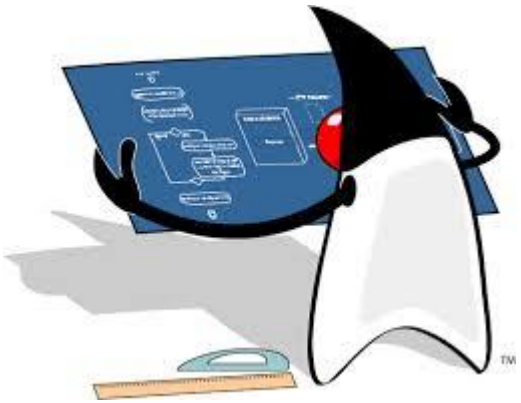
Advanced Topics in Programming - important design patterns

Dr. Eliahu Khalastchi

2017

Design Patterns

Today's lesson is about some important design patterns



Design patterns

- This course is not just about Java,
- We need to learn how to program OOP correctly!
- We need to learn a few design patterns…

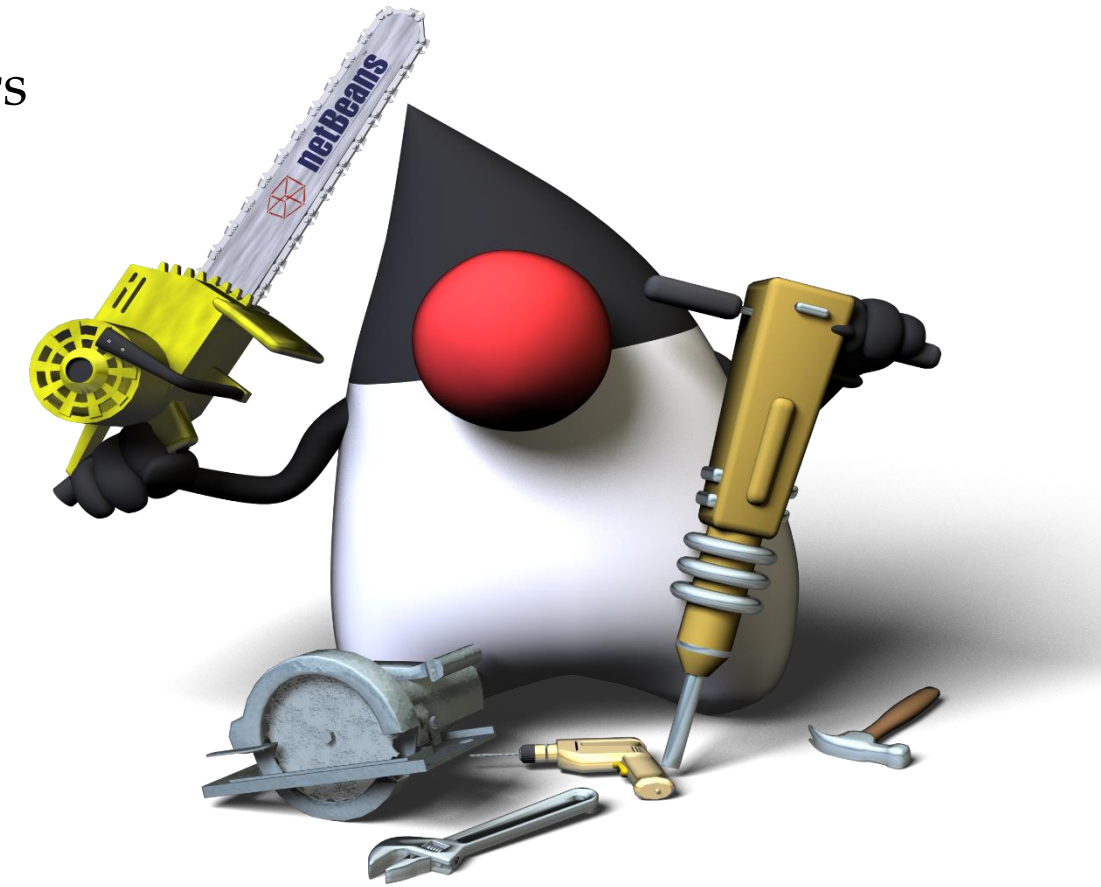
- We will cover the Java implementation of some of the important design patterns along the lectures and recitations
- The project will emphasize the use of design patterns

Lab Exercise 1

Implement a **bubble sort** algorithm to sort workers

- Pseudo code ➔ Object Oriented Java code
- Hierarchy of classes
- strategy design pattern

1. Define a Worker class:
   each Worker has a **name**, an **age**, and a **salary**
2. Create a structure of classes to implement the sorting alg'
3. We want to be able to sort by names, ages, and salaries
   *** in the most flexible way possible!



Bubble Sort Pseudo code:

Is it the only sorting algorithm?

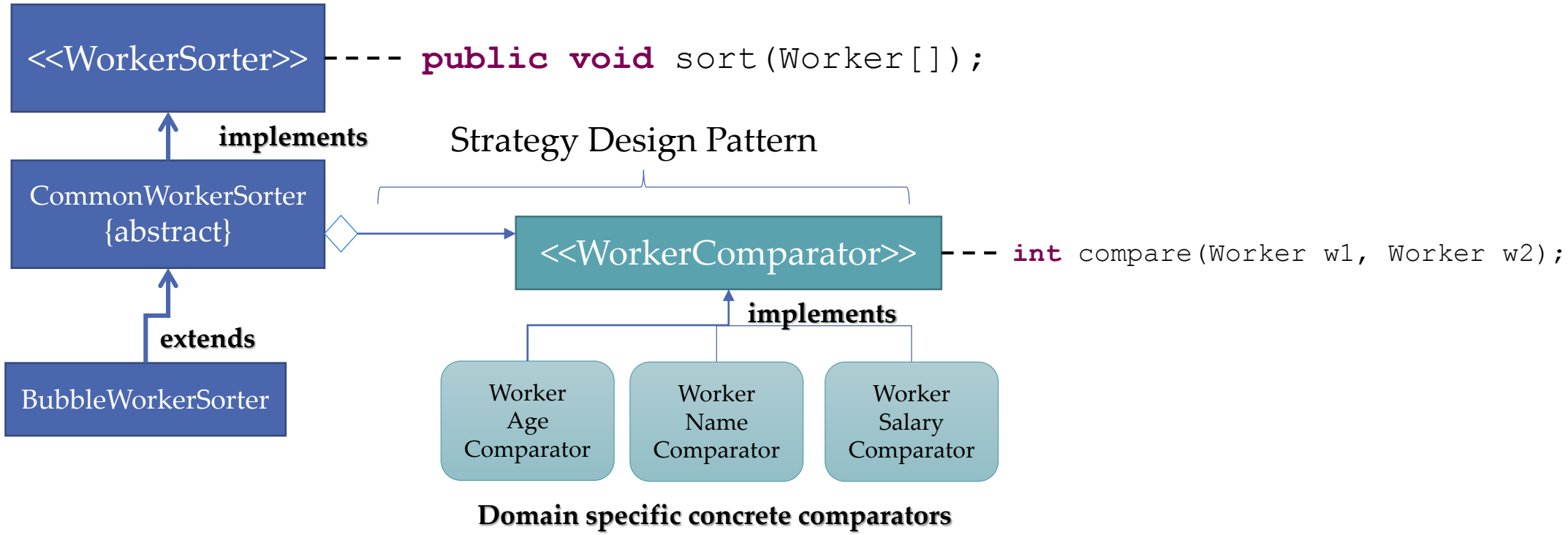
We have 3 different ways to answer this:
name, age, and salary

```
procedure bubbleSort( A : list of sortable items )
  n = length(A)
  repeat
    swapped = false
    for i = 1 to n-1 inclusive do
      /* if this pair is out of order */
      if A[i-1] > A[i] then
        /* swap them and remember something changed */
        swap( A[i-1], A[i] )
        swapped = true
      end if
    end for
  until not swapped
end procedure
```

What to consider?

- We want an algorithm that sorts workers
- Workers can be sorted by
  - name, age or salary…
- We wouldn't want 3 different implementations
  - There is a duplicate code – hard to maintain
- We wouldn't want 3 if statements in the code
  - An if statement for each sorter, damages the code's purity in sorting, and makes it hard to add new sorters…

Solution…


<<WorkerSorter>>
public void sort(Worker[]);
implements
Strategy Design Pattern
CommonWorkerSorter
{abstract}
<<WorkerComparator>>
int compare(Worker w1, Worker w2);
extends
implements
BubbleWorkerSorter
Worker Age Comparator
Worker Name Comparator
Worker Salary Comparator
Domain specific concrete comparators

Test your code!

```
Worker[] workers=new Worker[5];
workers[0]=new Worker("david",25,2500);
workers[1]=new Worker("moshe",28,2700);
workers[2]=new Worker("shalom",24,2600);
workers[3]=new Worker("israel",29,2900);
workers[4]=new Worker("yosef",30,2000);

Sorter ws=new BubbleWorkerSorter(new WorkerSalaryComparator());
ws.sort(workers);

for(Worker w: workers){
 System.out.println(w.getName()+"\t"+w.getAge()+"\t"+w.getSalary());
}
```

yosef	30	2000
david	25	2500
shalom	24	2600
moshe	28	2700
israel	29	2900

SOLID checklist

- Single Responsibility Principle
- Open / Closed Principle
- Liskov Substitution Principle
- Interface Segregation Principle
- Dependency Inversion Principle

Lab Exercise 2

Apply the **bubble sort** algorithm in a task manager

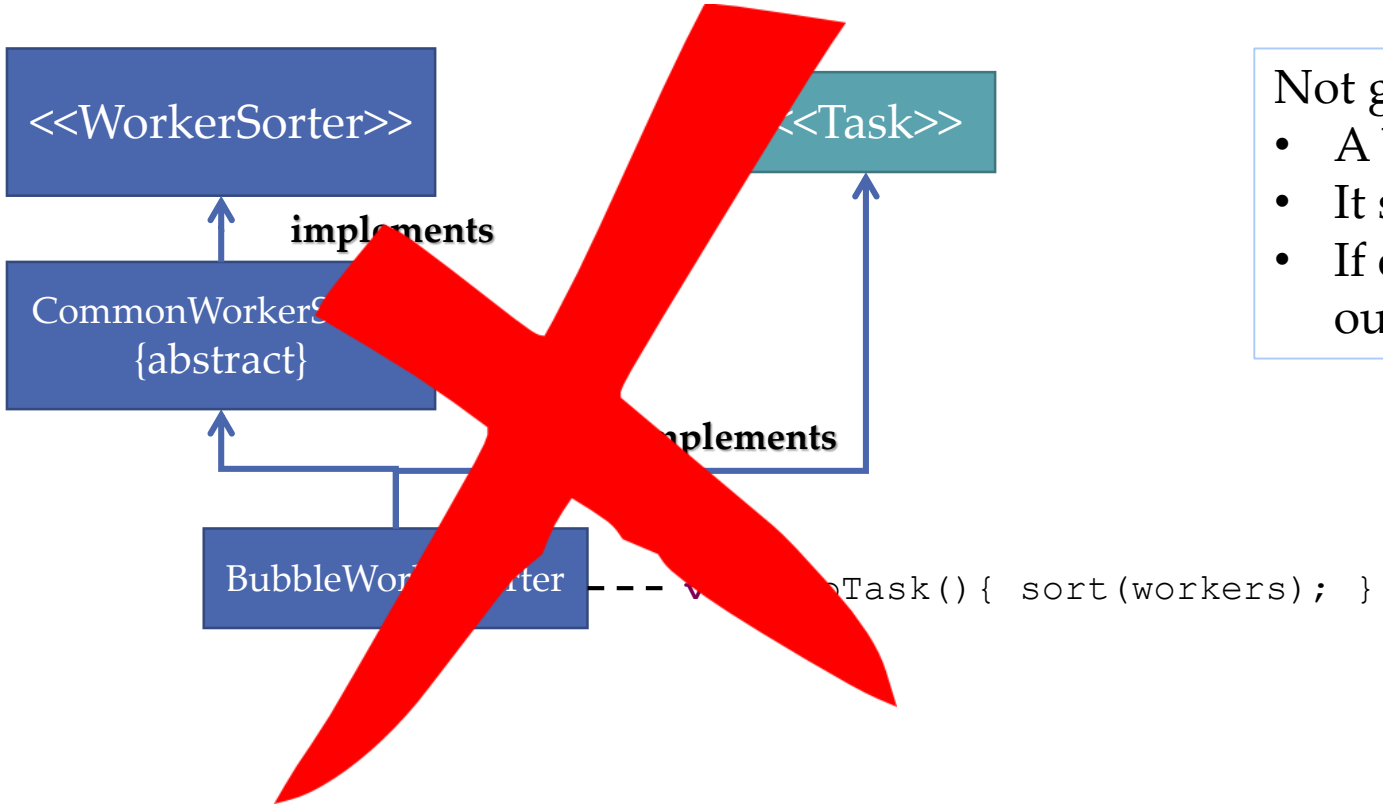
- Class adapter
- Object adapter

1. Define a **Task** interface with a method:
   public void doTask();

2. Define a class **TaskManager** with a method:
   public void runInBackground(Task t);

3. Get the bubble sort to run as a task

Solution 1: implement both interfaces



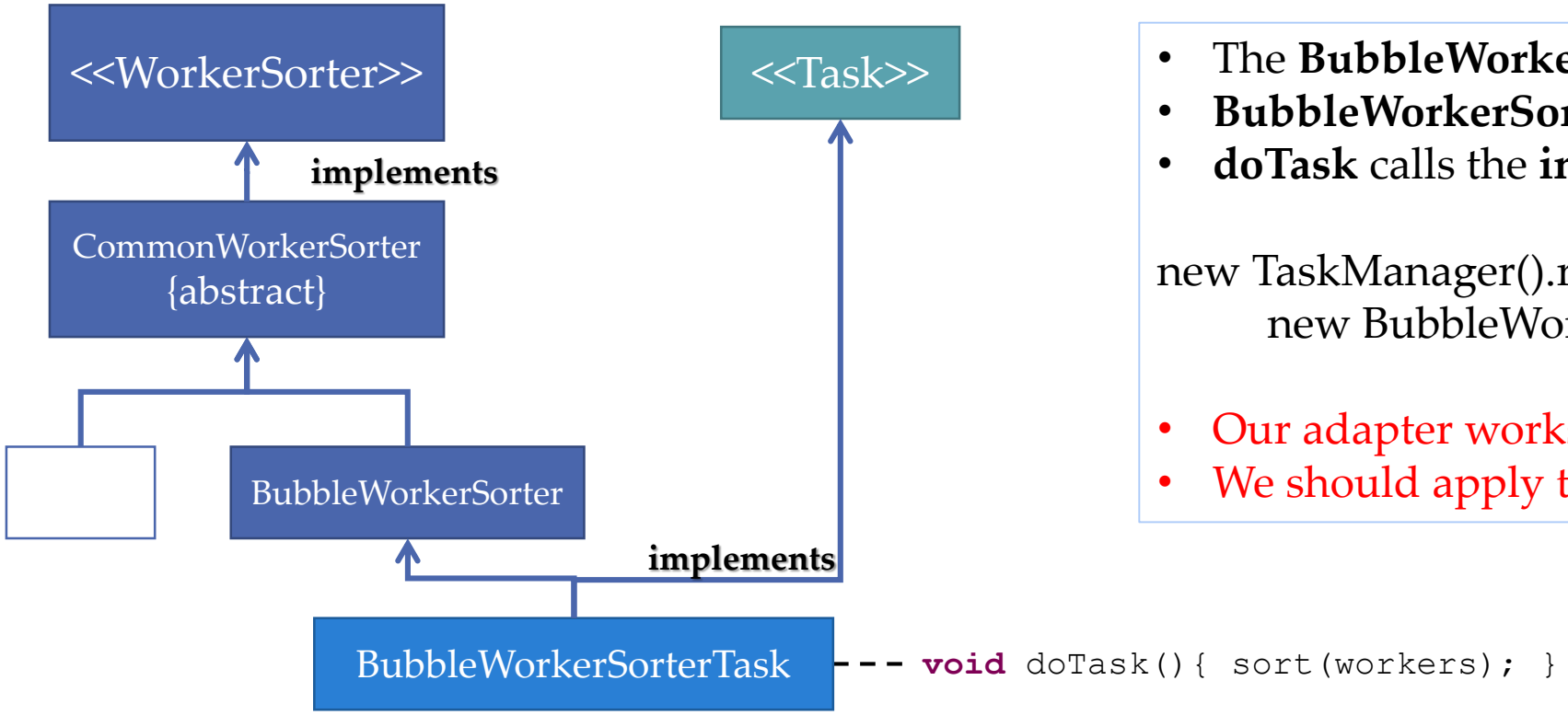
Not good!

- A **BubbleWorkerSorter** is not a **Task**!
- It should stay a pure sorter…
- If each interface has 10 methods then our class will have 20!

SOLID checklist

- Single Responsibility Principle
- Open / Closed Principle
- Liskov Substitution Principle
- Interface Segregation Principle
- Dependency Inversion Principle

Solution 2: class adapter



- The **BubbleWorkerSorter** stays a pure sorter…
- **BubbleWorkerSorterTask** is a **Task**
- **doTask** calls the **inherited sort** method

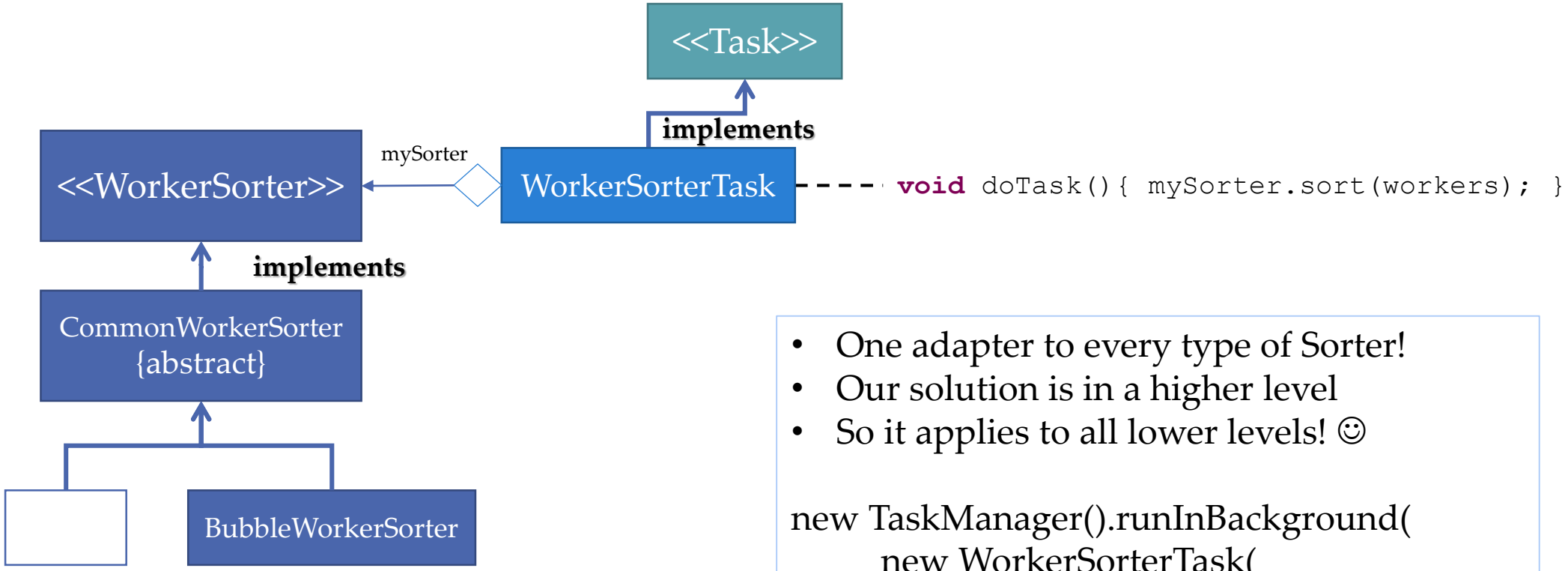
```
new TaskManager().runInBackground(
        new BubbleWorkerSorterTask());
```

- Our adapter works only for bubble sort
- We should apply the same solution to each sorter…

SOLID checklist

- Single Responsibility Principle
- Open / Closed Principle
- Liskov Substitution Principle
- Interface Segregation Principle
- Dependency Inversion Principle

Solution 3: object adapter



- One adapter to every type of Sorter!
- Our solution is in a higher level
- So it applies to all lower levels! ☺

```
new TaskManager().runInBackground(
        new WorkerSorterTask(
                new BubbleWorkerSorter()));
```

SOLID checklist

- Single Responsibility Principle
- Open / Closed Principle
- Liskov Substitution Principle
- Interface Segregation Principle
- Dependency Inversion Principle

Lab Exercise 3

Make the task manager a singleton

- Make sure only one object can be instantiated!

Singleton pattern

- Sometimes we want to use an object that is allocated only once in the memory
  - Because it takes large amount of memory
  - Because it manages some queue
  - etc…

- We need to prevent instancing of the class
- And provide only one instance
- How do you suggest we do that?

Solution - Singleton pattern

```
public class TaskManager {
 private static TaskManager instance=null;
 private TaskManager() {
// "private" CTOR, prevents others from
// instancing it
 }
 public static TaskManager getInstance(){
  if(instance==null){
   instance=new TaskManager();
  }
  return instance;
 }
// define here other methods...
}
```

```
TaskManager tm=new TaskManager(); // error!
```

```
TaskManager tm=TaskManager.getInstance();
tm.runInBackGround(...);
```