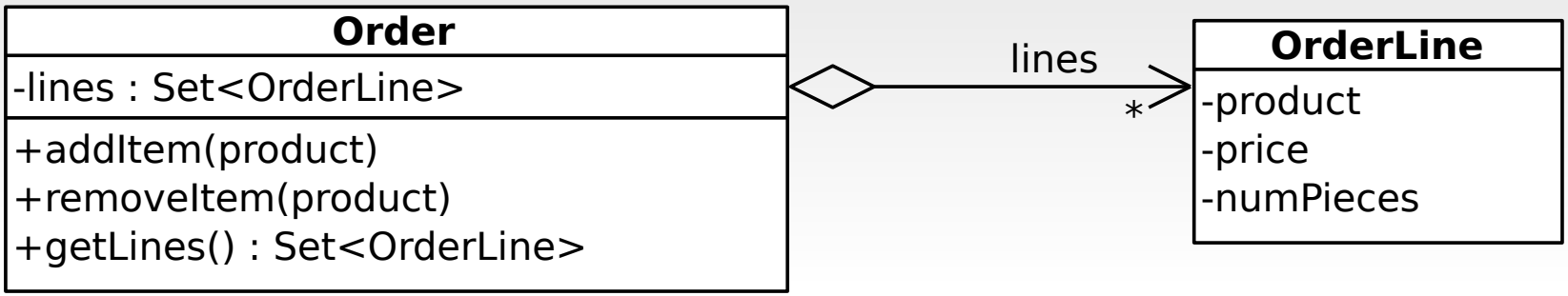
PA103 - Object-oriented Methods for Design of Information Systems

How to Think When Applying Design Patterns

© Radek Ošlejšek
Fakulta informatiky MU
oslejsek@fi.muni.cz

Initial model



Goals

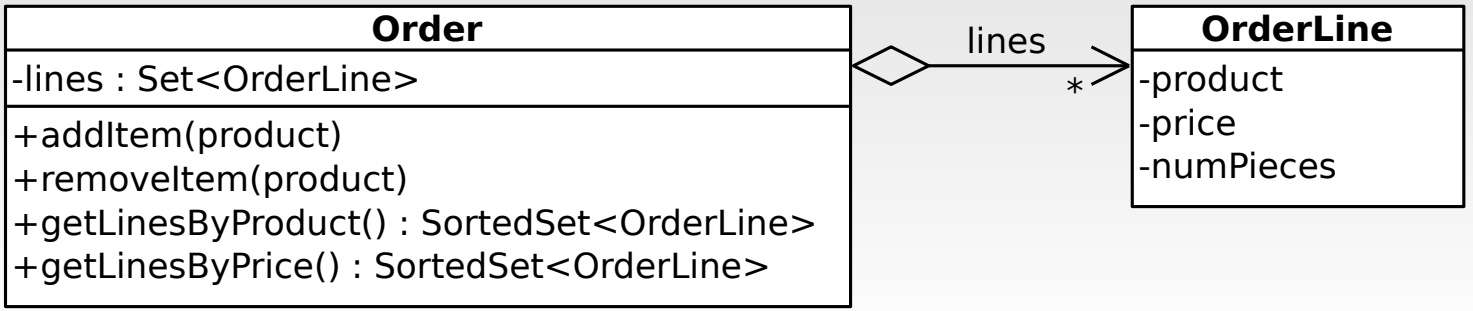
- **Goals:** extend the model so that:
  - A) We are able to sort order lines by name, date, price, and other criteria.s
  - B) We have various types of orders, e.g. for retail customers vs. bulk customers. They differ in stored data.
  - C) both.
- **Approach:**
  - Ad A) Employ **behavioral** design patterns because sorting is behavior.
  - Ad B) Employ **structural** design patterns because we have to deal with spreading pieces of information across classes.
  - Ad C) Put them together.

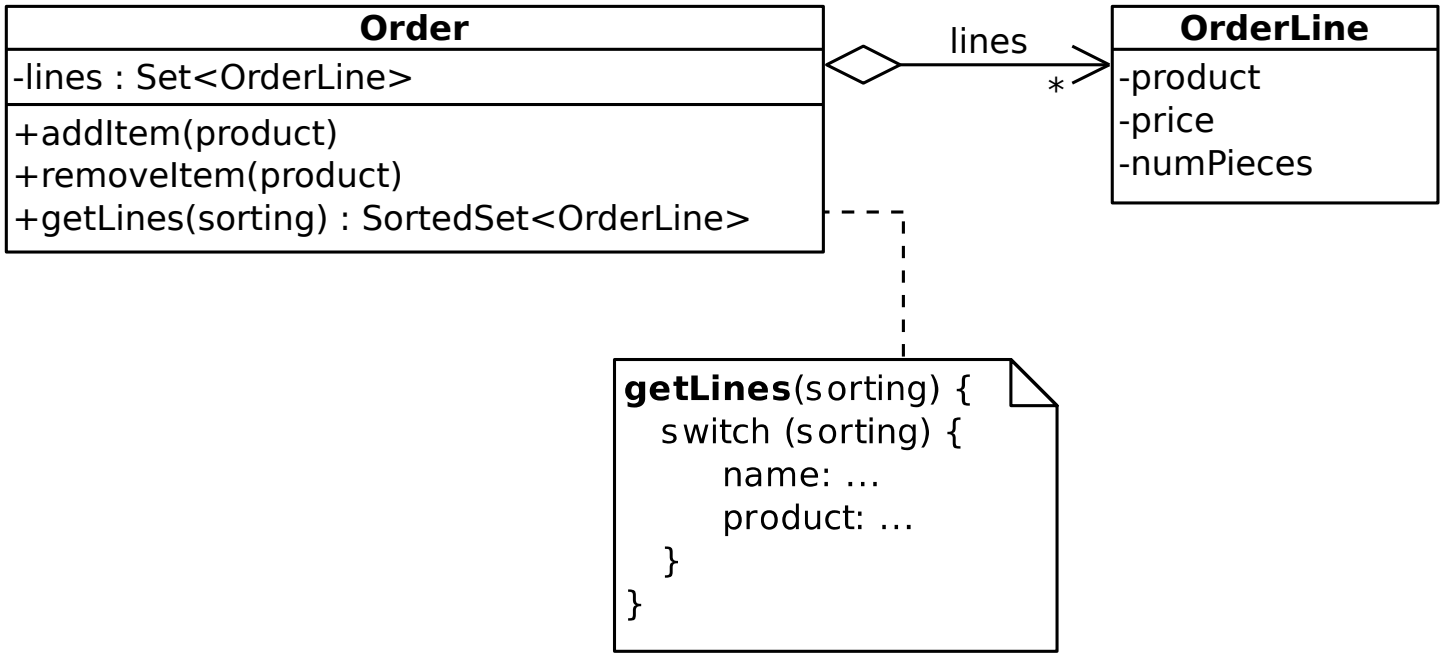
Goal A: Sorted lines

- **Design decision:** Should be the *Order* class responsible for sorting lines or is it the responsibility of that who wants it?
  - Probably the second case is correct. But to demonstrate the application of design pattern we give this responsibility to *Order* class.

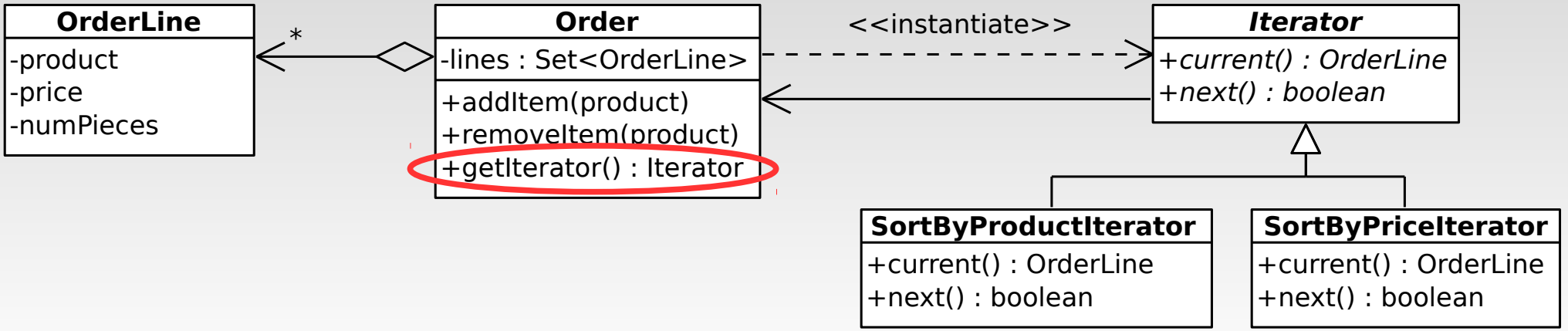
Sorted lines: Naive solution

- Adding a new ordering requires to change the existing *Order* class. Two naive solutions:



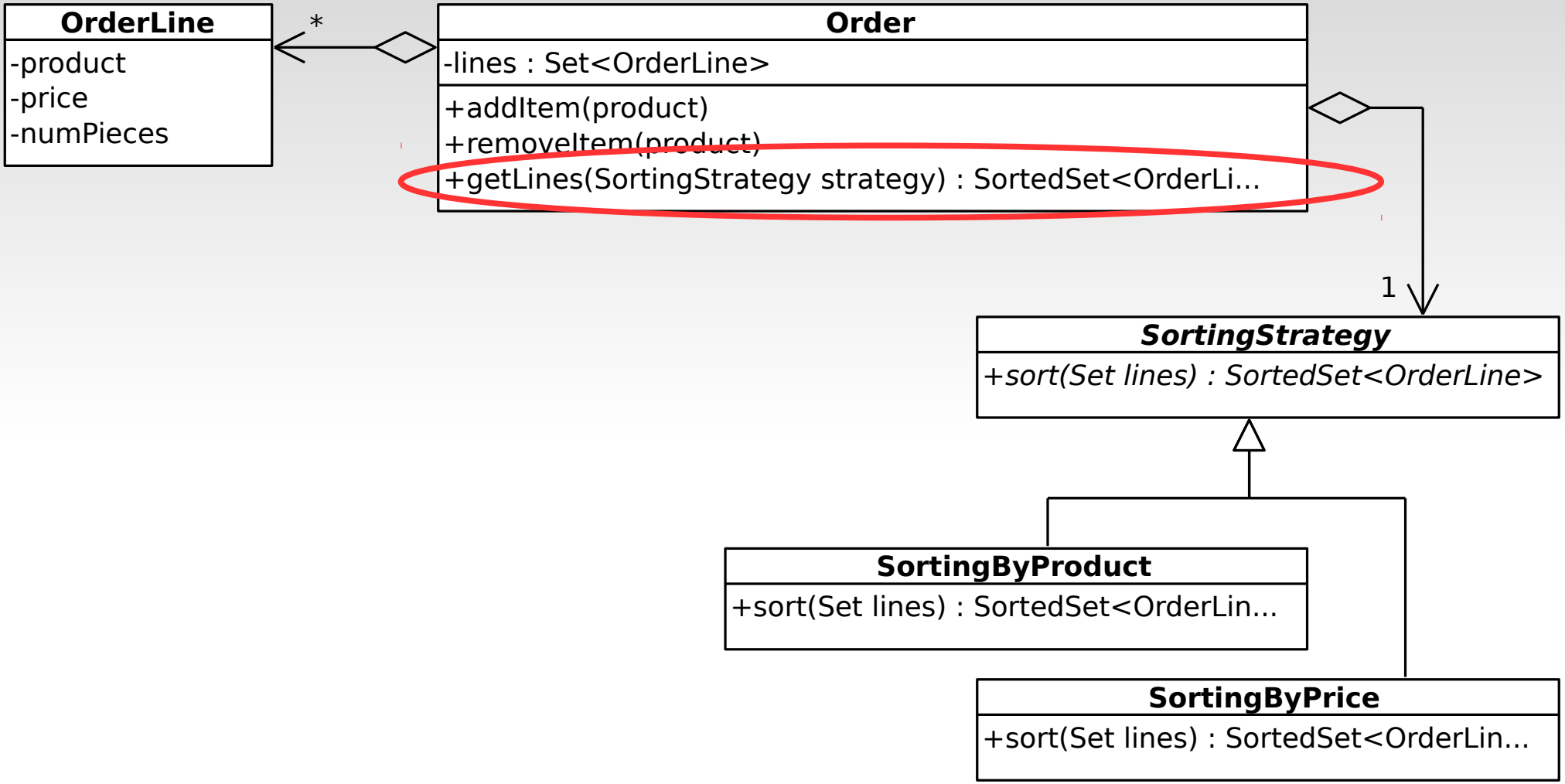


Sorted lines: Iterator



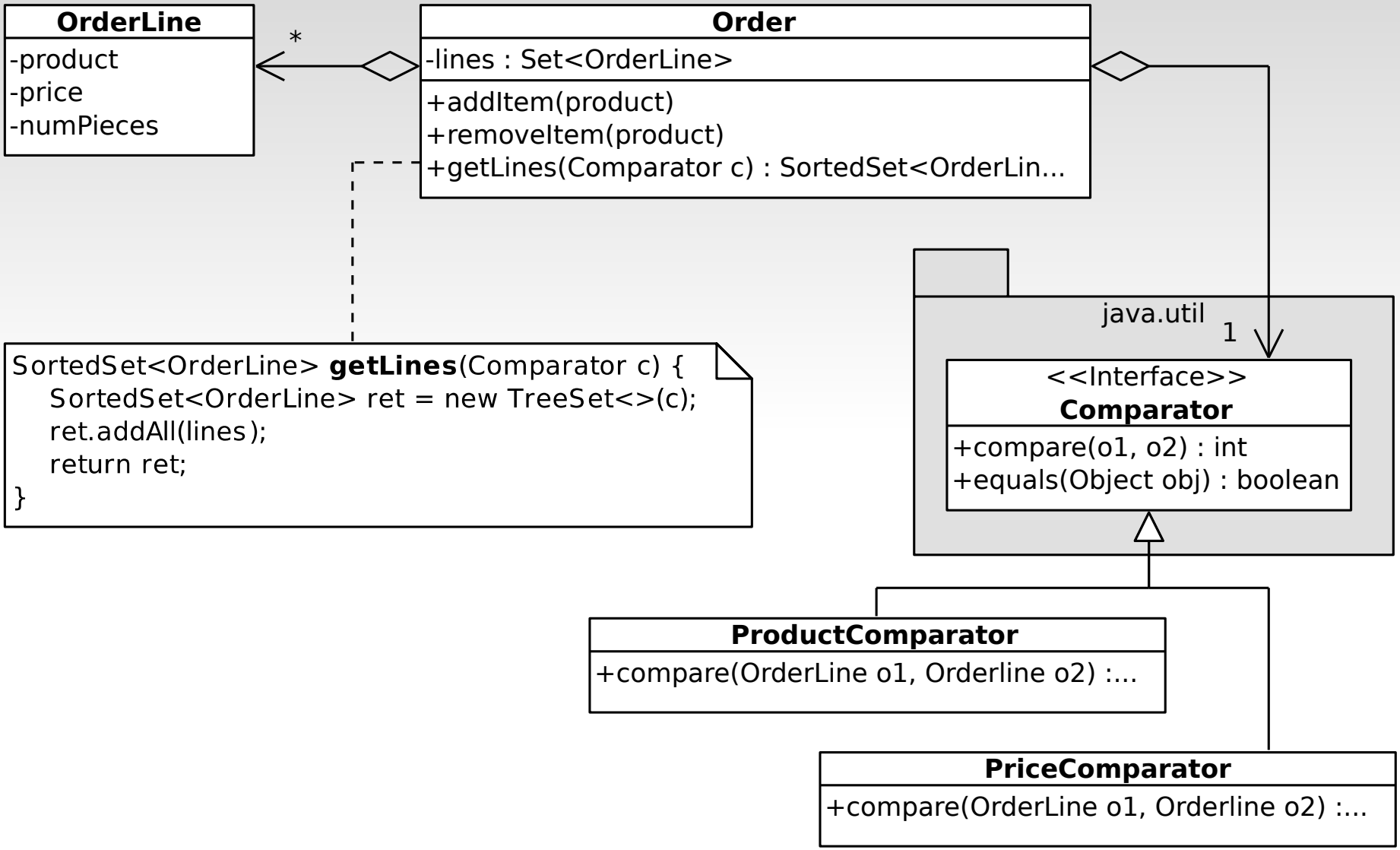
- Iterators providing lines in a specific sequence.
- But how to say what iterator is to be used?
  - By parameter or providing more variants of the *getIterator()* method?
  - Similar problem to the previous naive solution. Nevertheless, the code of the *getIterator()* is more simple than in the naive approach

Sorted lines: Strategy



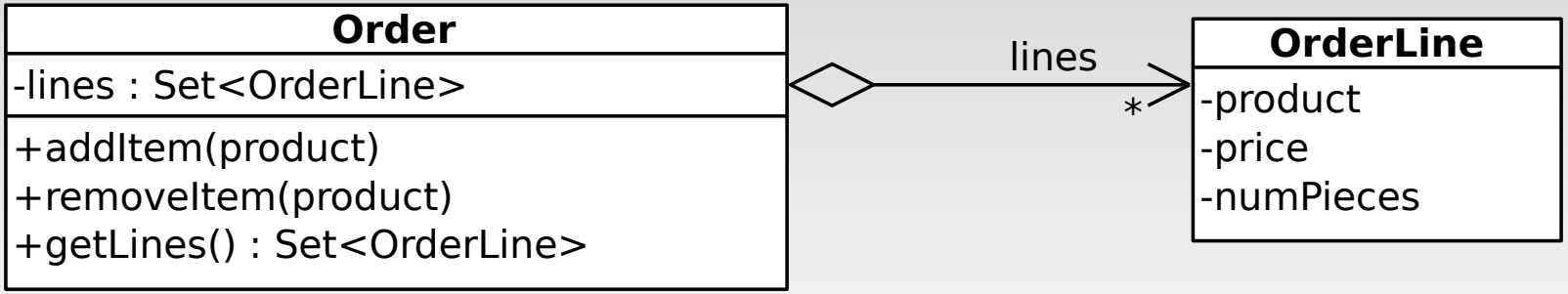
- A client decides which strategy to use!

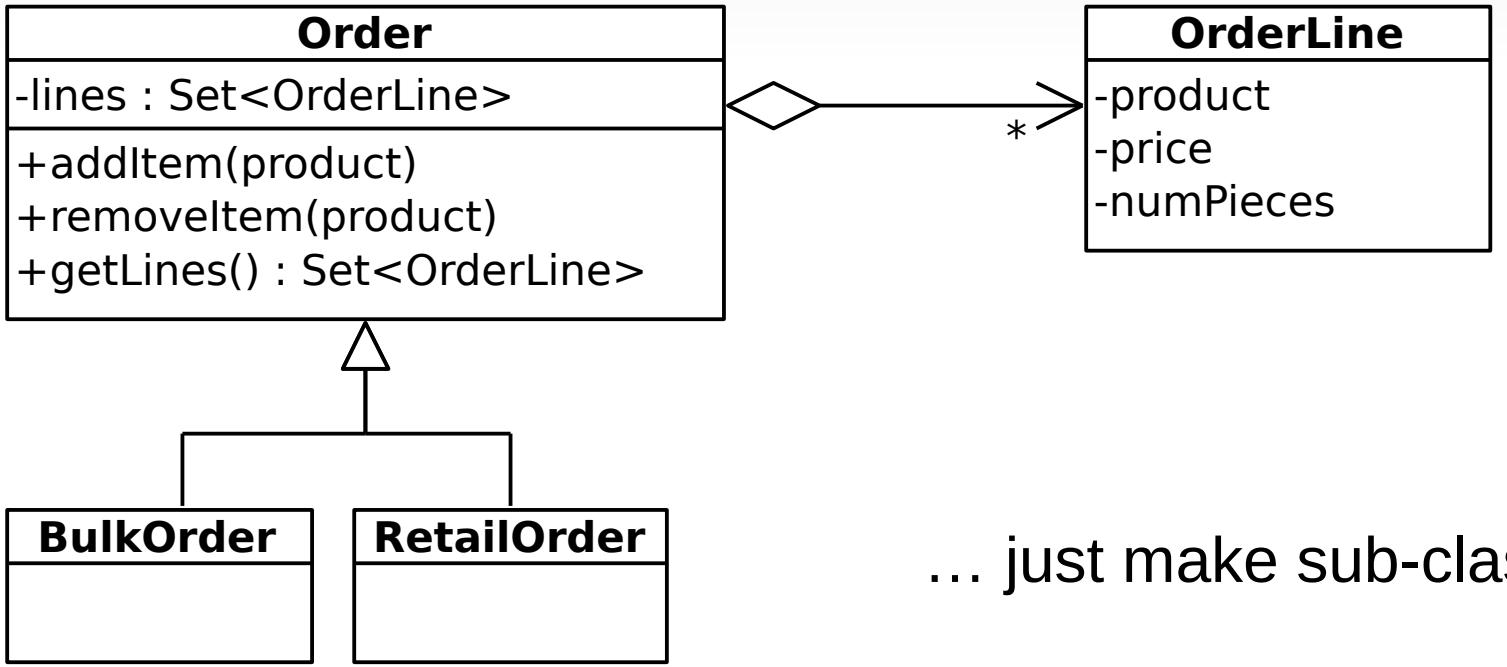
Sorted lines: Strategy in Java = Comparator



- … because *java.util.Comparator* is nothing else than „sorting strategy“ for sorted Java collections.

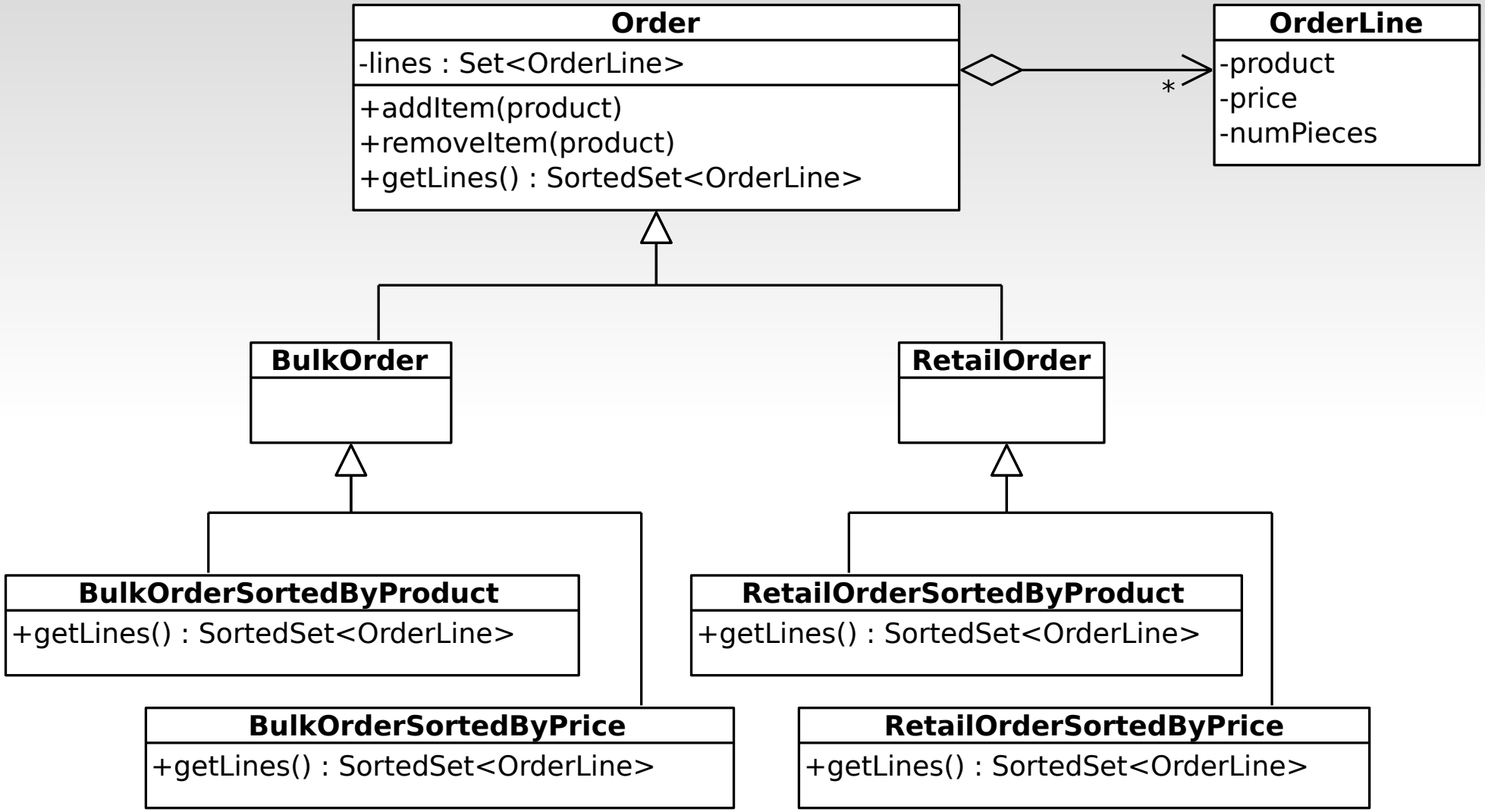
Goal B: Variable Orders





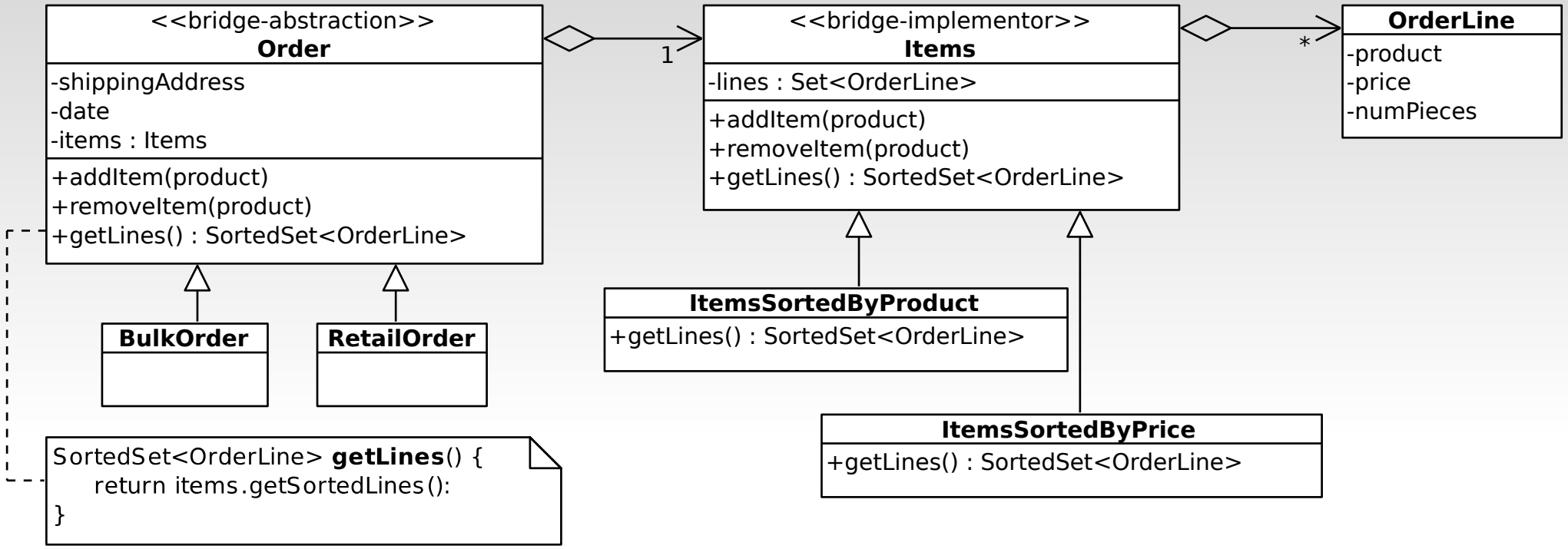
… just make sub-classes

Goal C: Variable Orders With Sorting Strategies



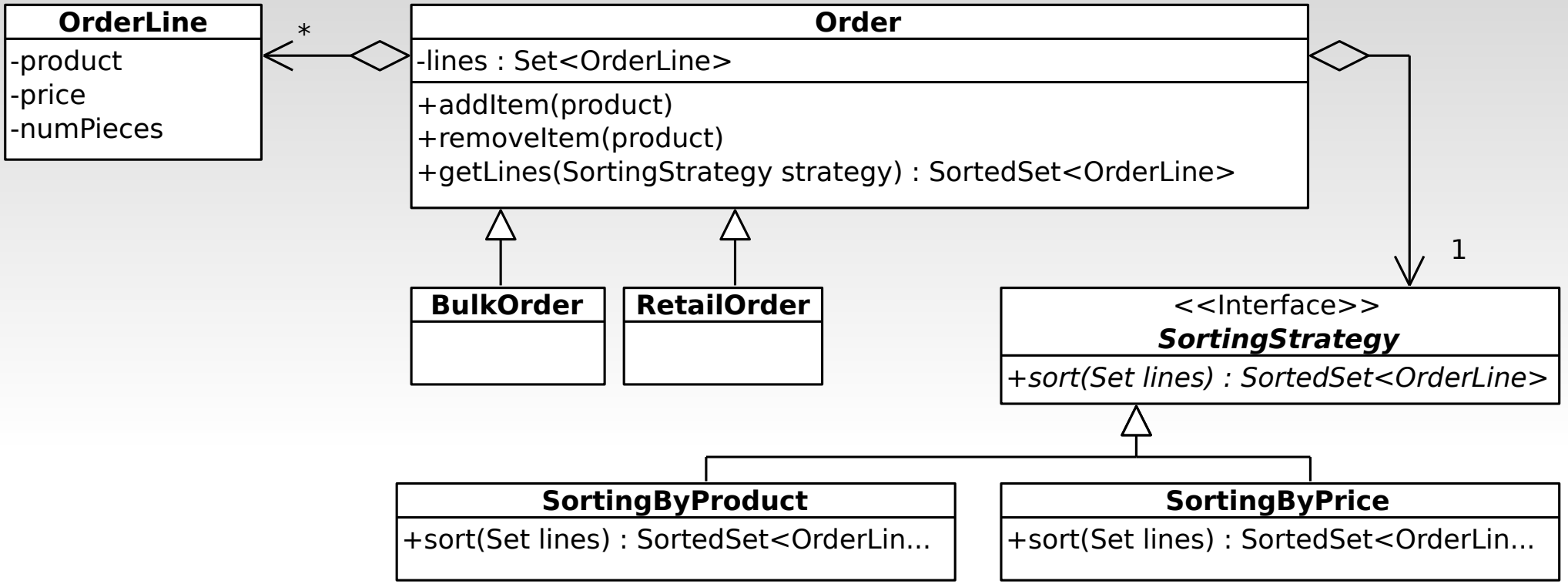
… explosion of sub-classes => apply Bridge

Goal C: Bridge in Action



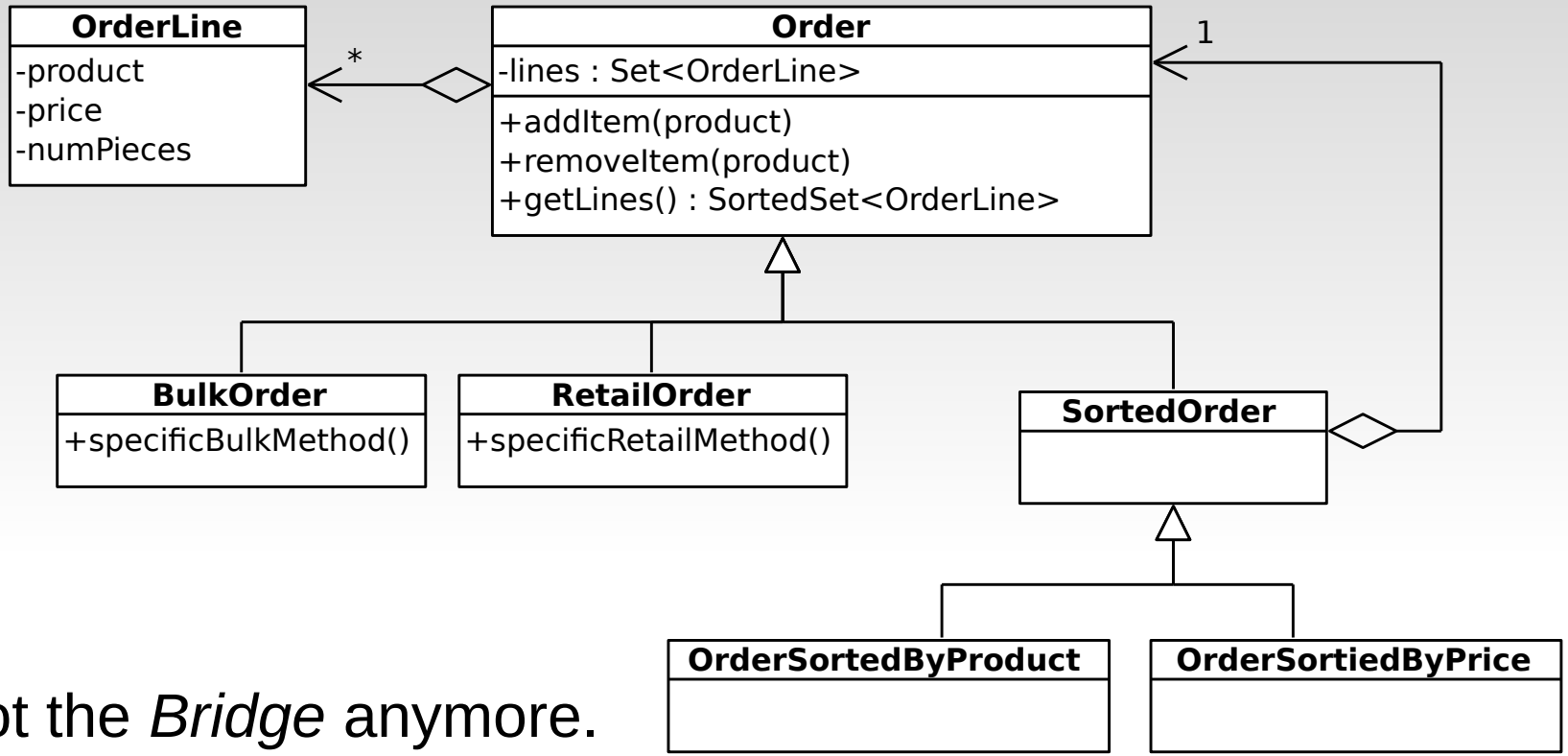
- OK, but Bridge, as a structural pattern, does not explain neither who chooses the *Items* sub-class nor who sets the schoosen sub-class to *Order*. These dynamic aspects are defined by behavioral patterns.
- In this case, the *Items* class can be understood as a sorting strategy.

Goal C: Strategy in Action (Again ;)



- Is it *Bridge* or *Strategy*? It depents on the point of view. Static view => *Bridge*, dynamic view => *Strategy*.
- Only sorting by product or by price is allowed. But what if we want to combine sorting methods?
  - Define combined strategy or use *Decorator*

Goal C: Decorator in Action



- This is not the *Bridge* anymore.
- **Drawback:** *SortedOrder* has not the specific methods of *BulkOrder* and *RetailOrder*. Therefore, clients have to distinguish between them.
- **Drawback:** Implementation of *OrderSortedByPrice* taking into account *OrderSortedByProduct*, for instance, would be much complicated that introducing a new specialized sub-class.

Questions?