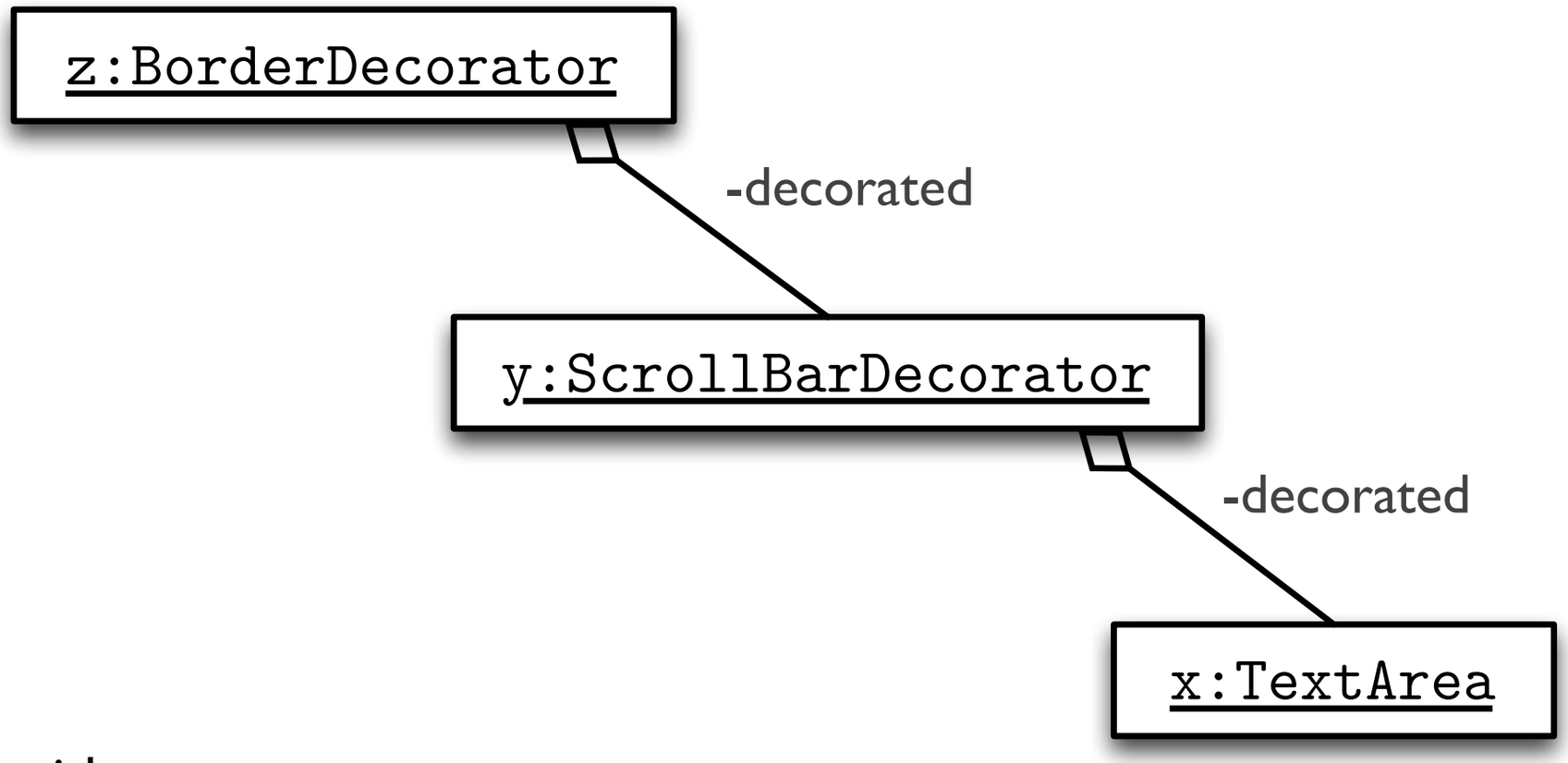
# Let's revisit the example





# Object diagram



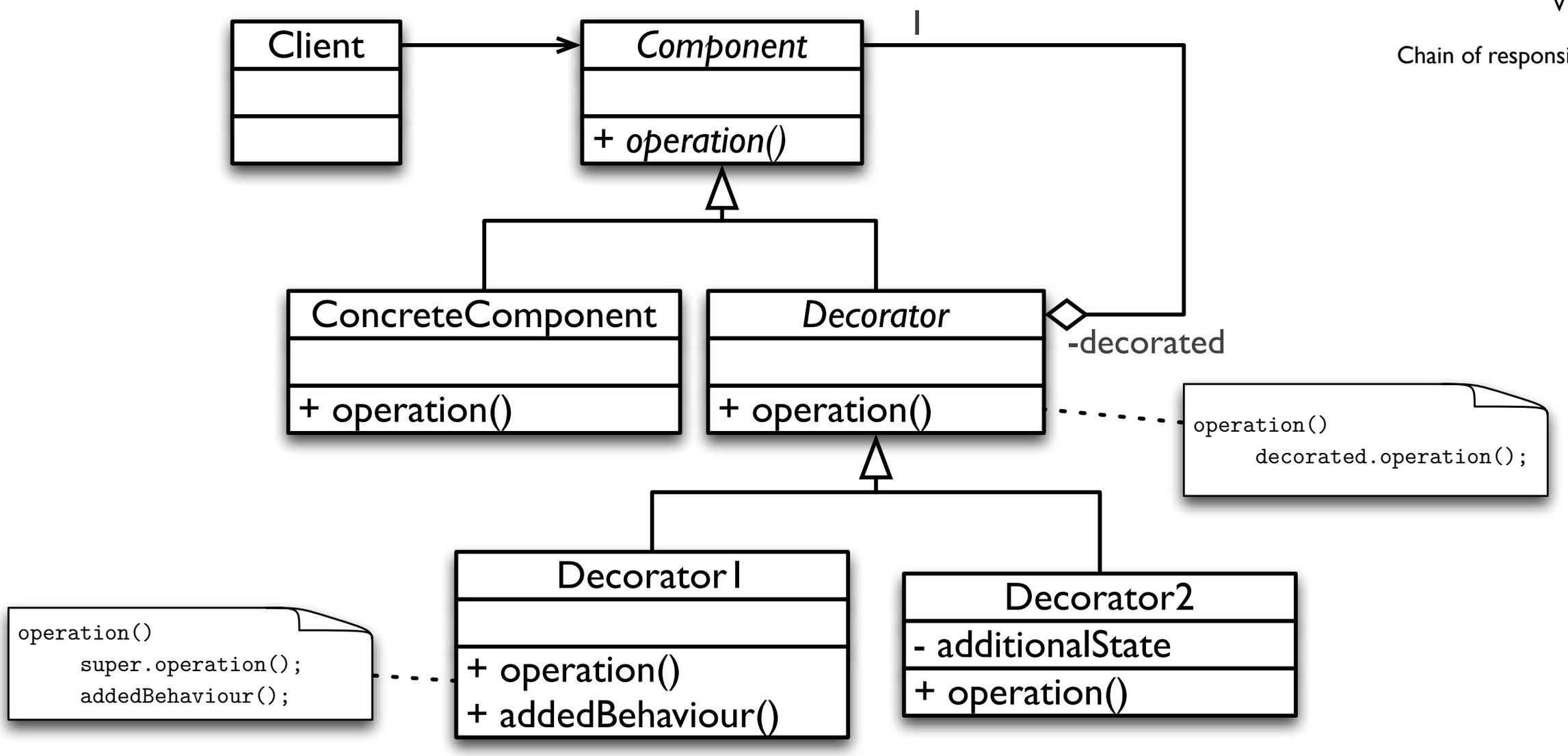



- Chain with
  - Multiple decorations
  - Eventually, an instance of `TextArea` would be:

```
new BorderDecorator( new ScrollBarDecorator( new TextArea() ) )
new ScrollBarDecorator( new BorderDecorator( new TextArea() ) )
```

# Decorator diagram





# Decorator versus inheritance



- Dynamic (run time)
- Without constraints for the client (any combination of decorators can be created)
- Add responsibility to a single object
- Avoids combinatorial explosion

- Static (compile time)
- Constraints the client (combinations of decorators which were not thought cannot be created)
- Add responsibility to (all the instances of) a class
- Can cause combinatorial explosion

# Decorator versus composite



- A decorator can be seen as a degenerate composite with one single component (see the multiplicity of the aggregation)

- However,
  - the purpose of a composite is to aggregate objects
  - the purpose of a decorator is to add responsibilities/behaviours/functionalities to an object

- They can be used together

# Decorator versus adapter



- Decorator
  - enriches the functionalities without modifying the interface
  - it modifies the responsibilities of an object, not its interface
- Adapter
  - modifies the interface of the object
  - could also, in a limited manner, enrich the responsibilities/functionalities/ behaviours of the adapted object

# Bridge



- Purpose
  - Separate an abstraction from its implementation
  - so that they can vary independently
- WHAT???
- It is a nice useful pattern but is not that intuitive
- In the GoF it is not so well explained

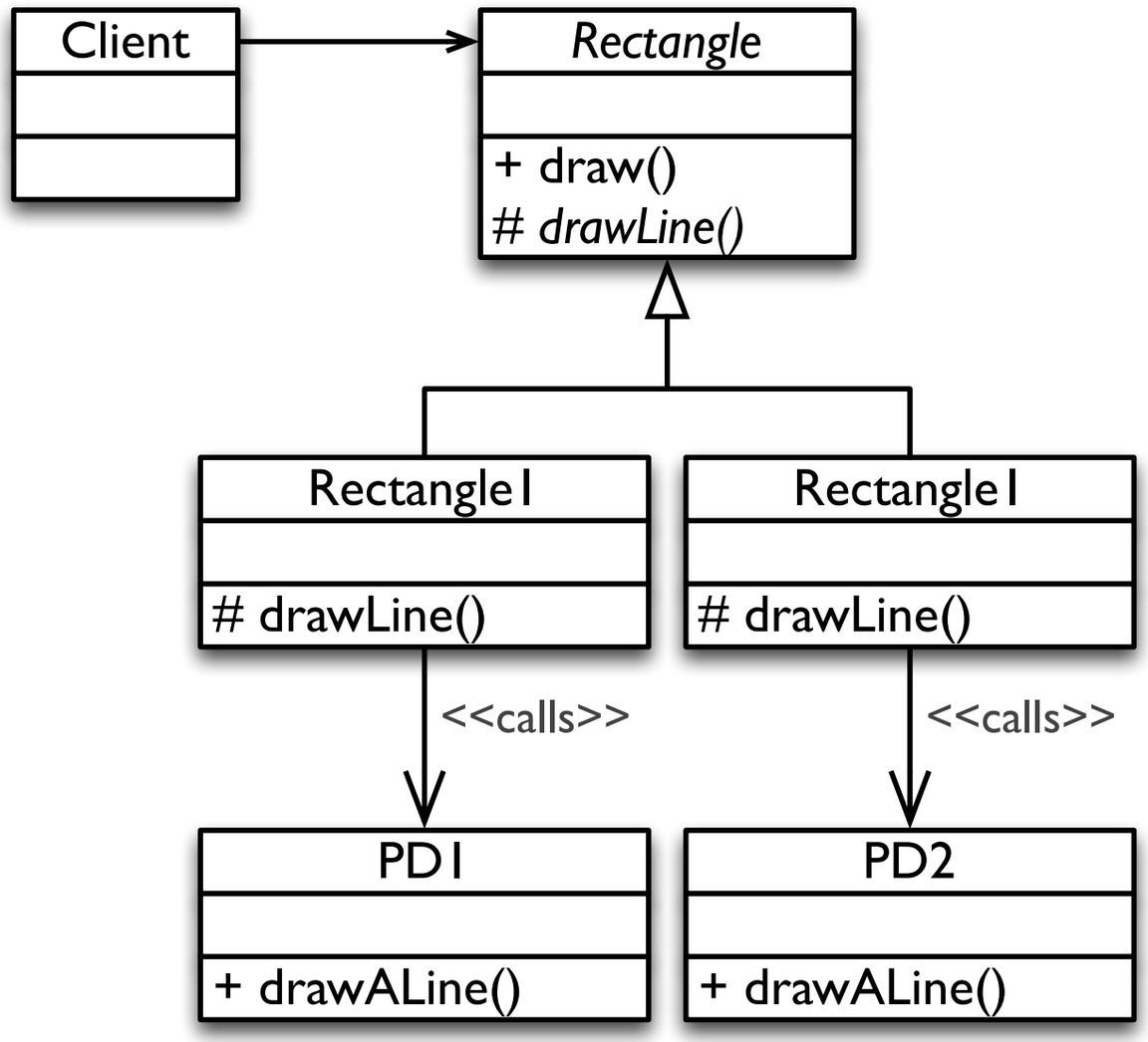
# Example



- We need to design a program that draws rectangles
- alternatively using one of two existing graphical programs
  - `PD1: drawALine(x1, y1, x2, y2)`
  - `PD2: drawALine(x1, x2, y1, y2)`
- I know which one to use when I have to instantiate the rectangles
  - The client does not have to be bothered about it

# 1st naive solution






- 2 types of rectangles, one using `PD1` the other `PD2`
- During instantiation, I know whether I need `Rectangle1` or `Rectangle2`

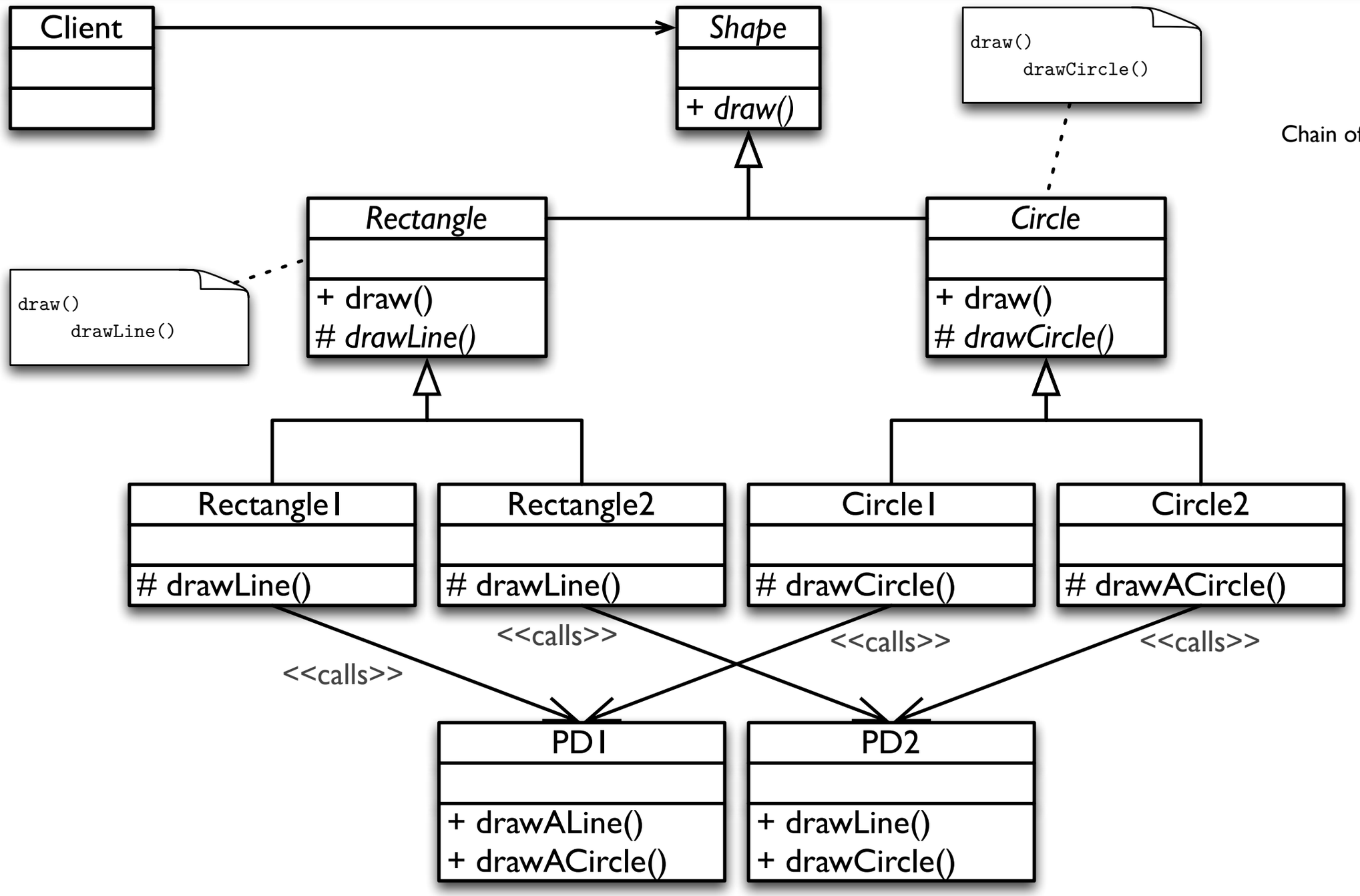
# … but the client also …



- … wants to draw circles
  - luckily PD1 and PD2 have
  - `PD1: drawACircle(x, y, r)`
  - `PD2: drawCircle(x, y, r)`

- … and also wants to be able to abstract
  - hence, handle circles like rectangles

# 2nd solution … based on the first



# Comments



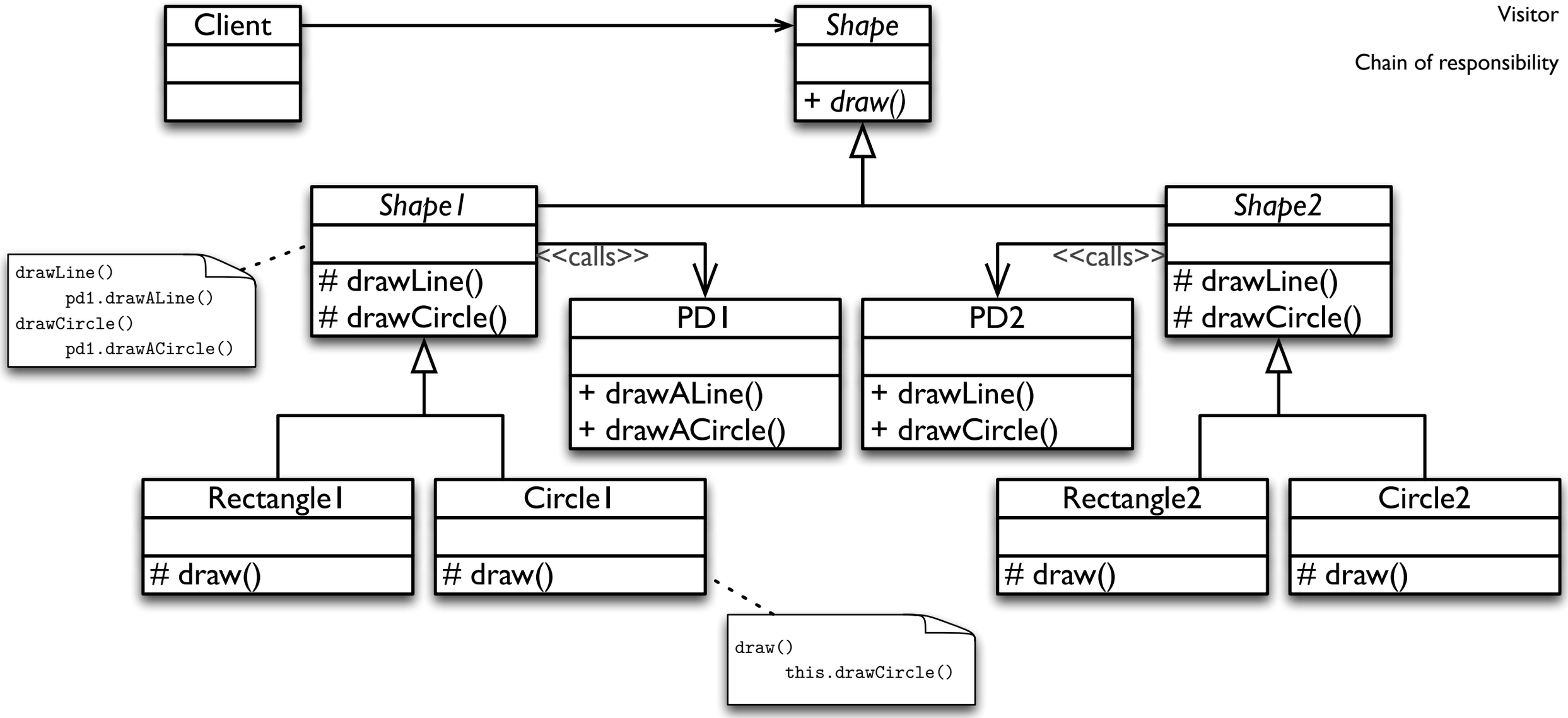
- What if I need to manage other shapes?
  - For each additional shape I have to add a small hierarchy (3 classes)
    - the abstract one
    - and one per PD

- And what if I need to manage other graphical programs?
  - For each additional PD, I have to add:
    - a class `PDi`
    - and a class for each hierarchy

- It seems to be working … but it is quite ugly: with N shapes and M PDs, we have N*M classes at the 3rd level.

# Alternative





# Comments



- I changed the "order":
  - from: first type of shape then PD version
  - to: first PD version then type of shape

- But not much has changed
  - It is still N*M …

# Let's look back



- Purpose
  - Separate an abstraction from its implementation
  - so that they can vary independently

- Now we can understand what it means
  - Abstraction: the shapes
  - Implementation: the graphical programs

- With the current solution there is …
  - Very high coupling between figures (abstraction) and the PDs (implementation)
  - I cannot make them vary independently
  - And there is also a lot of redundancy

# Is it a high quality solution?

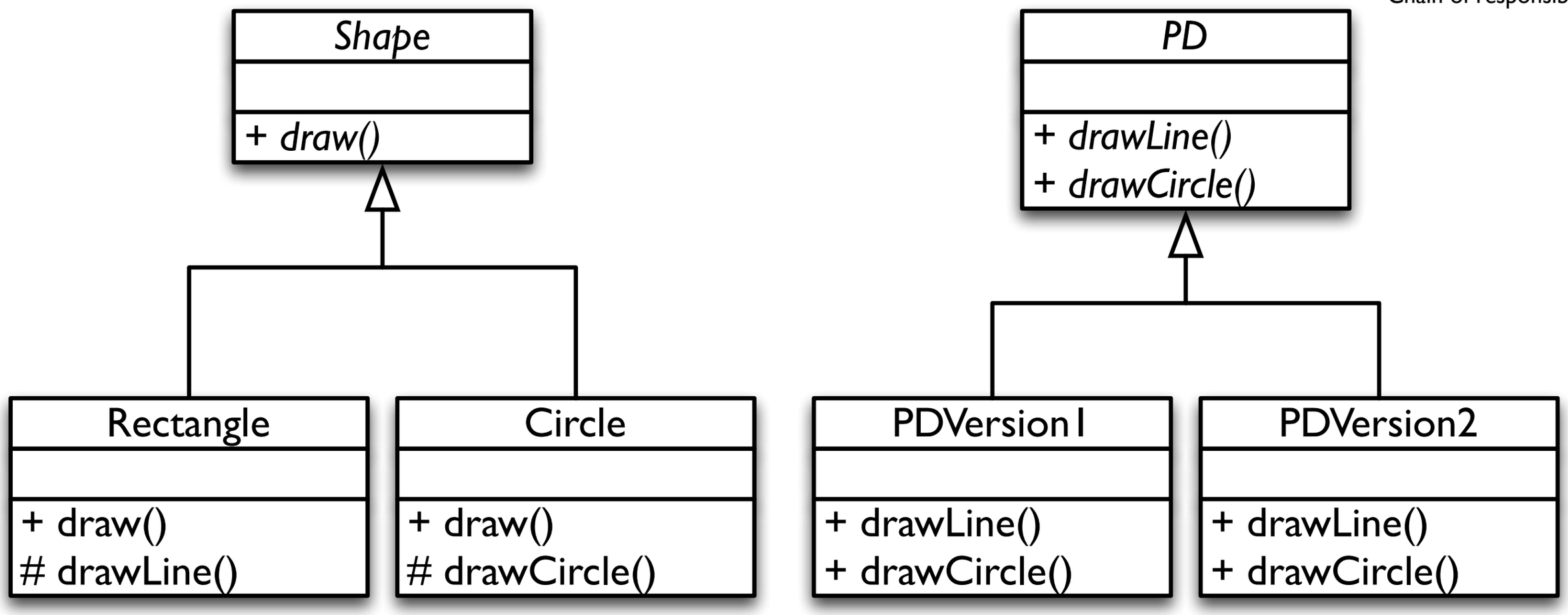


- Well, we really do not like it

- Let's separate the abstraction from the implementation
  - Abstraction: `Shape` (2 types)
  - Implementation: `PDs` (2 types of graphical programs)

# Let's separate abstraction and implementation



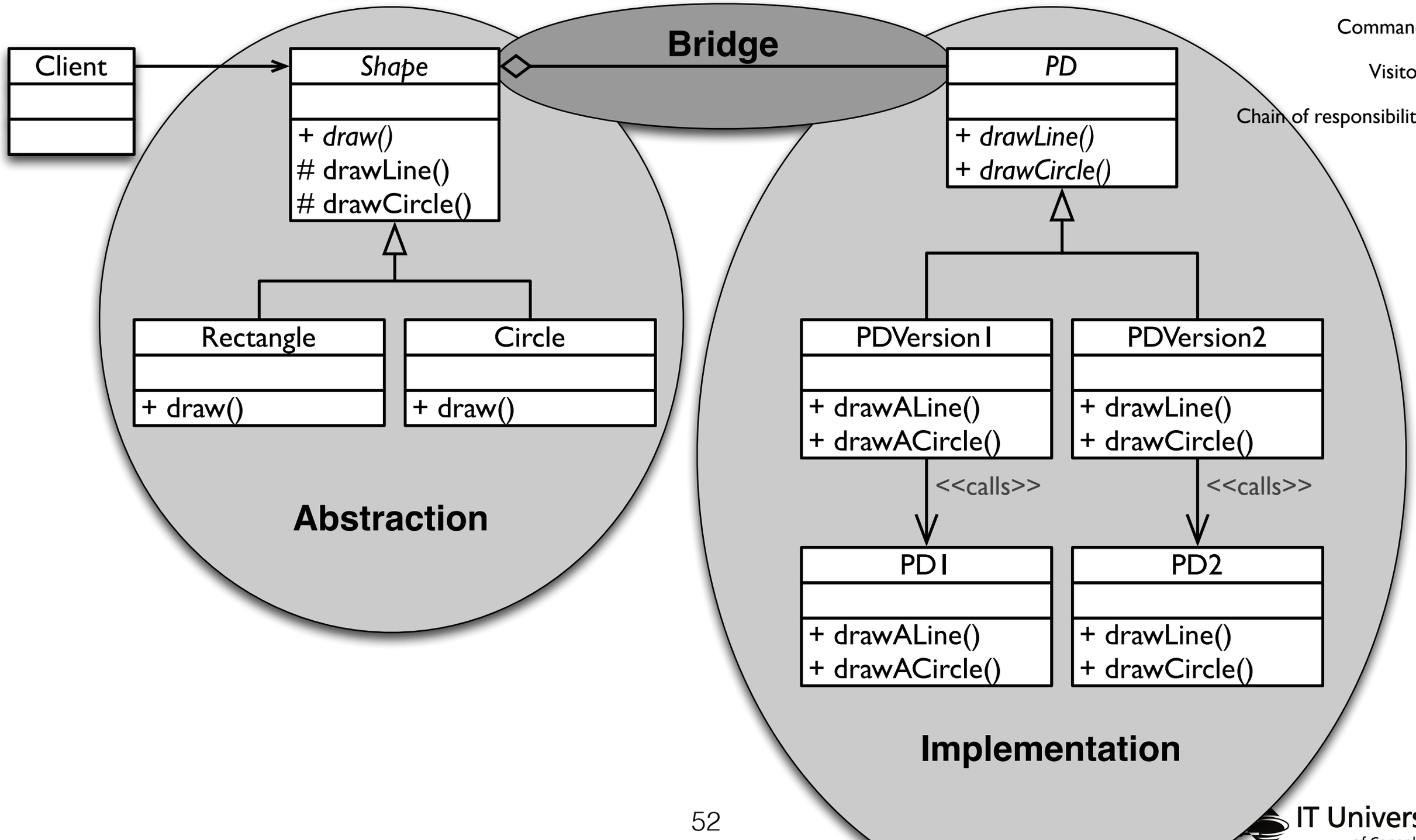


# How do we connect them?



- With a "bridge" …
- A `Shape` delegated the responsibility of drawing (`draw`) to a `PD`
- And given that we are already modifying this, a detail:
  - let's refactor `#drawLine` and `#drawCircle` to the superclass to avoid duplications
  - they are protected, therefore:
    - they are inherited
    - but they are not visible in the `Shape` interface

# Final solution



# Comments



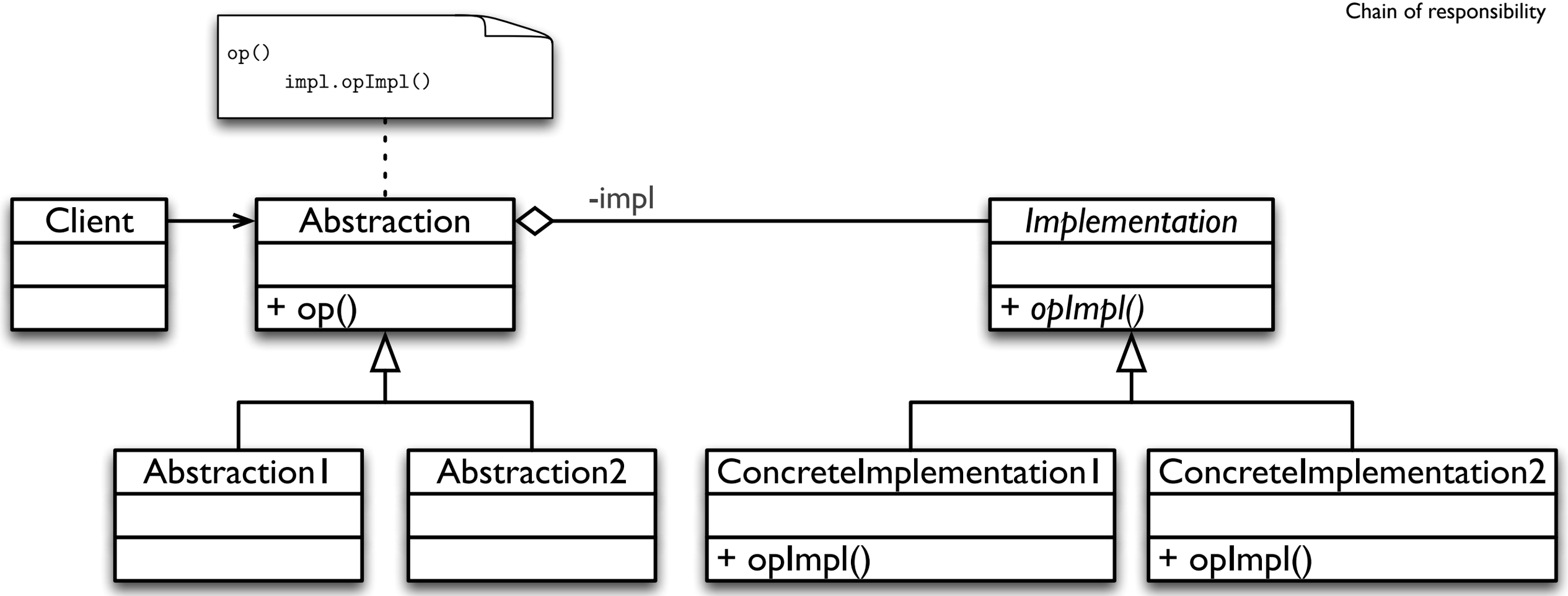
- We have
  - eliminated some inheritance
  - added aggregation/delegation

- There is no more risk of combinatorial explosion
  - N+M classes, not N*M

- How did we do it?
  - Purpose of the bridge (finally clear):
    - Separate an abstraction from its implementation
    - so that they can vary independently

# Bridge diagram





# Comments



- Often interface and implementation are in the same class or they become separated in superclass and subclass
- Bridge separates them more
- Avoids re-compilation
- More in general:
  - if you start having a hint that there is an area in your design that is prone to changes, do not accept the first solution …
    - … however, avoid "paralysis by analysis"

# Bridge versus adapter



- Different purposes
- Adapter adapts, while bridge separates the interface and the implementation to make them vary independently
- Bridge is something that is something that could be used in the early stages of a project
- While adapter is generally used once the classes involved have already been designed/implemented e they get reused
- However, the classes `PDVersion1` and `PDVersion2` are two object adapters

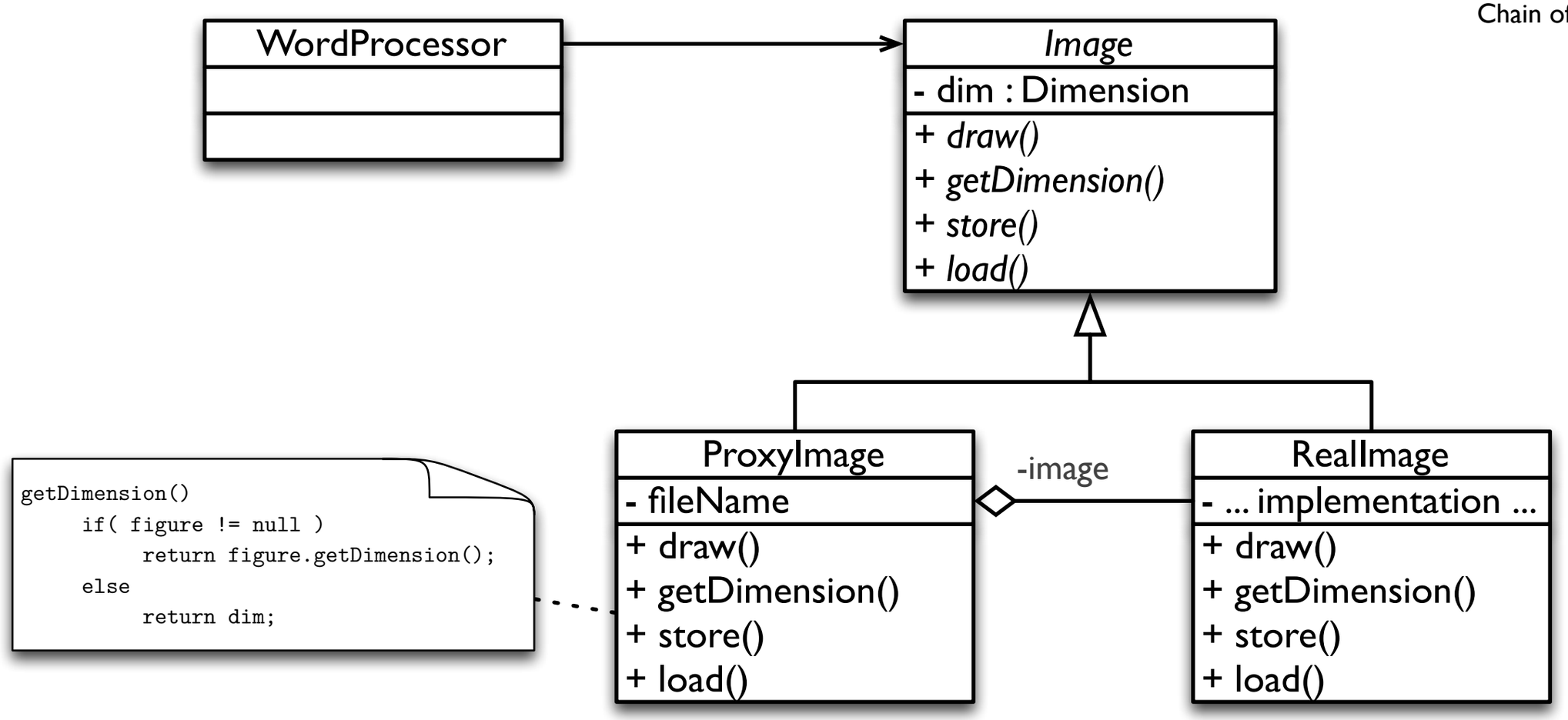
# Proxy



- Purpose
  - Provide a surrogate or a placeholder for an object …
  - … to control its access
- Examples
  - For performance: in a word processor, load the images only when needed avoiding to load them all at the beginning
  - For security: placeholder that checks and manages access control rights
  - …

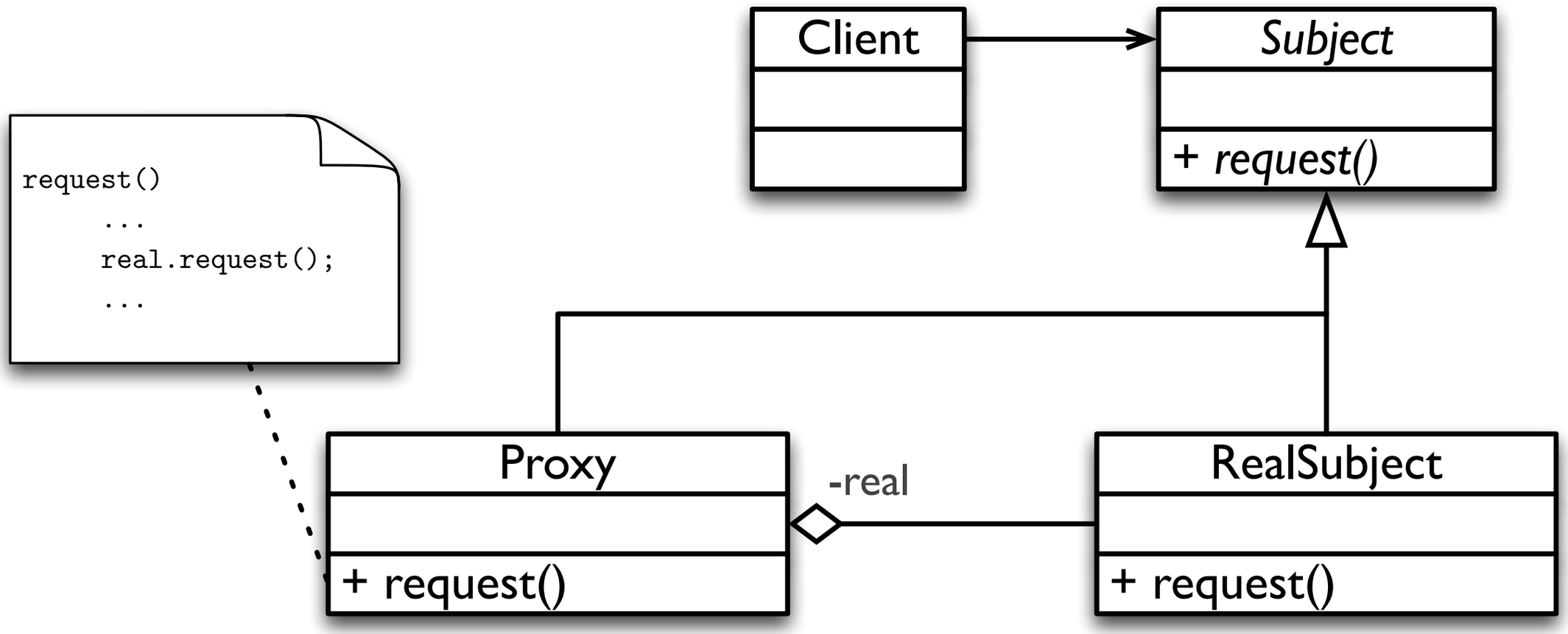
# Placeholder for a word processor





# Proxy diagram





# Comments



- It is a simple patters
- It is another type of wrapper

# Command

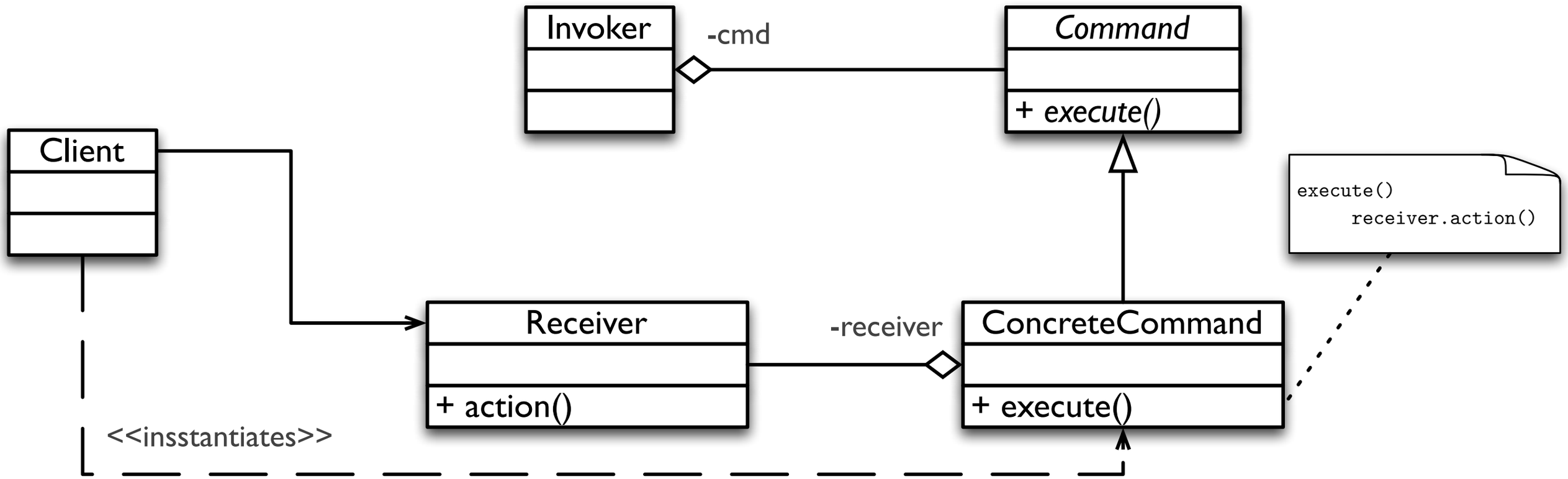


- Purpose
  - Encapsulate a request in an object …
  - … allowing to: configure clients with different requests, maintain a history of the requests, manage the undo

- If a request, a command, is an object,
  - It has its own "life": it can be memorized, passed as an argument, …

# Command diagram





# Mechanism



- An instance of `Client` creates an instance of `ConcreteCommand` passing the `Receiver`
  - The instance of `Command` is a command; at this point, the command is an object …
- Afterwards, an instance of `Invoker` (which keeps track of the commands instantiated so far) through polymorphism invokes `execute()` on one `ConcreteCommand`…
- … finally the instance of `ConcreteCommand` invokes `action` on the `Receiver`.

# Comments



- `Command` de-couples `Invoker` (the one invoking the request) from the `Receiver` (the one executing the request)
- Commands are objects. Therefore:
  - it is possible to compose commands in a single command (Composite)
  - it is easy to add new commands
  - it is possible to keep track of the executed commands to for instance allow undo

# Visitor



- Purpose
  - Allow the definition of a new operation for a hierarchy of classes without modifying the classes themselves

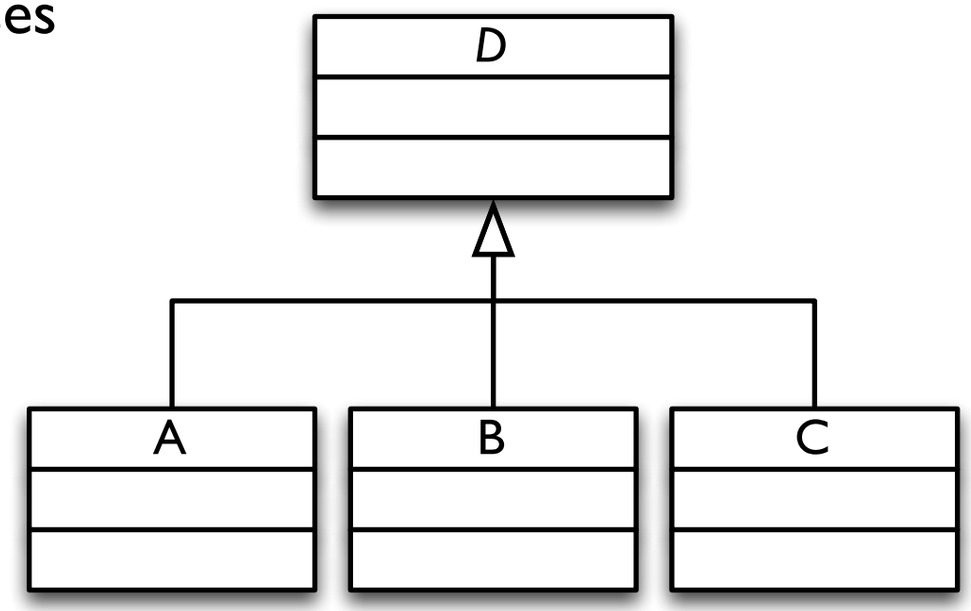
- Based on the visitor metaphor:
  - The visitor asks to be accepted
  - The owners accept and invoke a given method

# Adding behaviours to a hierarchy



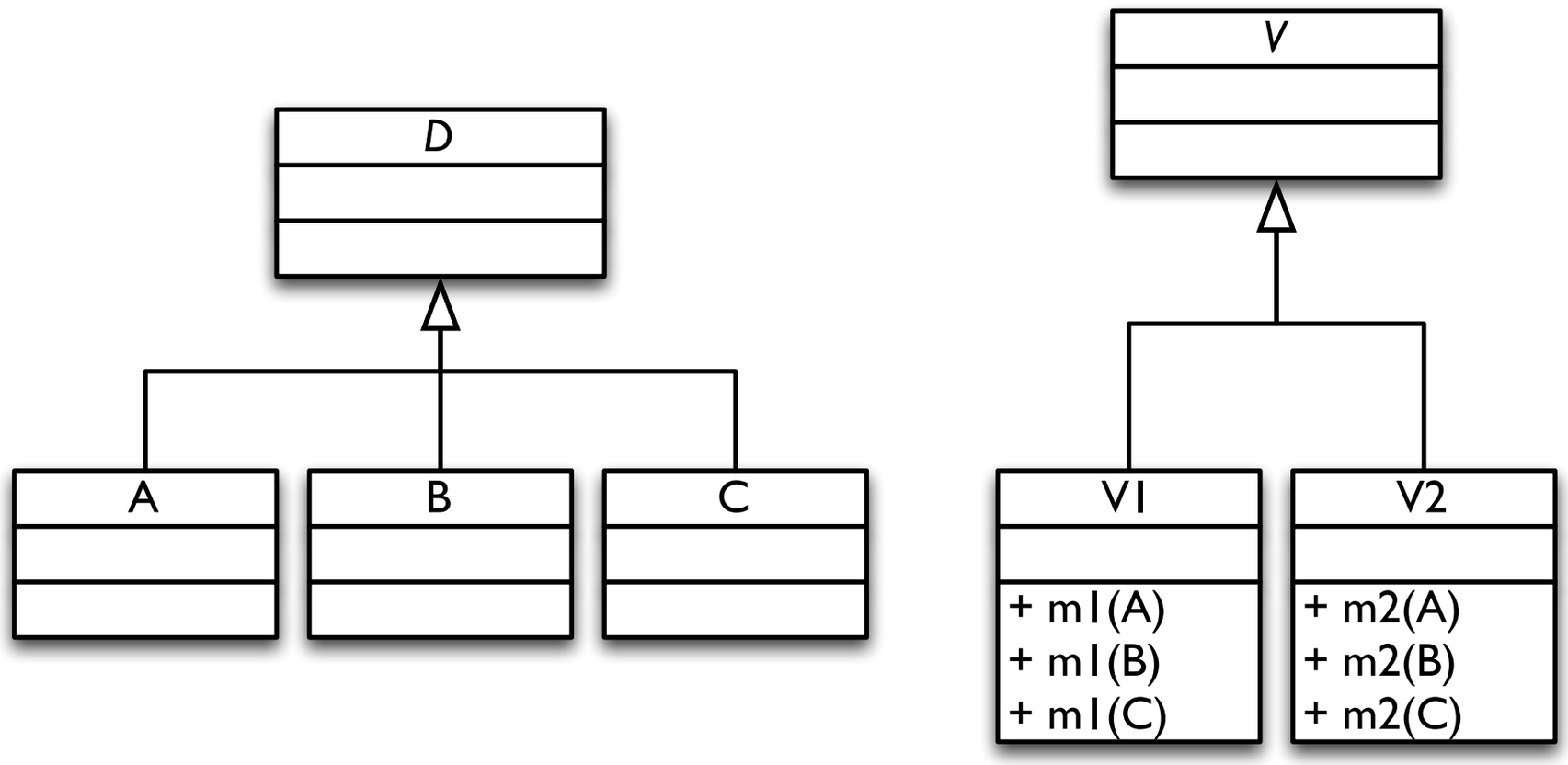
- In a hierarchy it is
  - easy to add classes but
  - complicated to add behaviour to all classes

- There is an alternative …



# … create a parallel hierarchy

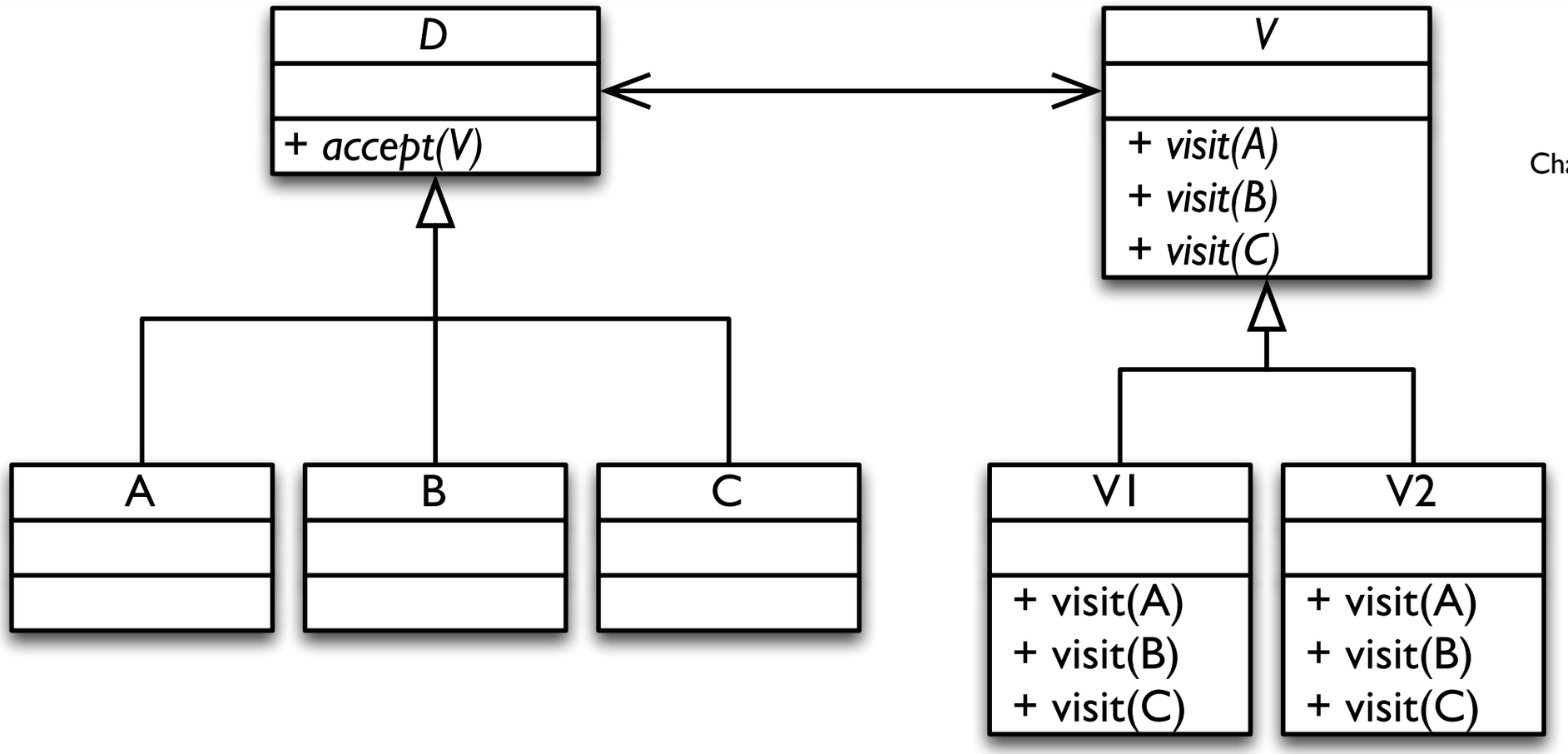





- A subclass for each operation that needs to be added
- Each subclass containing the implementation of the operation for each of the subclasses of the original hierarchy, which are passed as arguments

# Visitor



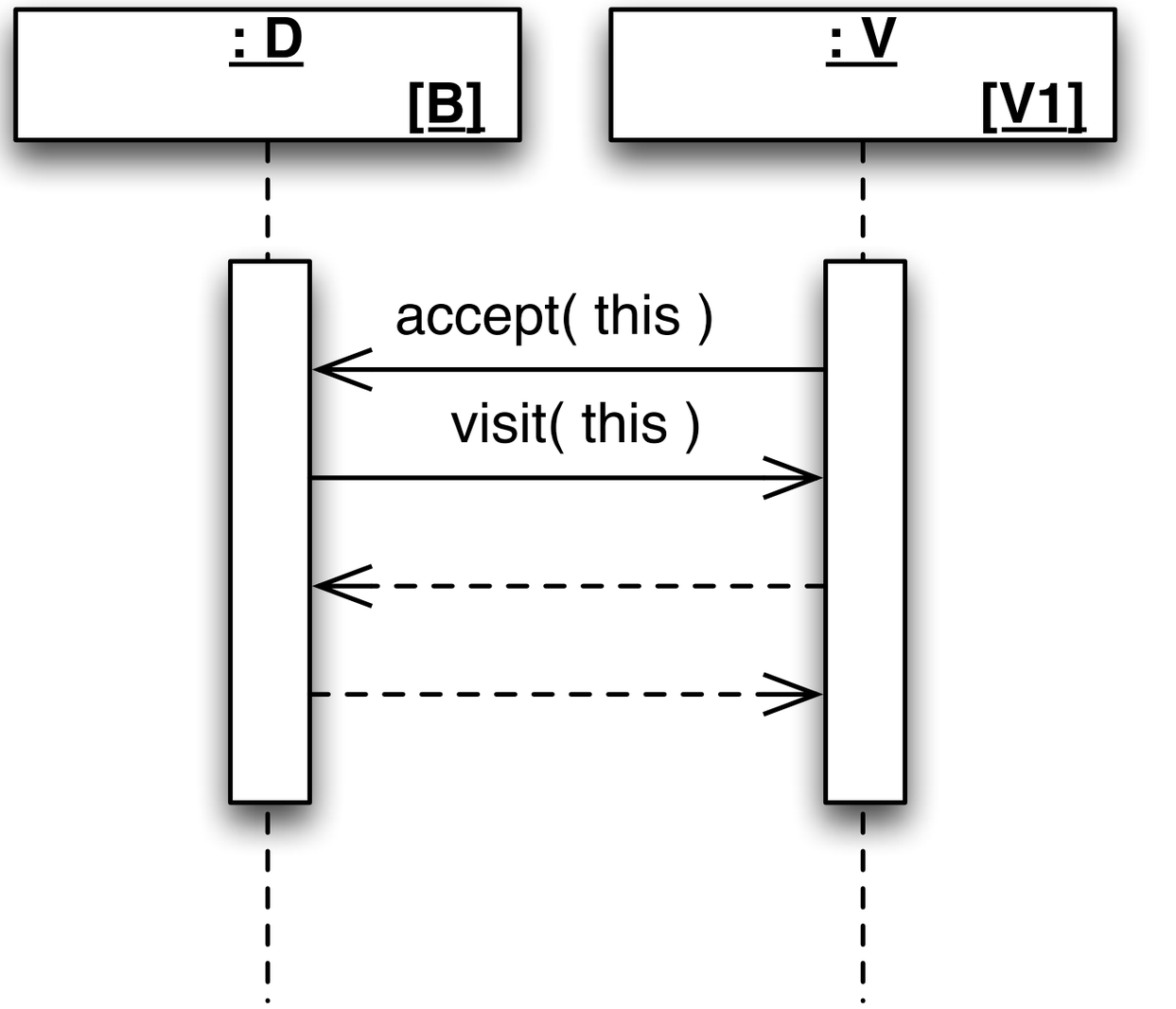



- A final step based on the visitor metaphor
  - the instance of the right hierarchy is the visitor
  - the instance of the left hierarchy is the owner
  - the visitor asks to be hosted/accepter
  - the owner accepts (by invoking the correct method)

# Double dispatching



- `V` invokes `accept()` on `D`
- `D` executes accept by invoking visit on `V`
- `this` is passed as a parameter to support the polymorphic selection on the hierarchy
  - `D` (or a subclass) could be rejected
  - The method `visit()` of `V` (or a subclass) is selected based on the parameter `this`



# Comments



- Controversial pattern
- To extend the behaviour of a hierarchy there is a need of an in depth knowledge of the hierarchy
- High coupling between the visitor and the visited hierarchy
- The alternatives are usually stronger

# Chain of responsibility

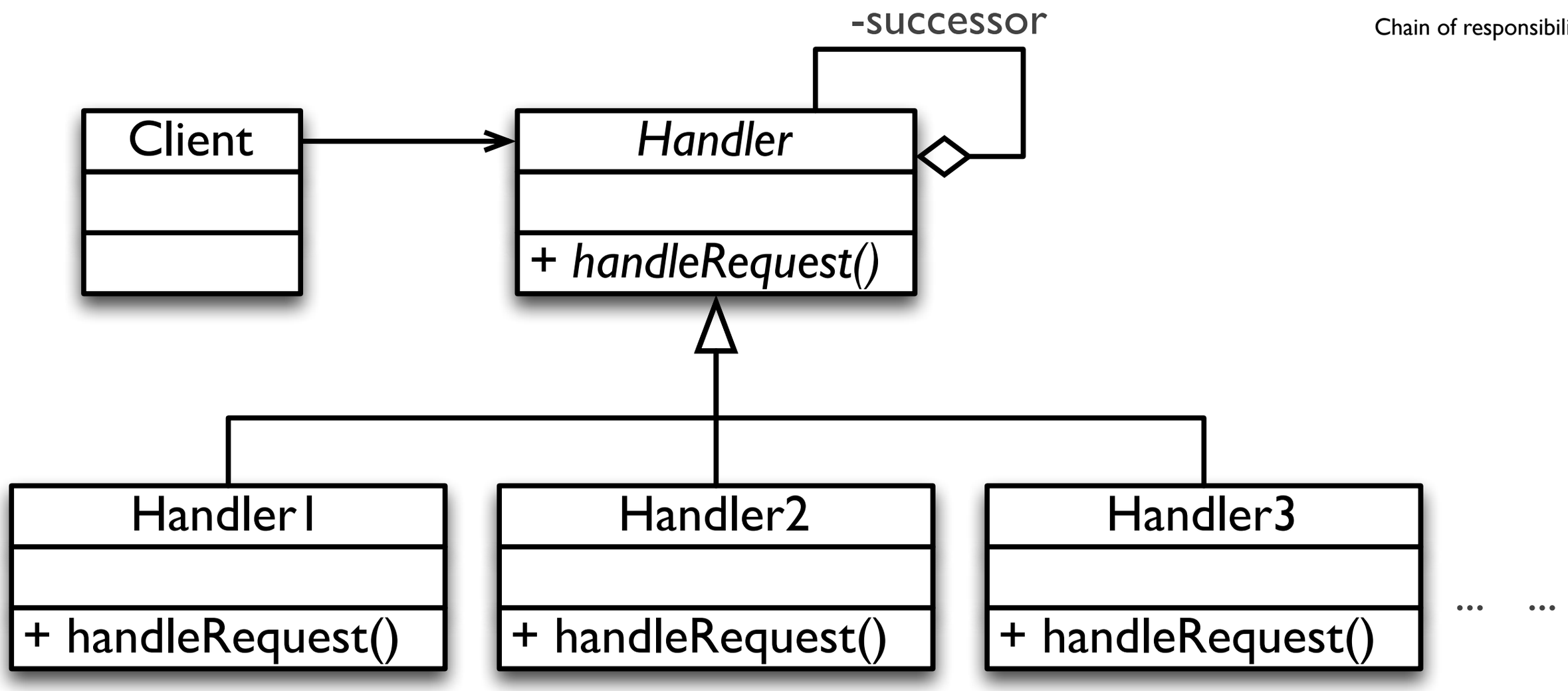


- Purpose
  - Avoid the coupling of the sender of a request with the receiver

- More objects can satisfy the request
  - the objects are chained
  - and they hop the request along the chain
  - until one of them can satisfy it

# Chain of responsibility diagram





# Comments



- The common interface
  - As usual the client "talks" to the (abstract) base class
  - The client does not know who will handle the request
  - The common interface for the handling and/or hopping of the request (`+handleRequest()` and `-successor`) ensures that the object satisfying the request remains implicit
- The chain
  - If the request arrives at the end of the chain, it means the request cannot be handled
  - It is possible to dynamically modifying the chain (e.g., its order or by adding/removing bits)
  - The chain could already exist, for instance, via a Composite
- The request representation
  - Could a method invocation (even static)
  - Or a parameter with the request code
  - Or the instance of a class `Request` (why not using the Command)

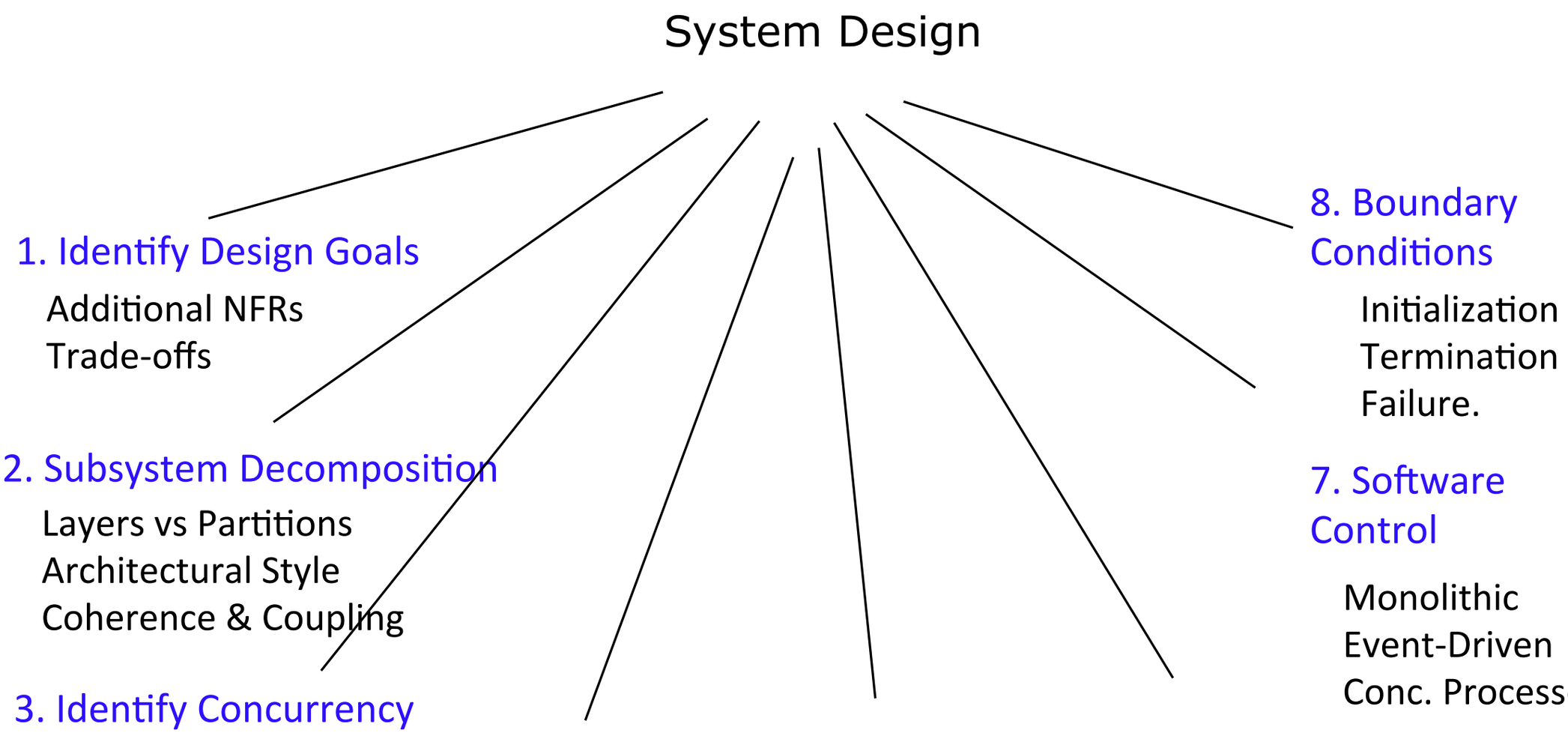
# Example



- Contextual help function in a GUI: the "?" button
- `Button` is contained in a `Panel`, which is contained in another `Panel`, which is contained in a `Frame`
- If a user clicks on the "?" button,
  - the request goes to the `Button`
  - then to `Panel1`
  - then `Panel2`
  - and finally the `Frame`
- The first one that can satisfy the request, does it

# System Design Document (SDD)

# System Design Document (SDD)



# System Design Document (SDD)

**System Design Document**

1. Introduction
   - 1.1 Purpose of the system
   - 1.2 Design goals
   - 1.3 Definitions, acronyms, and abbreviations
   - 1.4 References
   - 1.5 Overview
2. Current software architecture
3. Proposed software architecture
   - 3.1 Overview
   - 3.2 Subsystem decomposition
   - 3.3 Hardware/software mapping
   - 3.4 Persistent data management
   - 3.5 Access control and security
   - 3.6 Global software control
   - 3.7 Boundary conditions
4. Subsystem services

Glossary

# Concluding

# Outline

- Literature
  - [OOSE] ch. 7-8

- Topics covered:
  - Design Patterns (II)
  - SDD