Software Architectures

From Design Patterns to Enterprise Architecture

Design Patterns II

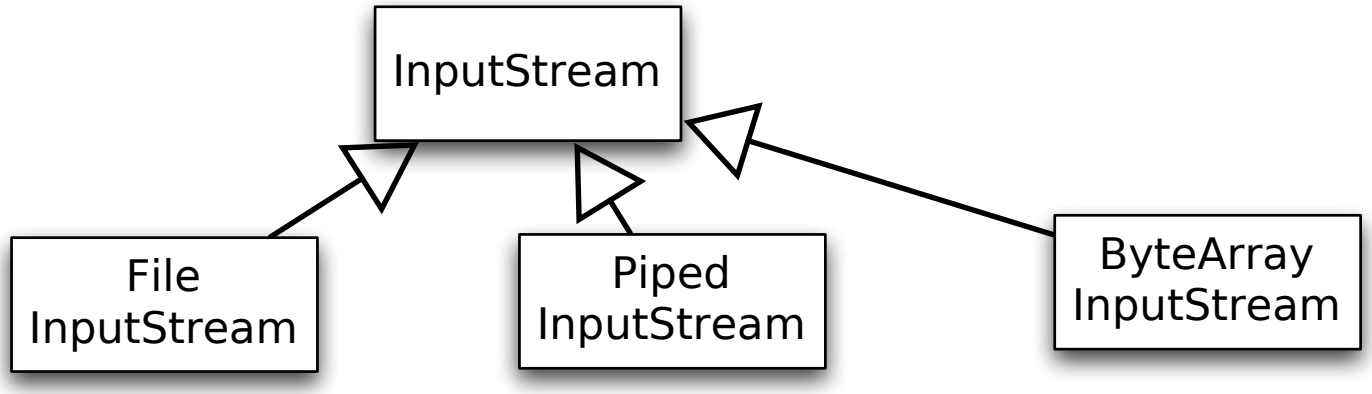
Last week on “Software Architectures”

- What is a Design Pattern?
- Why are we using Design Patterns?
- Template Method Pattern
- Model-View-Controller
- Observer Pattern

Patterns

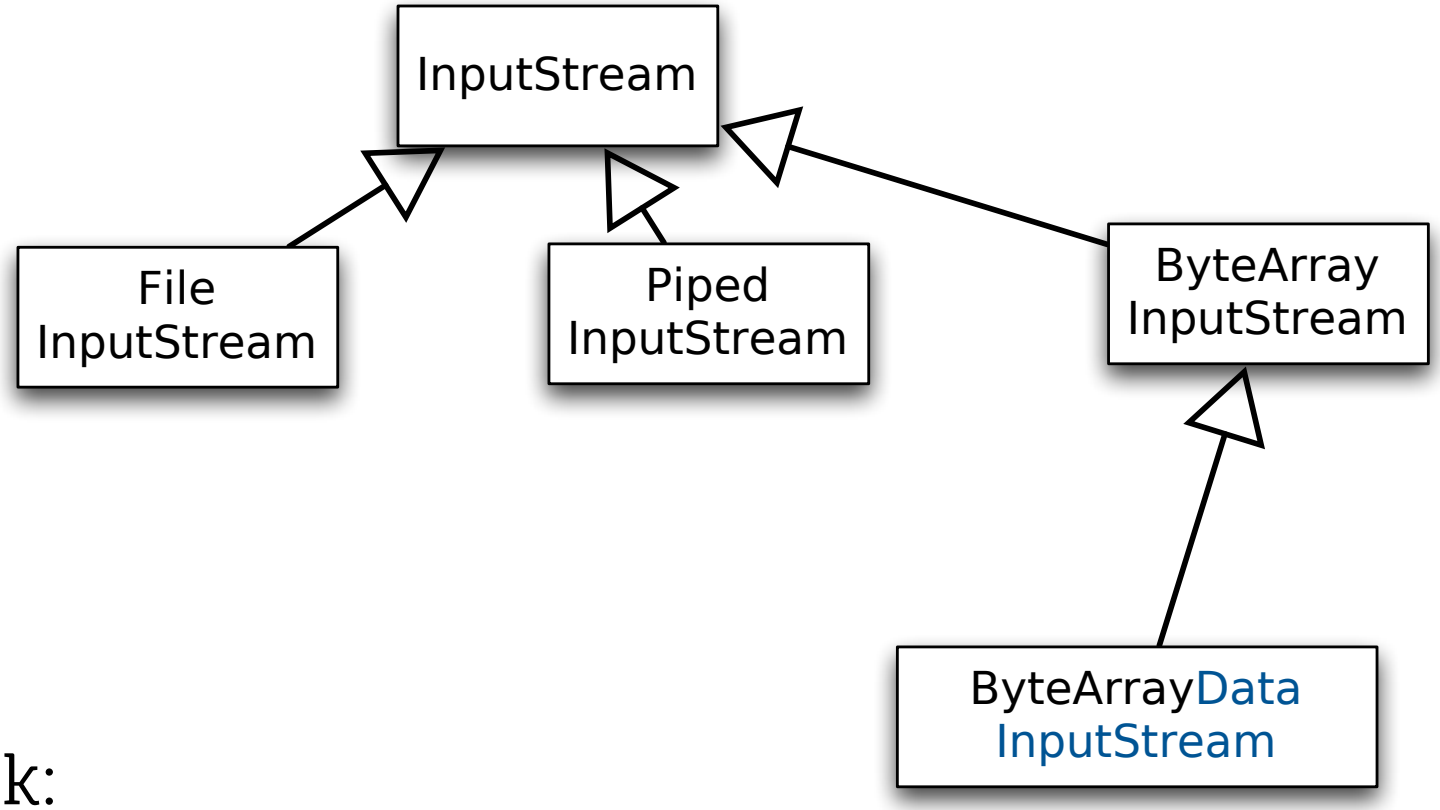
Decorator (175)

Decorator
Motivation



- Task:
  - Adding functionality to a ByteArrayInputStream to read whole sentences and not just single bytes.

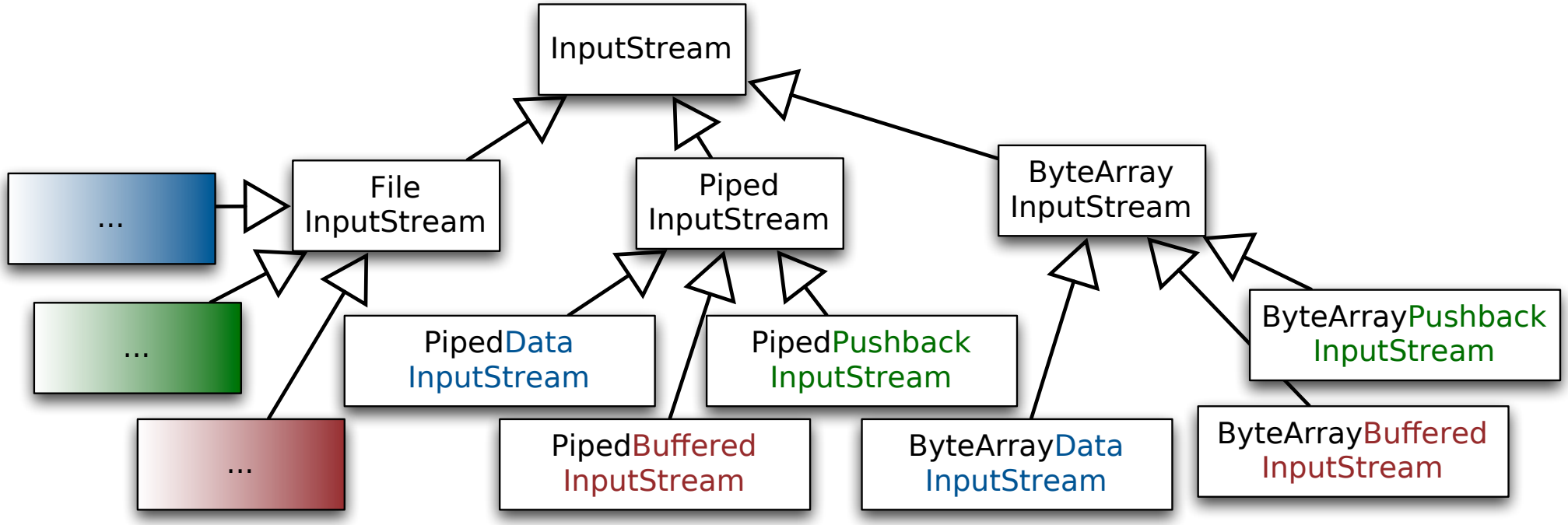
Decorator
Motivation



- New Task:
  - We also want to have the possibility to read whole sentences using FileInputStreams…

Decorator
Motivation

- And after a few iterations of that we get this...

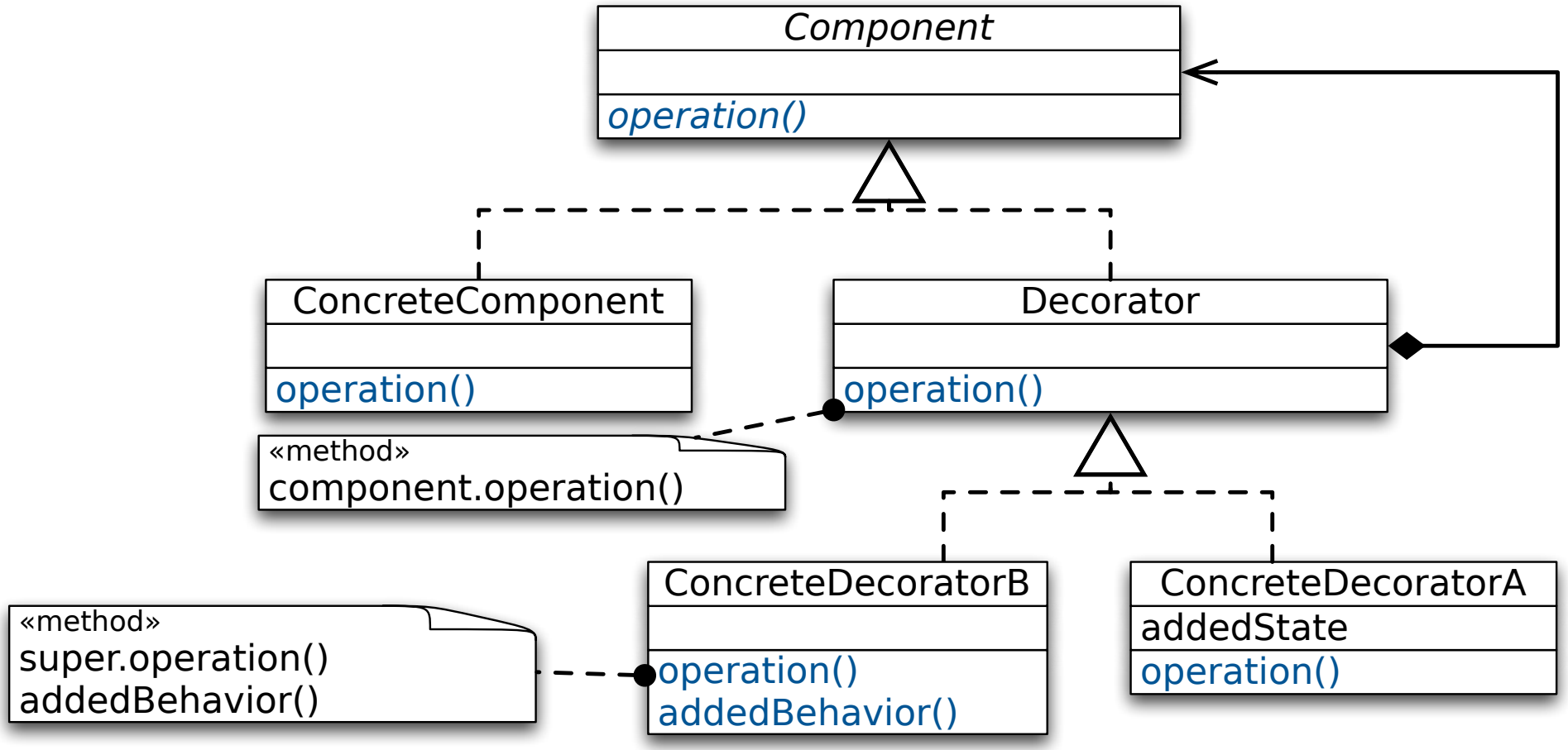


Decorator
Intent

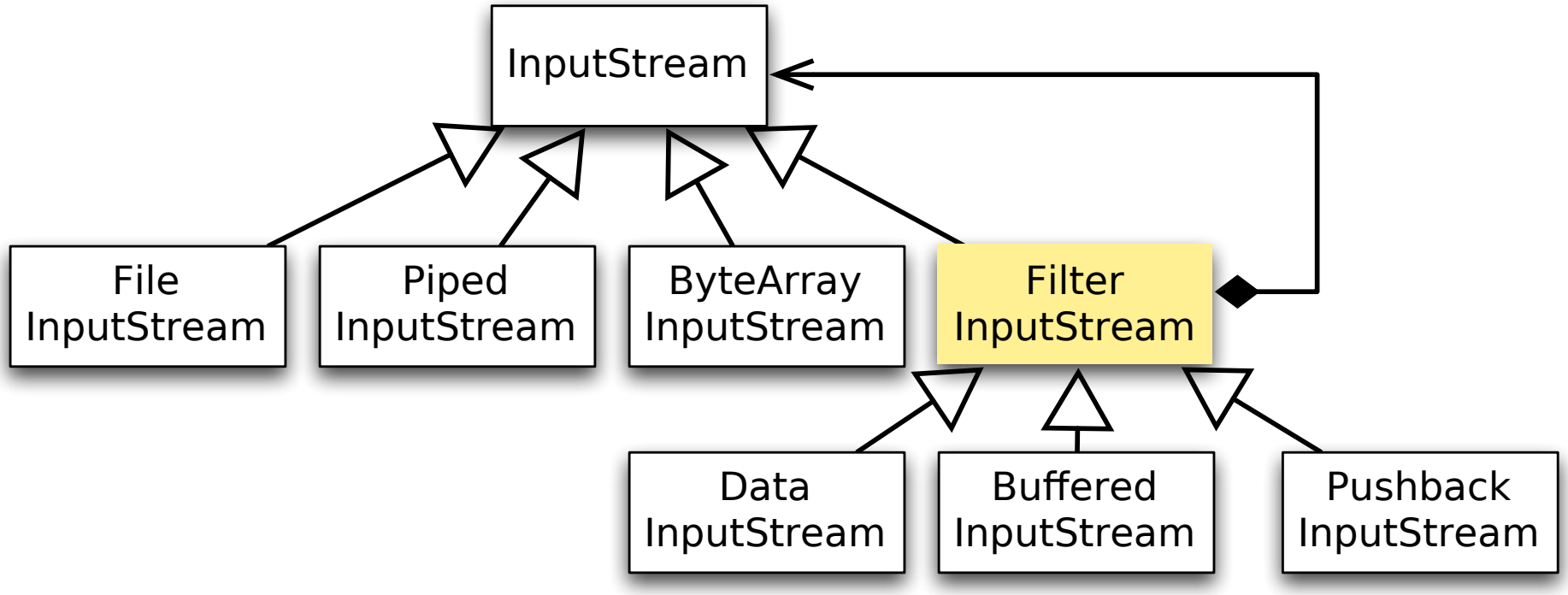
- Attach additional responsibilities to an object dynamically
- Decorators provide a flexible alternative to subclassing for extending functionality
- In other words: We need to add responsibilities to existing objects dynamically and transparently, without affecting other objects.

Decorator

Structure



Decorator
Example




```
new DataInputStream(new FileInputStream("…")).readUnsignedByte()
```

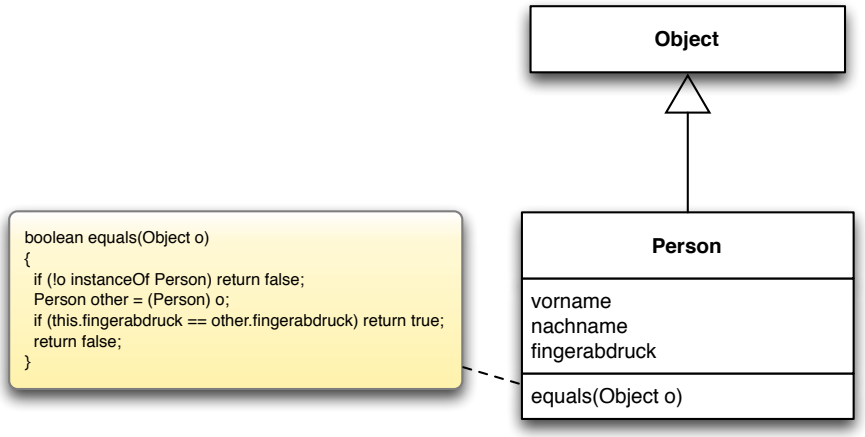
Decorator
Advantages

- Enables more flexibility than inheritance
- Avoids incoherent classes

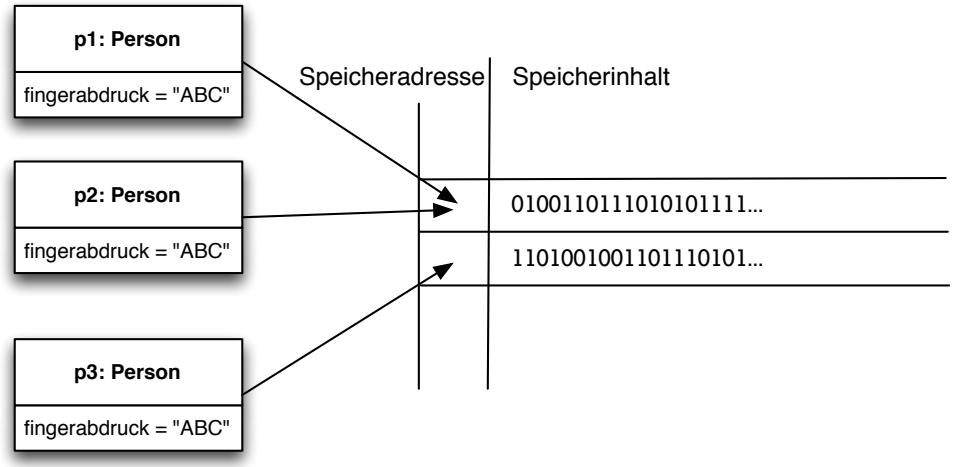
Decorator
Disadvantages

- Lots of little objects
- Object identity

Excursion: Object identity vs. object equality




```
p1 == p3   ---->   false
p1.equals(p3) ----> true
```

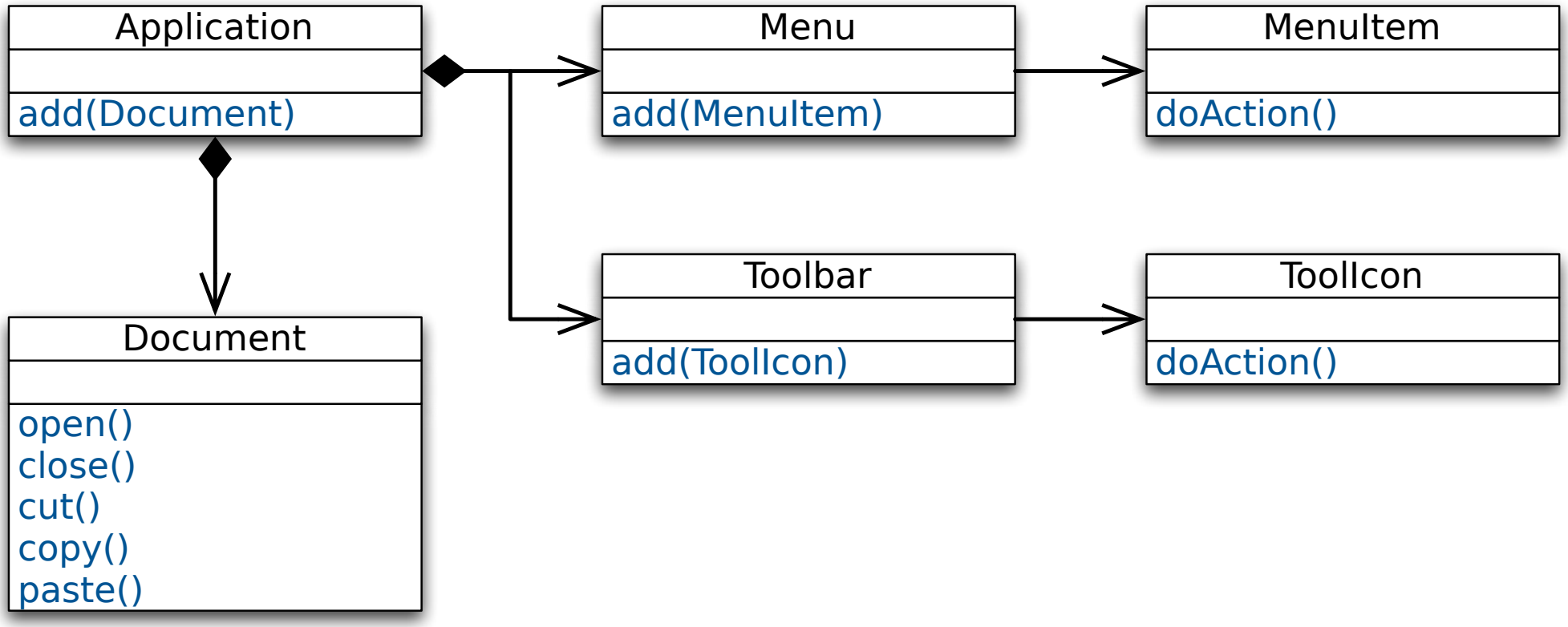


Patterns

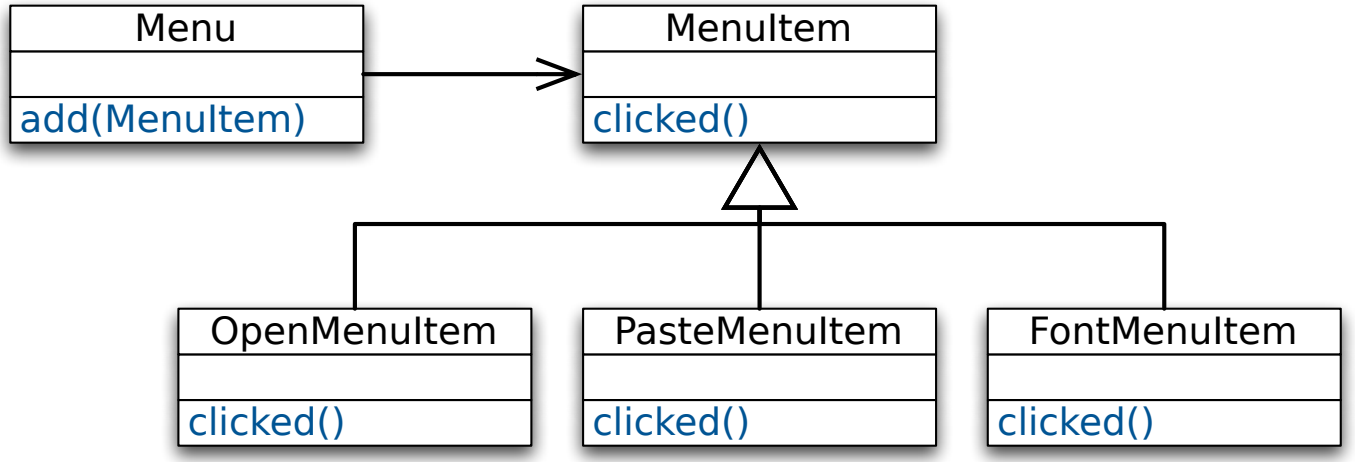
Command (233)

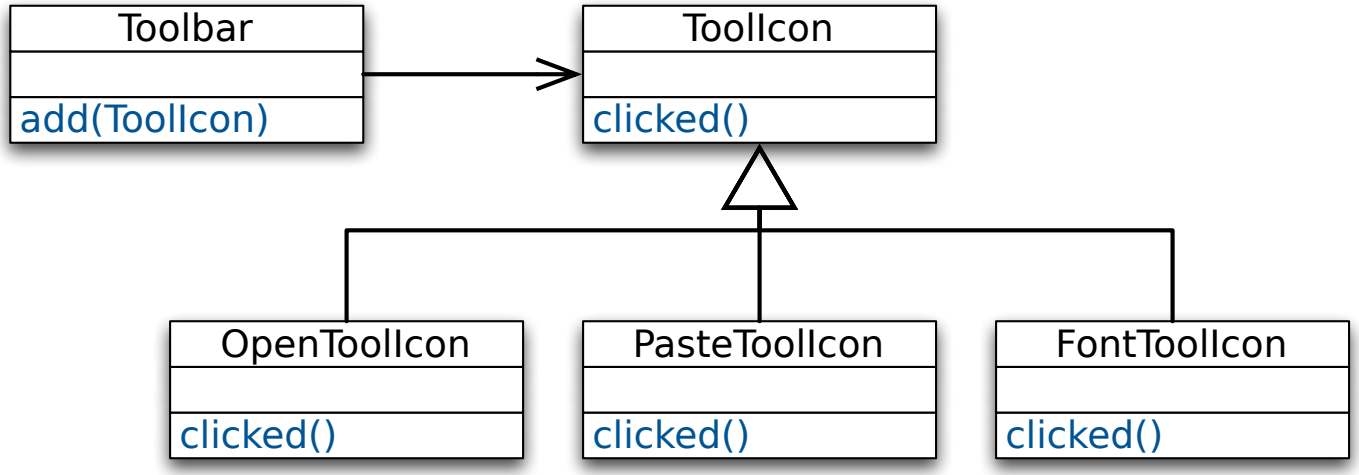
Command

Motivation

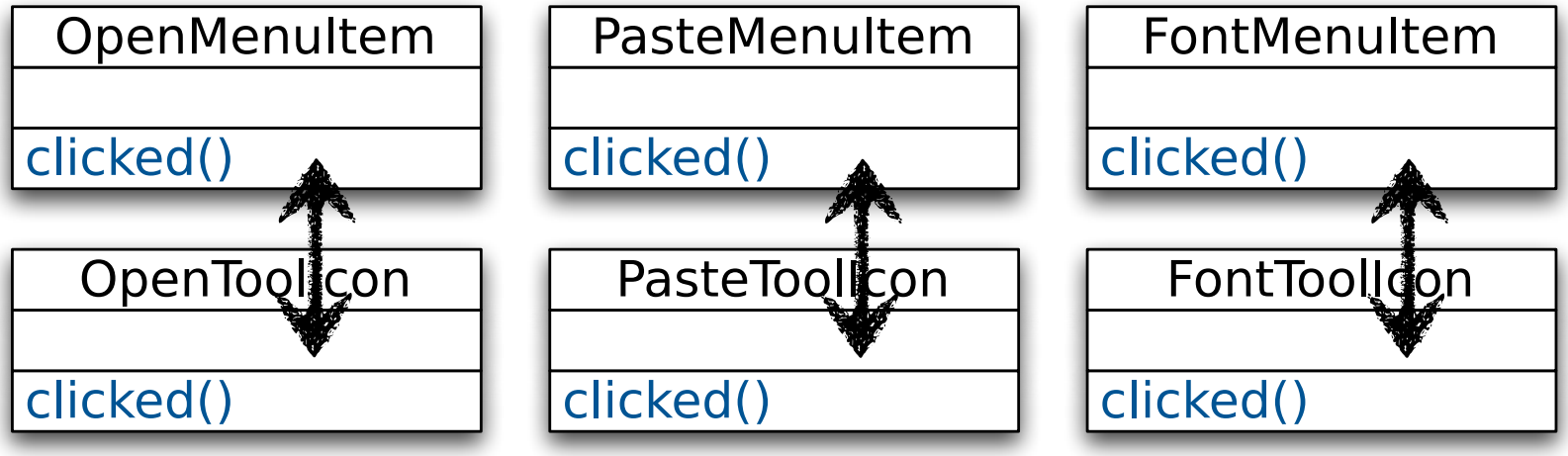


Command
Motivation





Command
Motivation

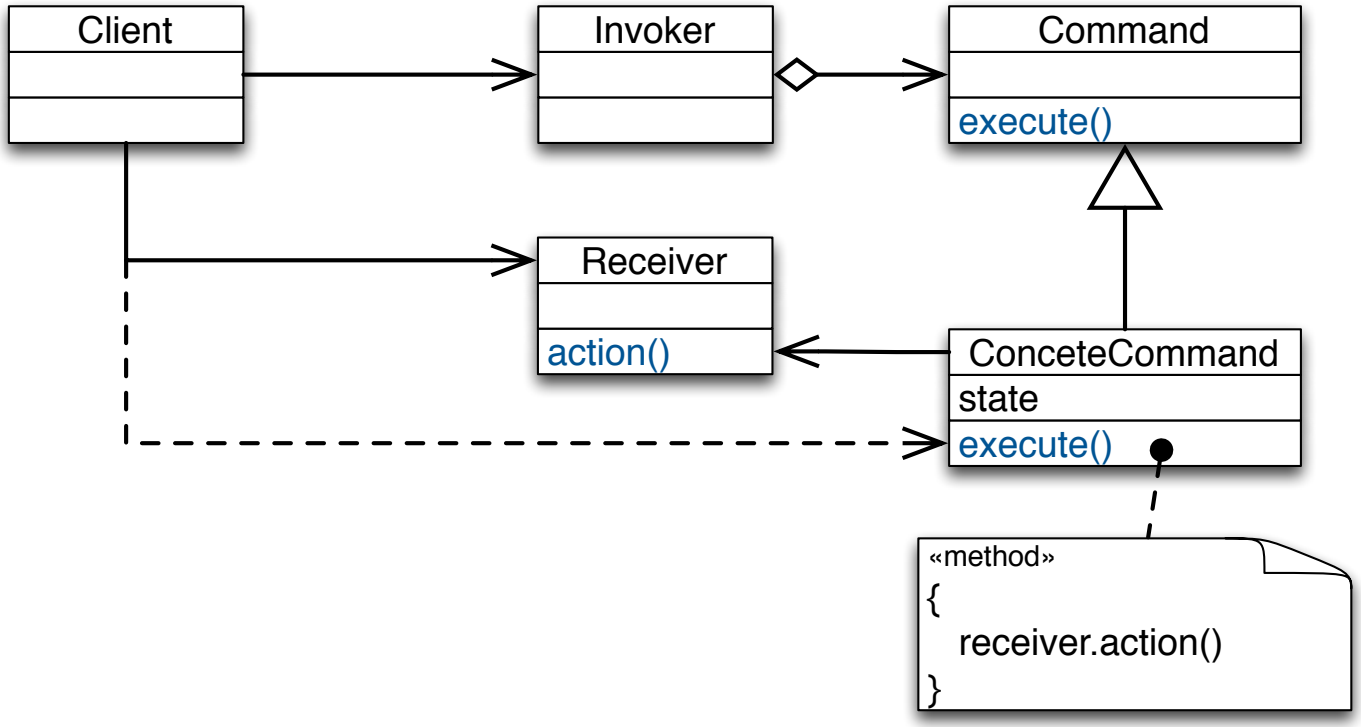


Command
Intent

- Encapsulate a request as an object, thereby letting you parametrize clients with different requests, queue or log requests, and support undoable operations

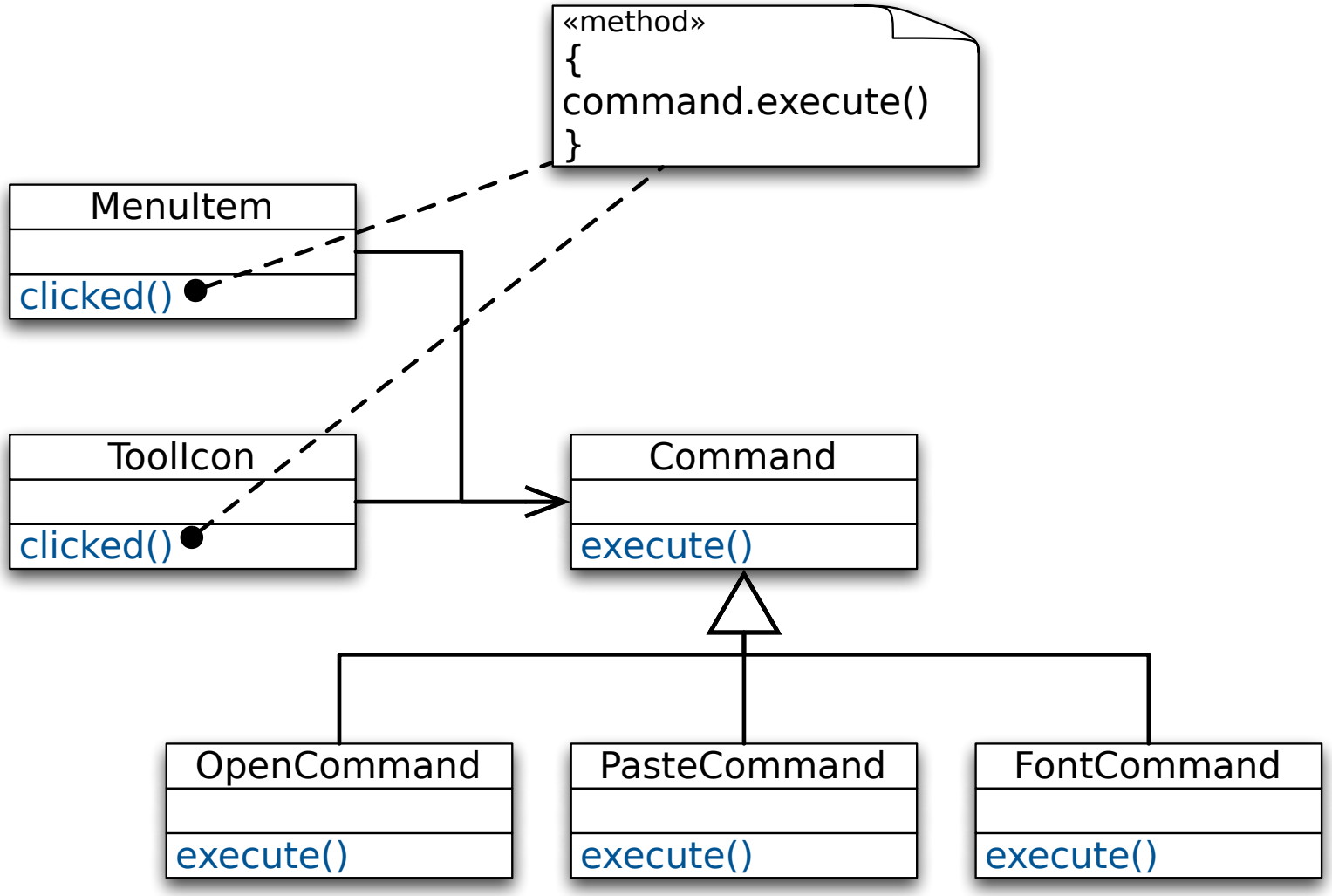
Command

Structure



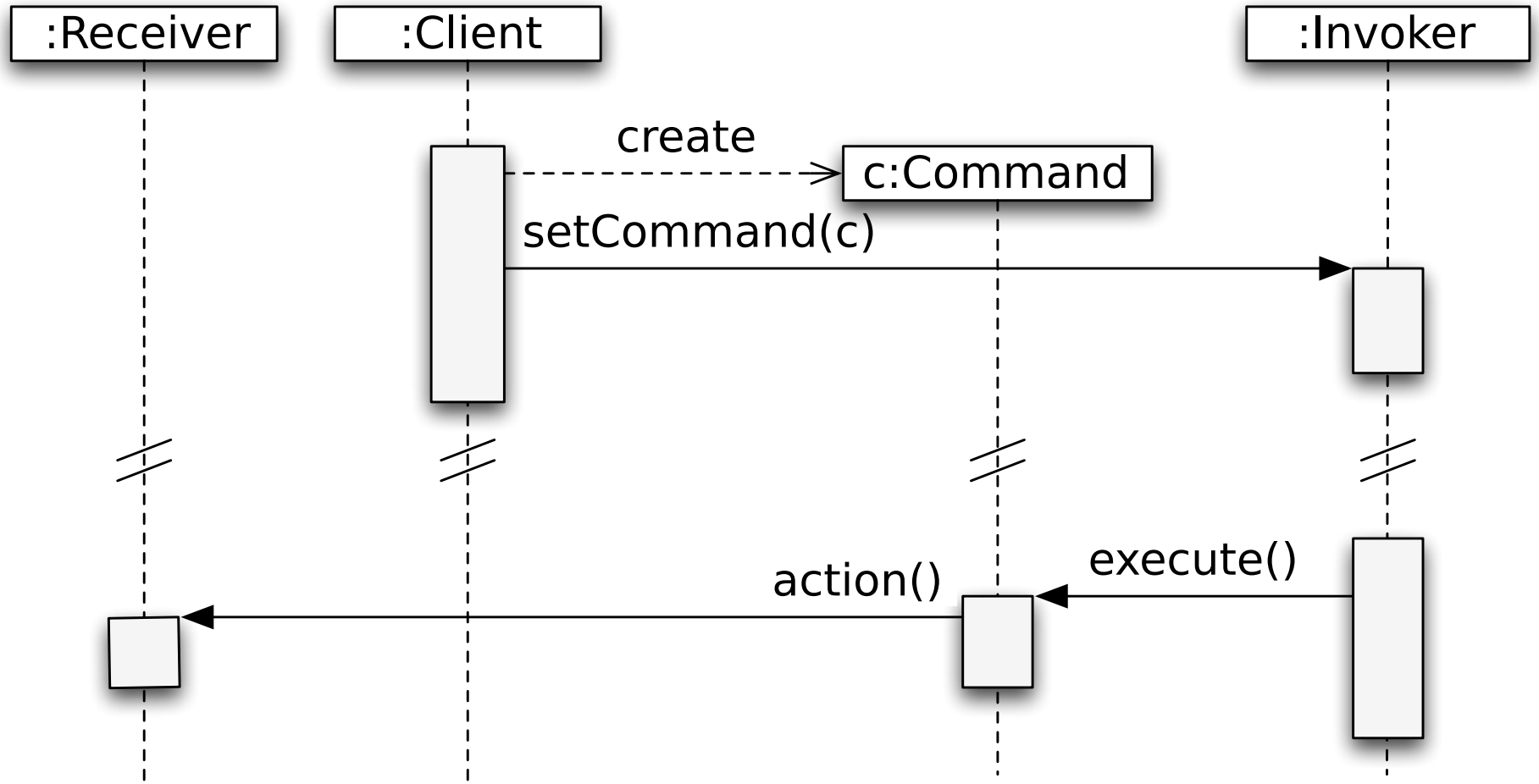
Command

Back to the example



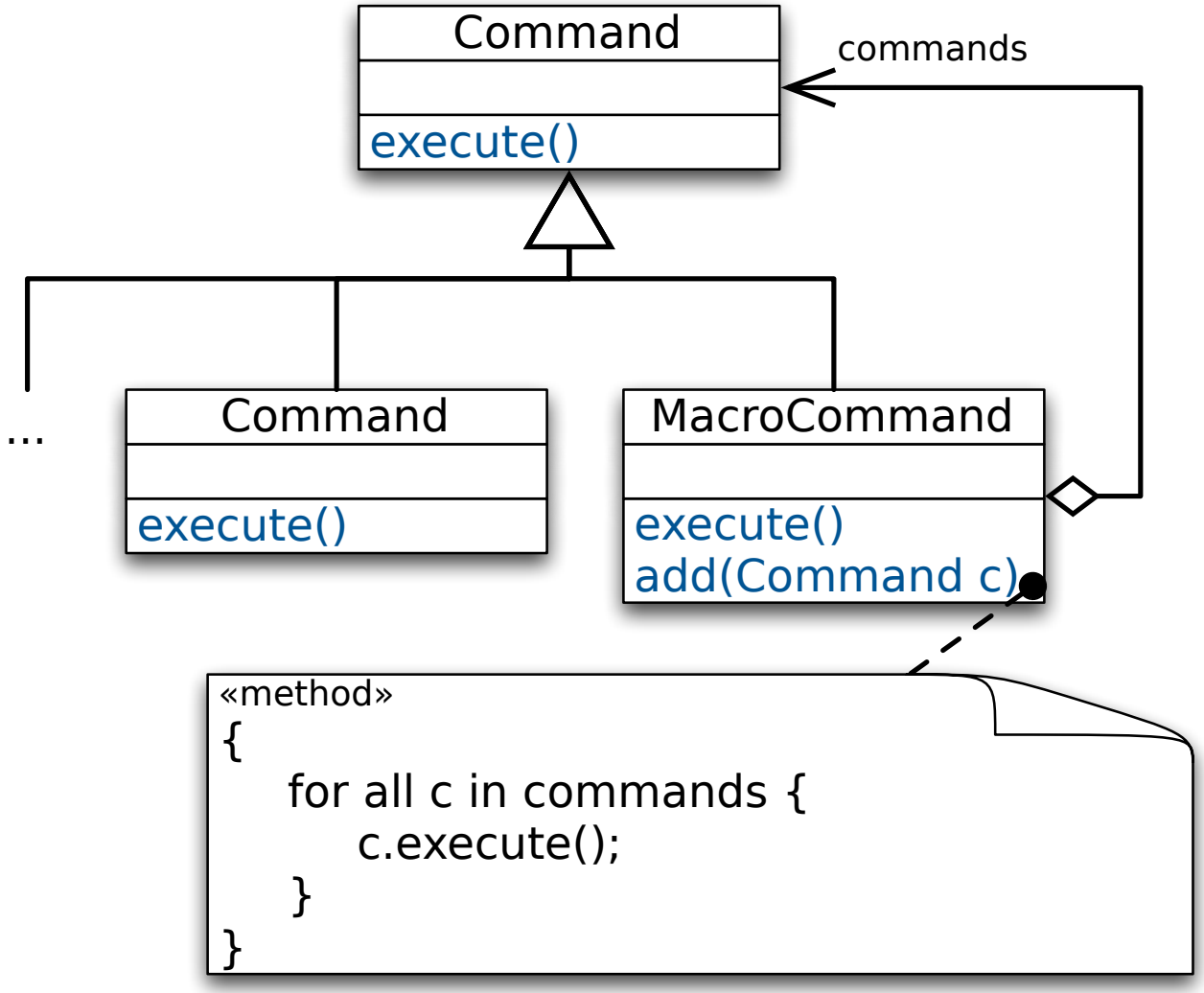
Command

Sequence



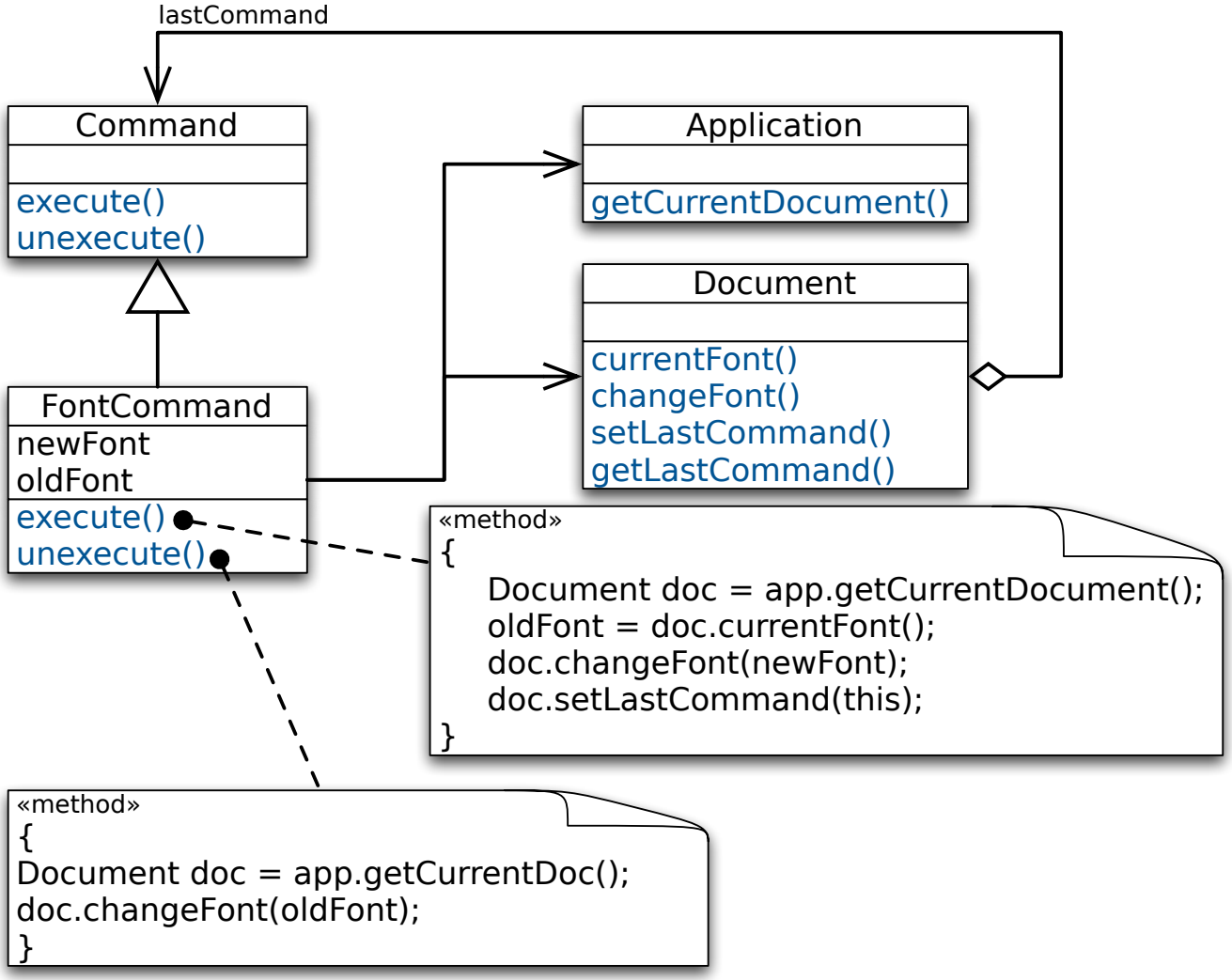
Command

Additional Features: Macros



Command

Additional Features: Undo/Redo



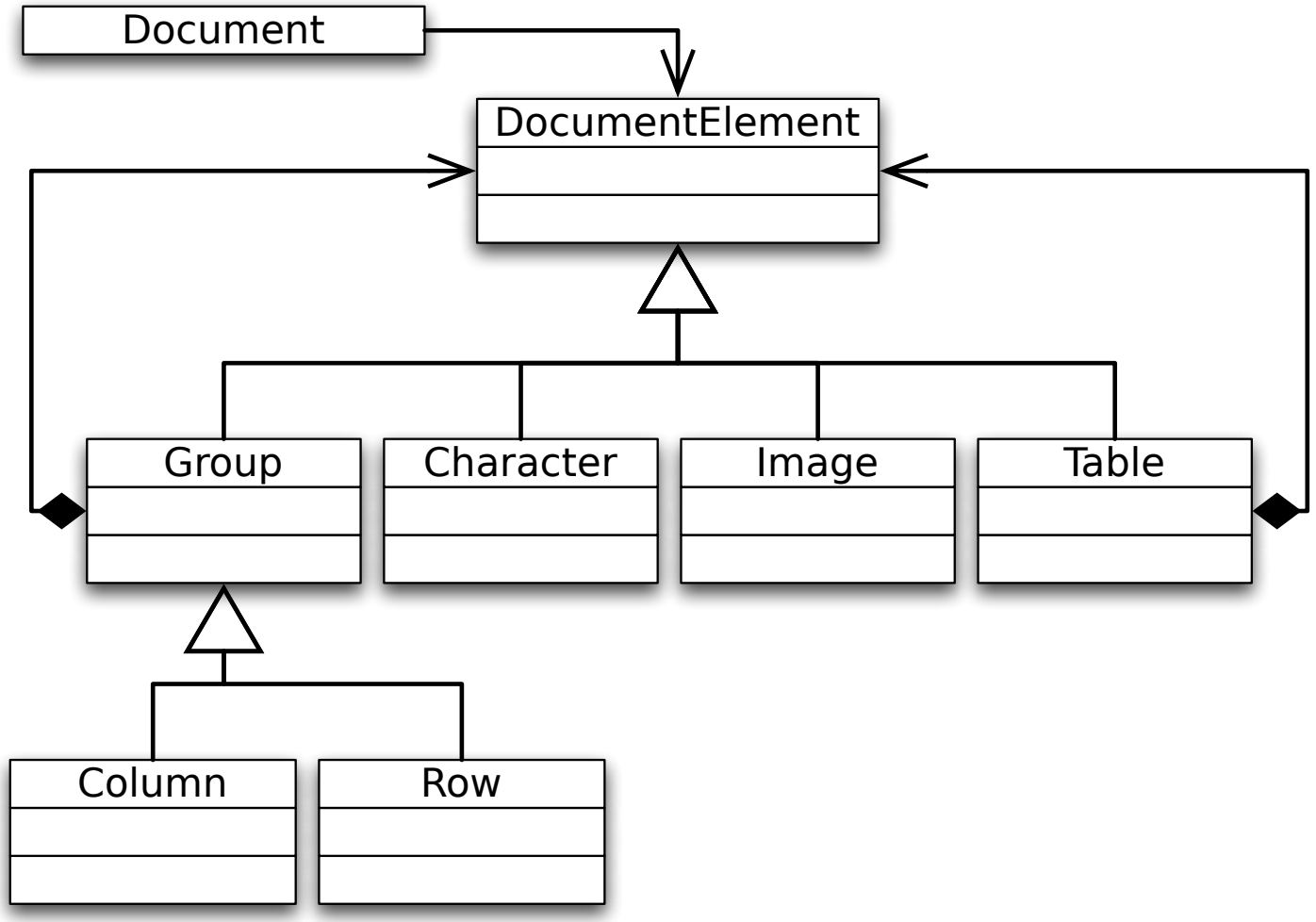
Command
Advantages / Disadvantages

- De-couples the invoker from the performer
- Easy to add new commands
- Commands are first-class objects
- Supports queuing of commands
- Supports logging and replay of commands
- Supports creation of macro commands
- But it adds an additional class

Patterns

Iterator (257)

Iterator
Motivation

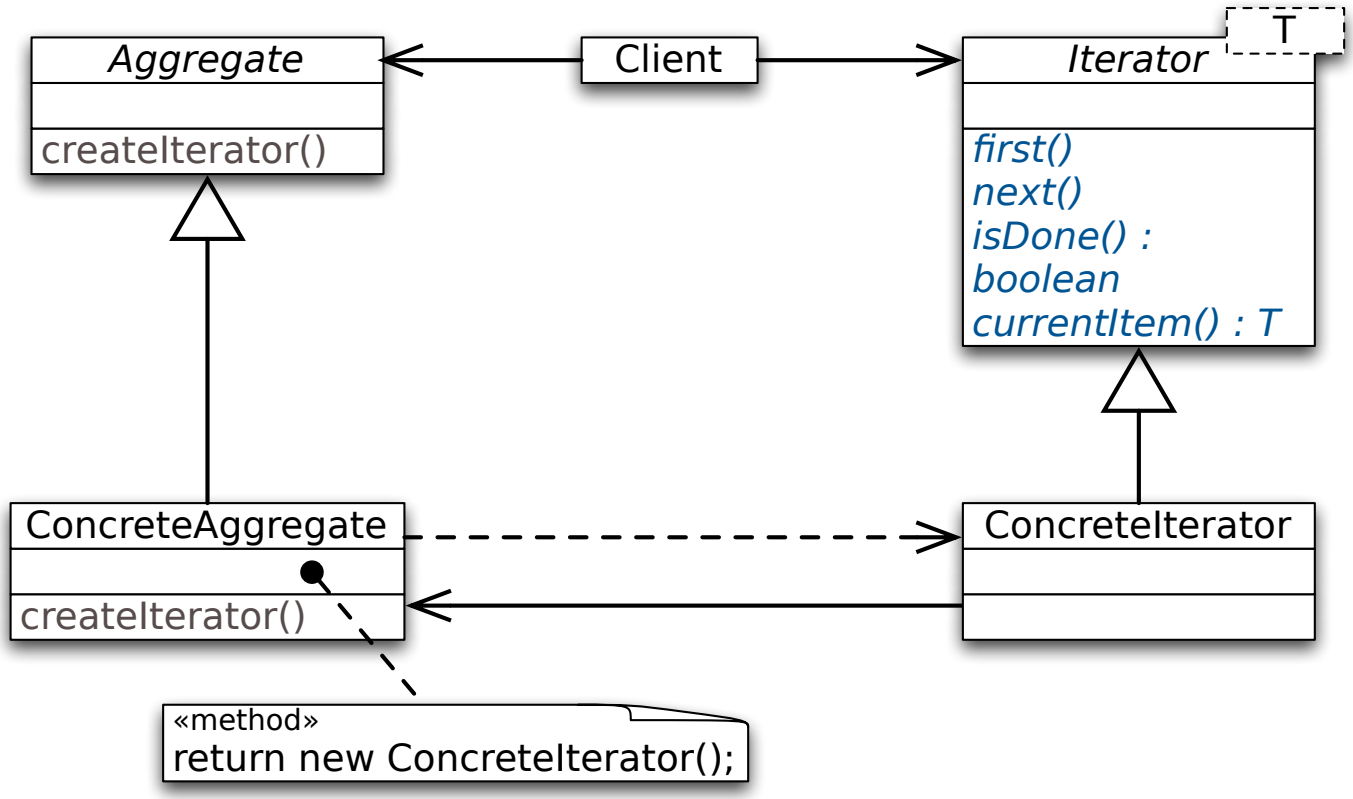


Iterator
Intent

- Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation

Iterator

Structure



Iterator

Using an iterator


```
Iterator i = Aggregate.createIterator();
i.first();
while (!i.isDone()) {
   doSomething(i.currentItem());
   i.next();
}
```

VS.

```
public abstract class DocumentIterator {

   public DocumentIterator(Document doc) {…}
   public boolean traverse() {…}

   public abstract processElement(DocumentElement elem);
}
```

Iterator
Consequences

- Abstract traversal of an aggregation
  Clients don’t know the internal representation of the aggregation
- Iterators simplify the Aggregate interface
  Aggregates don’t have to supply traversal functions other than an iterator creation function
- Supports variation on the traversal strategy
  Concrete iterators can provide different traversal mechanisms (e.g., going backwards though the aggregate)
- More than one traversal can be active on an aggregate
  Each iterator keeps track of its own position in the aggregate

Wrap Up

- Decorator Pattern
- Command Pattern
- Iterator Pattern