CS 320

Introduction to Software Engineering
Spring 2017

March 20 2017

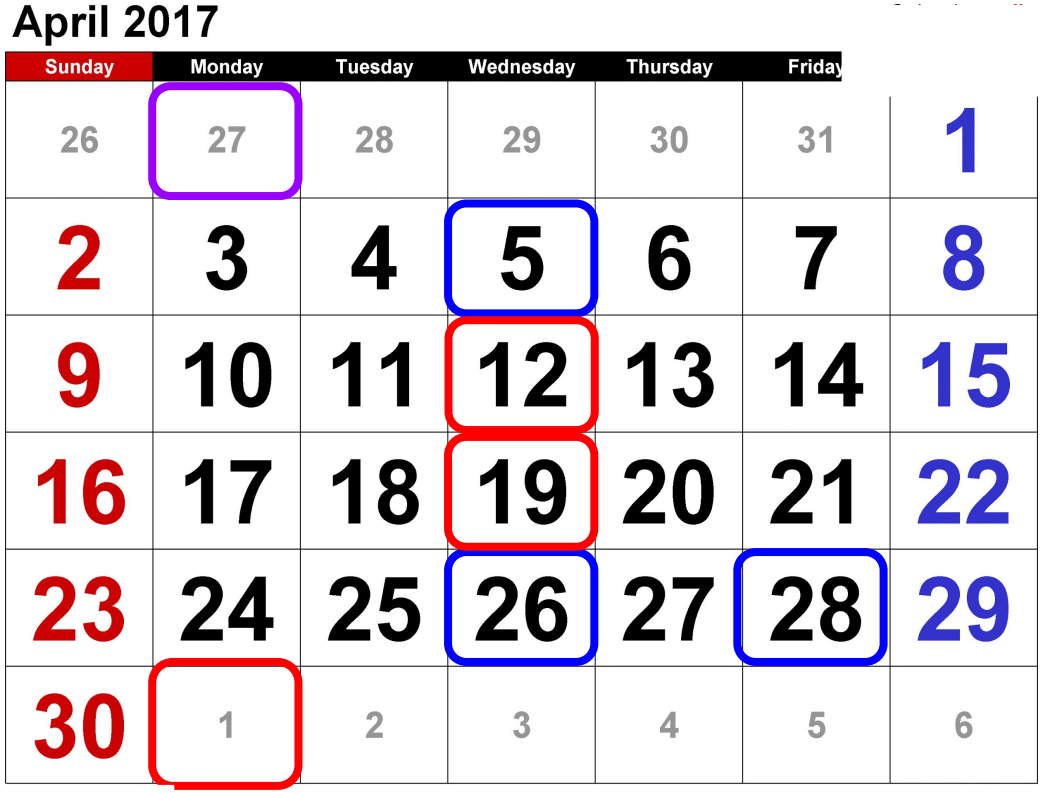
Today

- Logistics: timeline and assignments
- Overview of the study guide (midterm exam)
- Overview of the SDD requirements

- Recap of software design principles
- A concrete software design problem

Logistics

Logistics: timeline and assignments



03/27: Midterm exam

04/05: Finalize SDD

04/12: In-class exercise

04/19: In-class exercise

04/26: Project presentation

04/28: Final deliverables

05/01: In-class exercise (extra credit)

All deadlines and assignments are posted on the course website and Moodle.

Midterm exam: study guide

Logistics

- **60 minutes**
- Closed book, closed notes.
- No laptops or mobile devices.

Topics (see study guide on Moodle)

- The software development process -- 30%
- Verification & validation and program analysis -- 10%
- Object oriented programming -- 15%
- Software design principles -- 30%
- Version control systems -- 15%

Discussion session on 03/22: brainstorm/discuss your solutions and come up with clarification questions.

SDD: format and requirements

The SDD should describe the following:

- **Overview**
  - Purpose
  - Assumptions and definitions
  - References (e.g., to the SRS)
- **Software architecture**
  - High-level overview of components
  - Include a diagram of the overall system architecture
  - Specification of inter-component interfaces
- **Data design and representation**

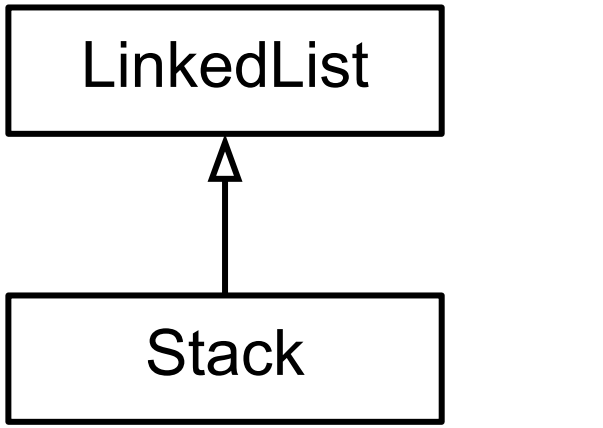
SDD: format and requirements (cont.)

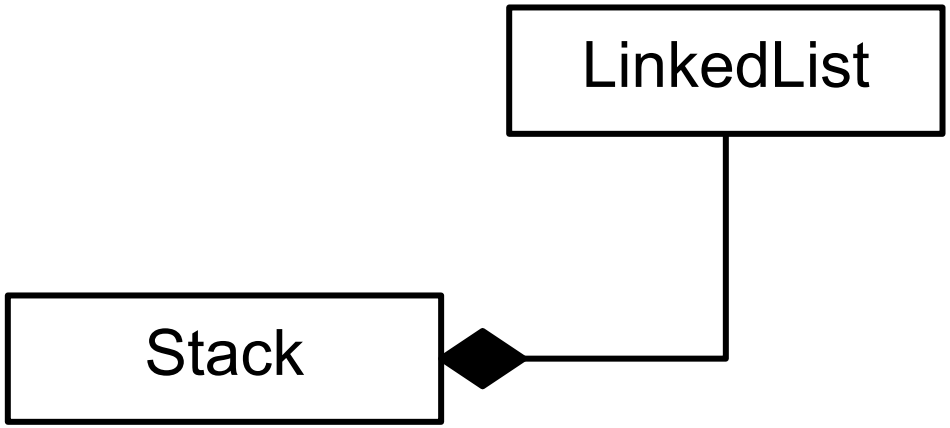
- **Software design**
  - Detailed design of each component
  - Include a diagram for each component
  - Specification of intra-component interfaces

- **User/external interface design**
- **Design for testability**
- **Rationale for the chosen architecture and design**

Recap: software design principles

Recap: inheritance vs. composition/aggregation

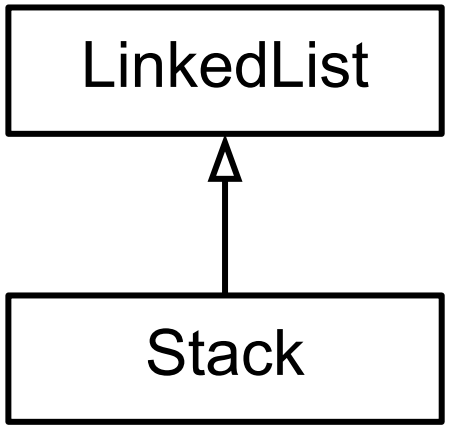





```
public class Stack<E>
     extends LinkedList<E> {
  ...
}
```

```
public class Stack<E> {
  private List<E> l = new LinkedList<>();
  ...
}
```

Recap: inheritance vs. composition/aggregation

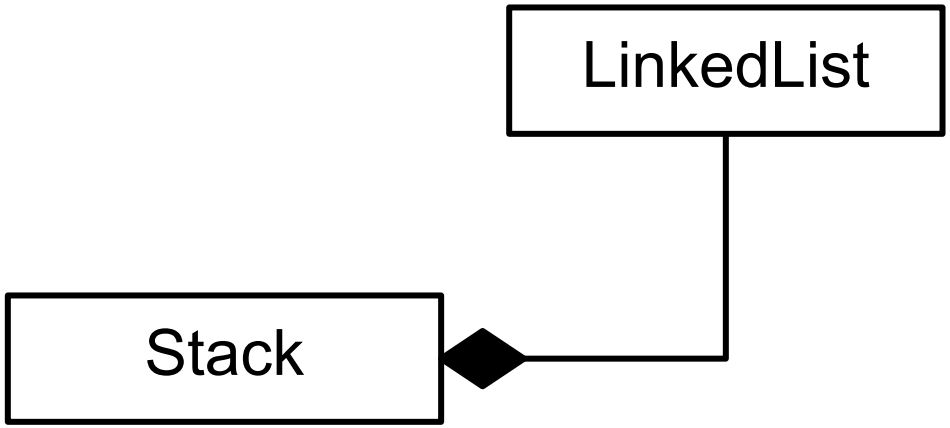


Pros

- No delegation methods required.
- Reuse of common state and behavior.

Cons

- Exposure of all inherited methods (a client might rely on this particular superclass -> can't change it later).
- Changes in superclass are likely to break subclasses.



Pros

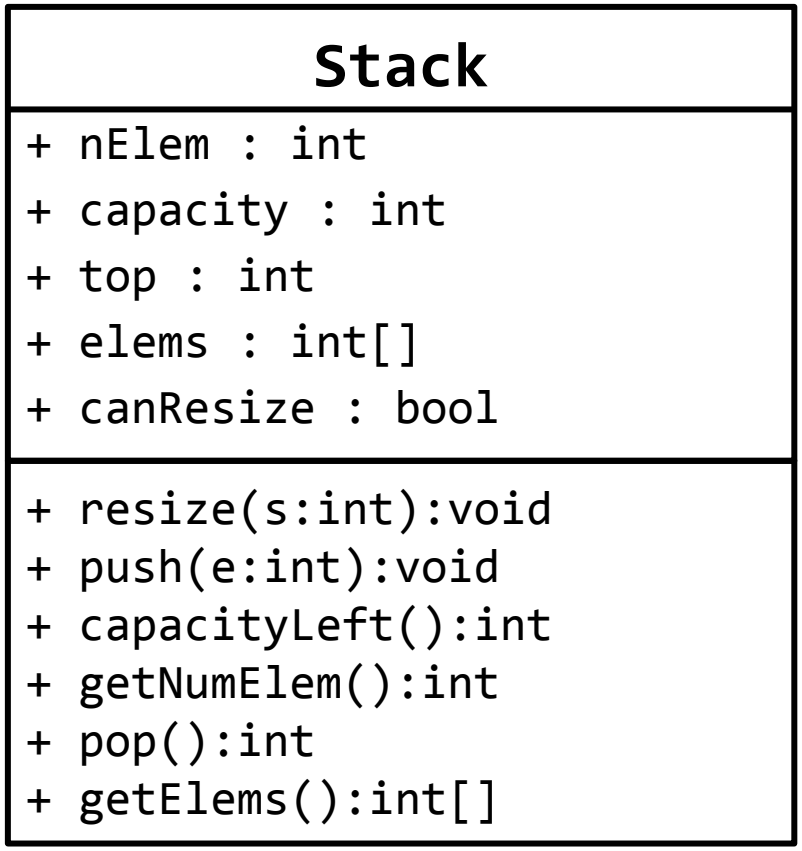
- Highly flexible and configurable.
- No additional subclasses required for different has-a relationships.

Cons

- All interface methods need to be implemented -> delegation methods required, even for code reuse.

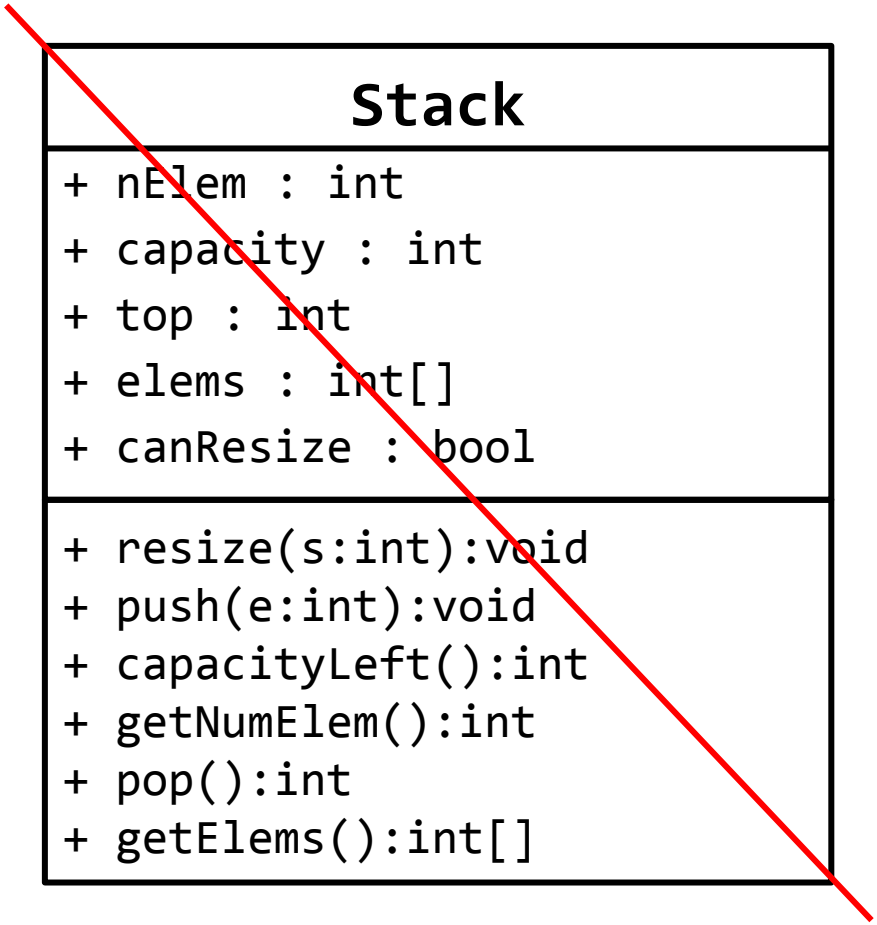
Composition/aggregation over inheritance allows more flexibility.

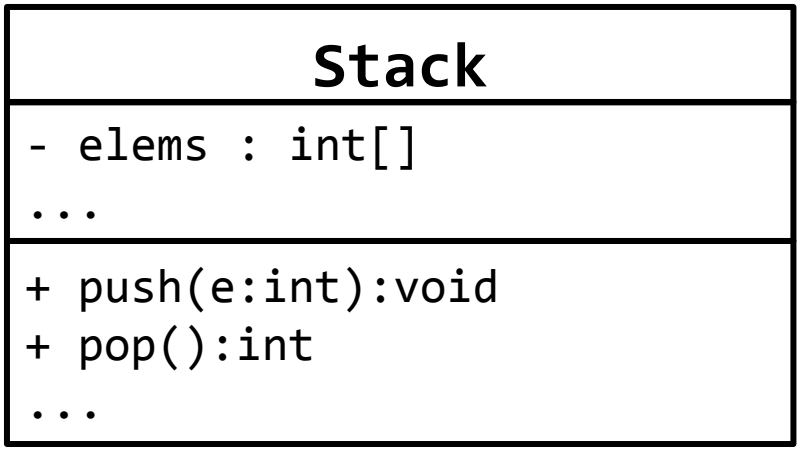
Recap: information hiding




```
public class Stack {
  public int nElem;
  public int capacity;
  public int top;
  public int[] elems;
  public boolean canResize;
  ...
  public void resize(int s){...}
  public void push(int e){...}
  public int capacityLeft(){...}
  public int getNumElem(){...}
  public int pop(){...}
  public int[] getElems(){...}
}
```

Recap: information hiding

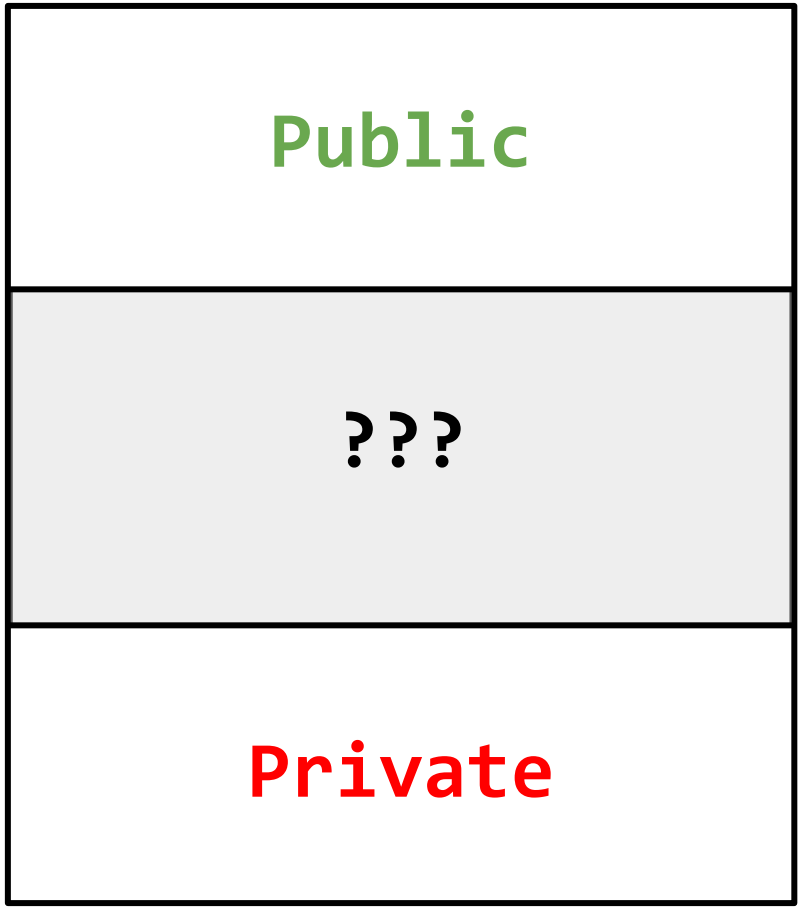




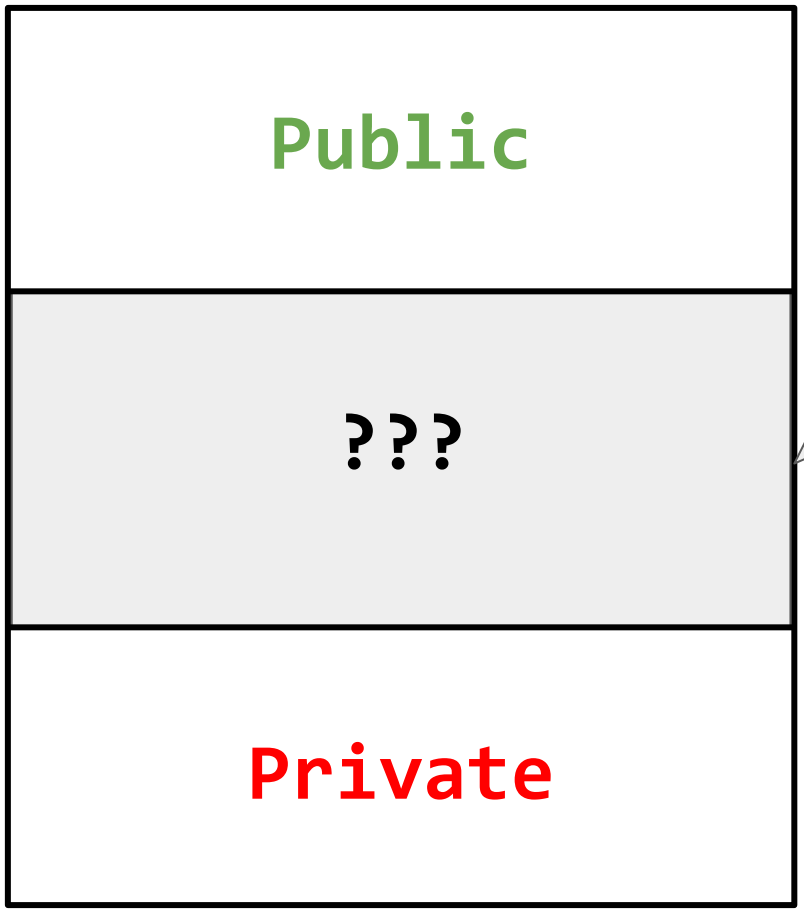
Information hiding:

- Reveal as little information about internals as possible.
- Segregate public interface and implementation details.
- Reduces complexity.

Recap: information hiding vs. visibility



Recap: information hiding vs. visibility



- Protected, package-private, or friend-accessible (C++).
- Not part of the public API.
- Implementation detail that a subclass/friend may rely on.

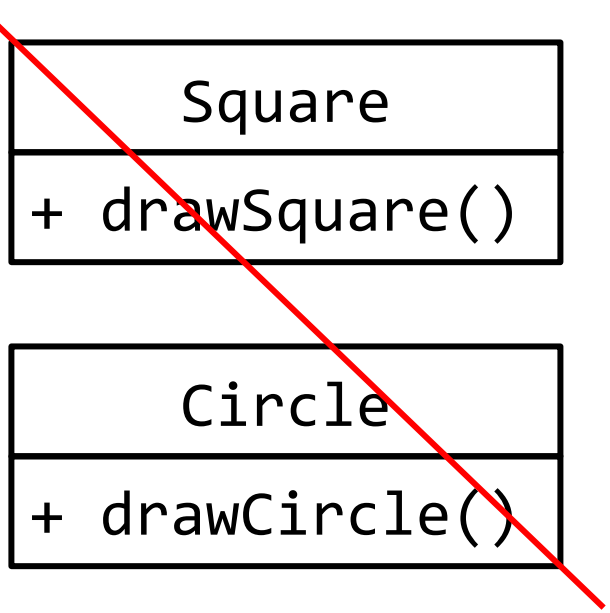
Recap: design principles

Open/closed principle

Software entities (classes, components, etc.) should be:

- **open** for extensions
- **closed** for modifications

```
public static void draw(Object s) {
  if (s instanceof Square) {
    drawSquare((Square) s)
  } else if (s instanceof Circle) {
    drawCircle((Circle) s);
  } else {
    ...
  }
}
```



Recap: design principles

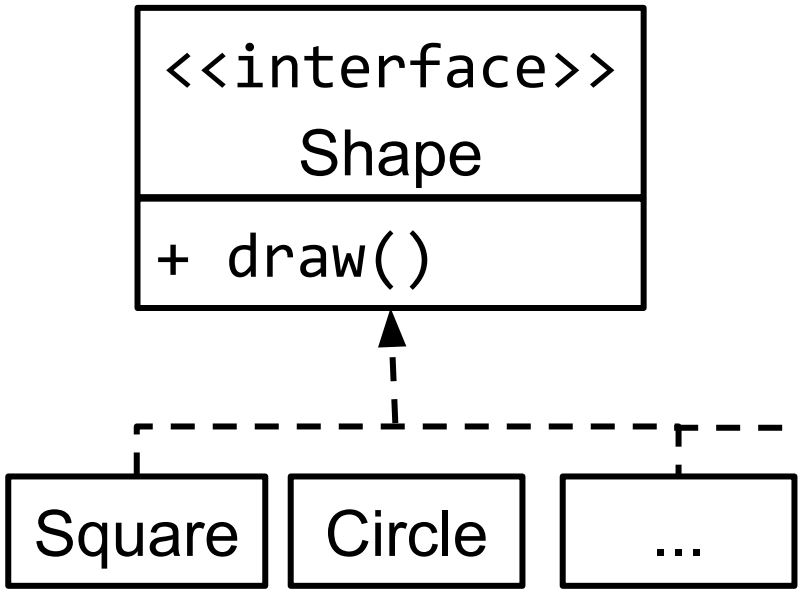
Open/closed principle

Software entities (classes, components, etc.) should be:

- **open** for extensions
- **closed** for modifications

```
public static void draw(Object s) {
  if (s instanceof Shape) {
     s.draw();
  } else {
     …
  }
}
```

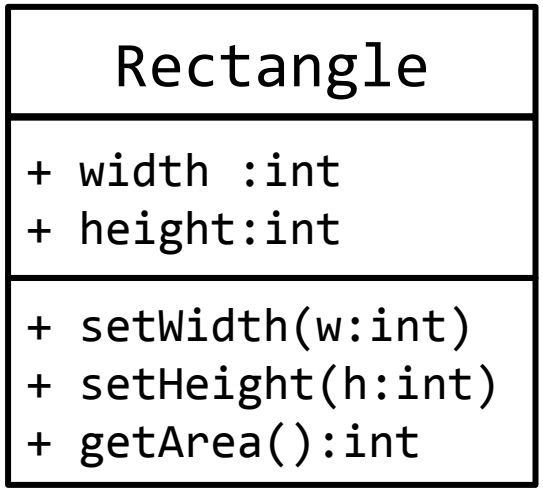
```
public static void draw(Shape s) {
  s.draw();
}
```



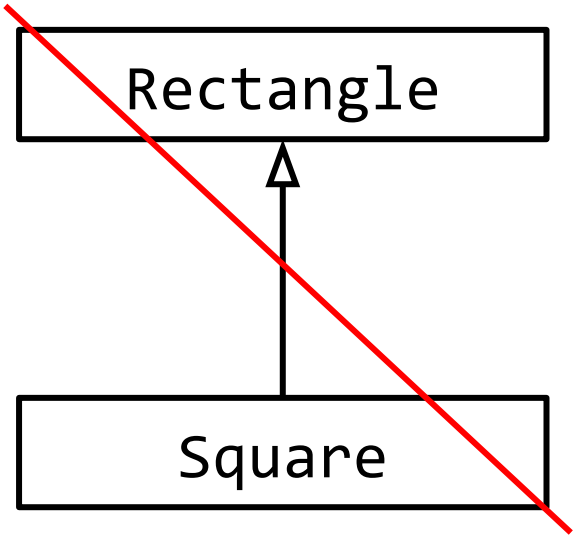
Recap: design principles

Liskov substitution principle

Let object x be of type T1 and object y be of type T2. Further, let T2 be a subtype of T1 (T2 <: T1). Any provable property about objects of type T1 should be true for objects of type T2.



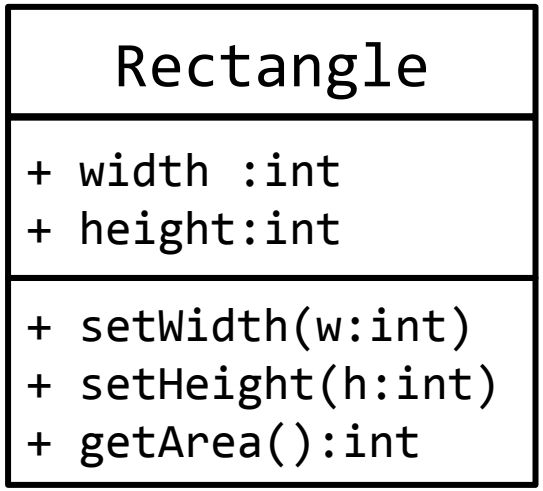

```
Rectangle r =
  new Rectangle(2,2);
  new Square(2);

int A = r.getArea();
int w = r.getWidth();
r.setWidth(w * 2);

assertEquals(A * 2,
    r.getArea());
```

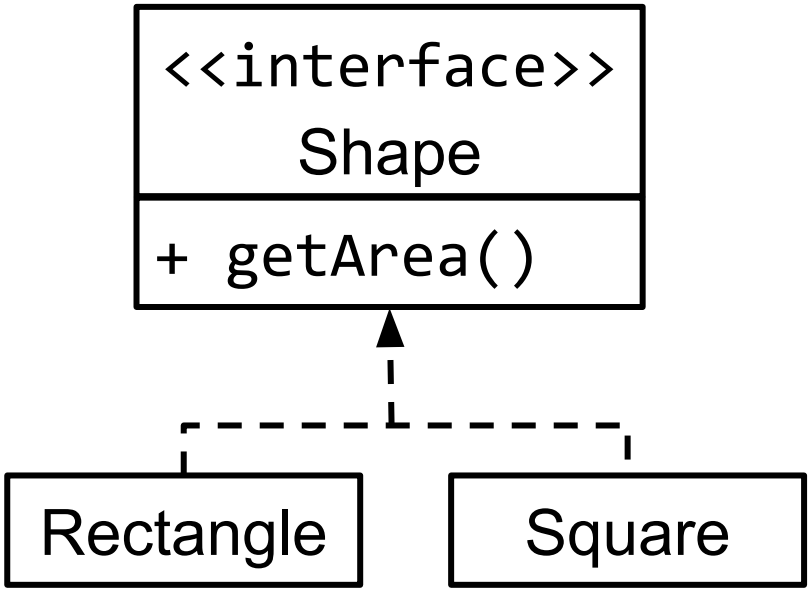


Recap: design principles

Liskov substitution principle

Let object x be of type T1 and object y be of type T2. Further, let T2 be a subtype of T1 (T2 <: T1). Any provable property about objects of type T1 should be true for objects of type T2.





A concrete design problem

Find the median in an array of doubles

Examples:

- median([1, 2, 3, 4, 5]) = ???
- median([1, 2, 3, 4]) = ???

Find the median in an array of doubles

Examples:

- median([1, 2, 3, 4, 5]) = 3
- median([1, 2, 3, 4]) = 2.5

Algorithm:

Input: *array* of length *n* **Output**: median

Find the median in an array of doubles

Examples:

- median([1, 2, 3, 4, 5]) = 3
- median([1, 2, 3, 4]) = 2.5

Algorithm:

Input: *array* of length *n* **Output**: median

1. Sort *array*
2. if *n* is *odd* return (*(n+1)/2)*th element
   otherwise return arithmetic mean of
   *(n/2)*th element and *((n/2)+1)*th element

Naïve solution

```
public static void main(String ... args) {
     System.out.println(median(1,2,3,4,5));
}

public static double median(double ... numbers) {
     int n = numbers.length;
     boolean swapped = true;
     while(swapped) {
          swapped = false;
          for (int i = 1; i<n; ++i) {
               if (numbers[i-1] > numbers[i]) {
                    ...
                    swapped = true;
               }
          }
     }
     if (n%2 == 0) {
          return (numbers[(n>>1) - 1] + numbers[n>>1]) / 2;
     } else {
          return numbers[n>>1];
     }
}
```

Source code is available on the course web site.

Naïve solution

```
public static void main(String ... args) {
     System.out.println(median(1,2,3,4,5));
}

public static double median(double ... numbers) {
     int n = numbers.length;
     boolean swapped = true;
     while(swapped) {
          swapped = false;
          for (int i = 1; i<n; ++i) {
               if (numbers[i-1] > numbers[i]) {
                    ...
                    swapped = true;
               }
          }
     }
     if (n%2 == 0) {
          return (numbers[(n>>1) - 1] + numbers[n>>1]) / 2;
     } else {
          return numbers[n>>1];
     }
}
```

What's wrong with this design (extensibility, testability, etc.)? How can we improve it?

Source code is available on the course web site.

Naïve solution: minor improvements

See code examples (online)

- naive
  - 1: Monolithic version, static context.
  - 2: Extracted sorting method, non-static context.
  - 3: Proper package structure and visibility, extracted main method.
  - 4: Proper testing infrastructure and build system.

Use `ant` to compile and test the code:

```
$ant -p          => list all targets
$ant compile     => compile the code
$ant test        => run all tests
```

One possible solution: template method pattern

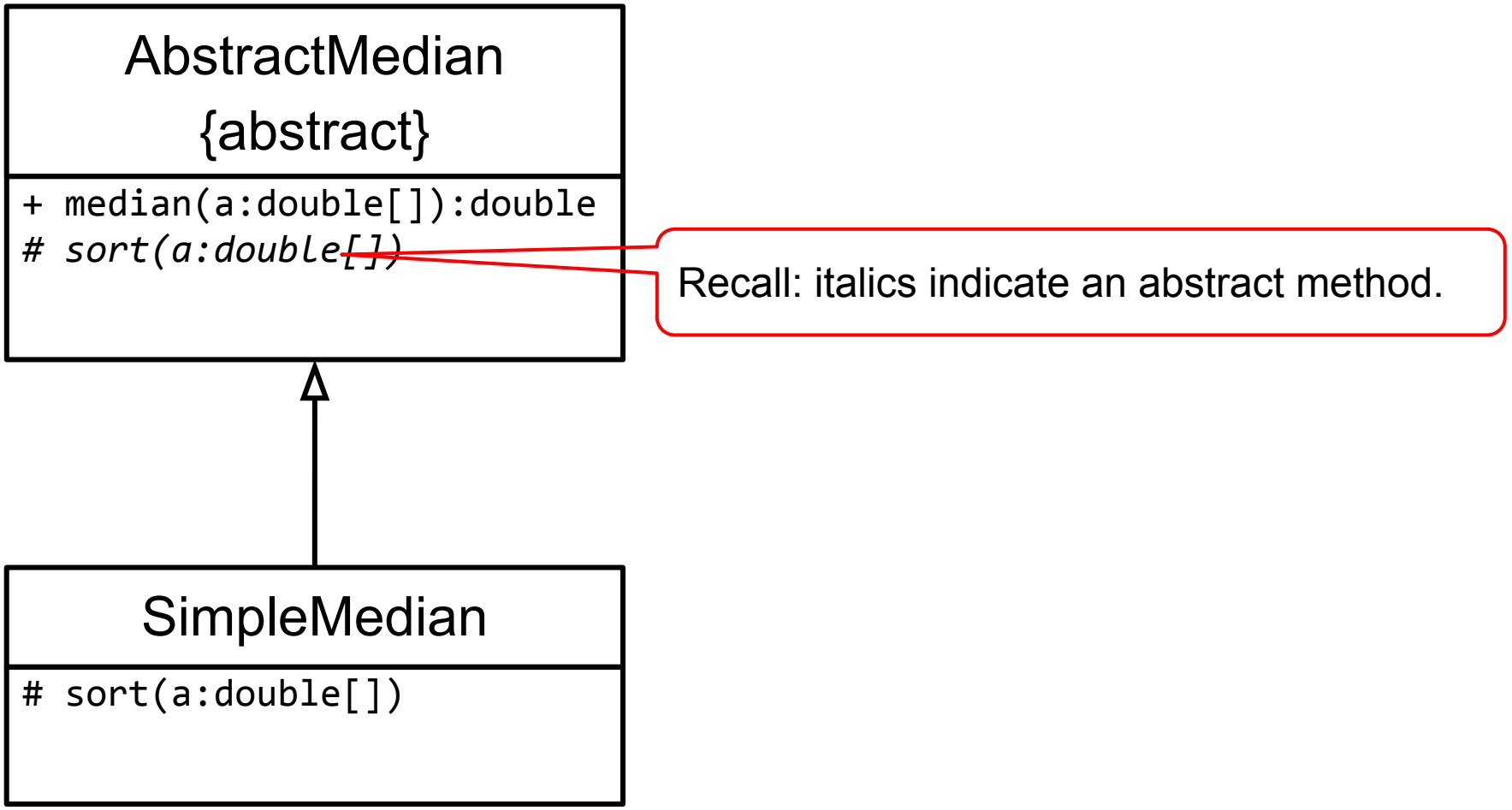
See code example (online)

- template
  - Abstract superclass with template method and abstract method *sort*.
  - Two subclasses with concrete implementations for the method *sort*.

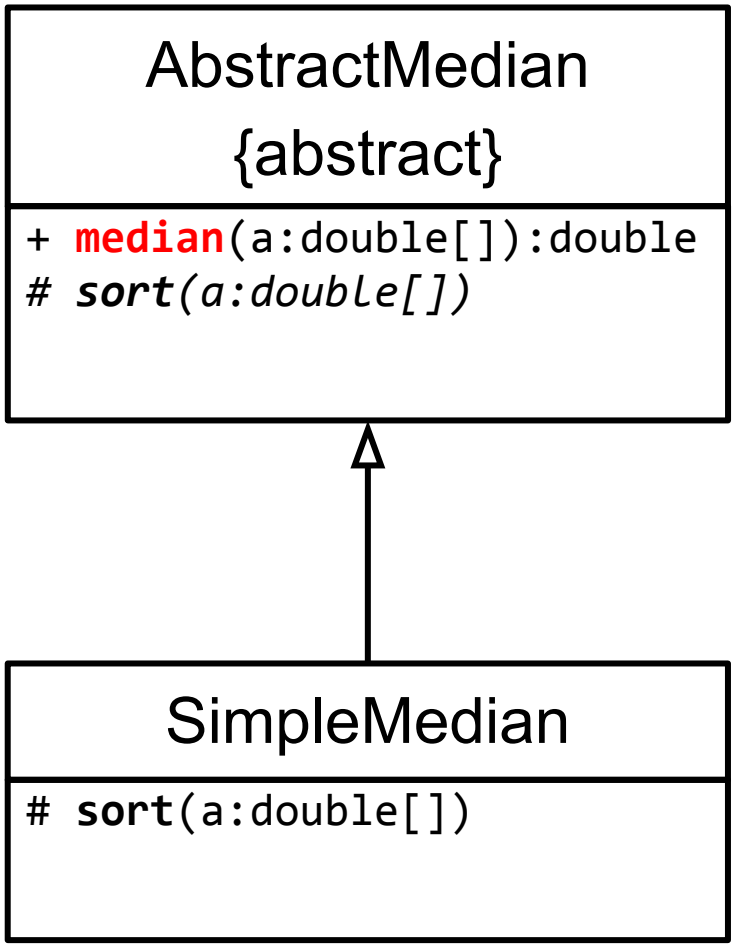
Use `ant` to compile and test the code:

```
$ant -p          => list all targets
$ant compile     => compile the code
$ant test        => run all tests (note the run-time differences between
                    SimpleMedianTest and QuickMedianTest!)
```

One possible solution: template method pattern



One possible solution: template method pattern



Should the median method be final?

- The template method (**median**) implements the algorithm but leaves the **sorting** of the array undefined.
- The concrete subclass only needs to implement the actual **sorting**.

Another possible solution: strategy pattern

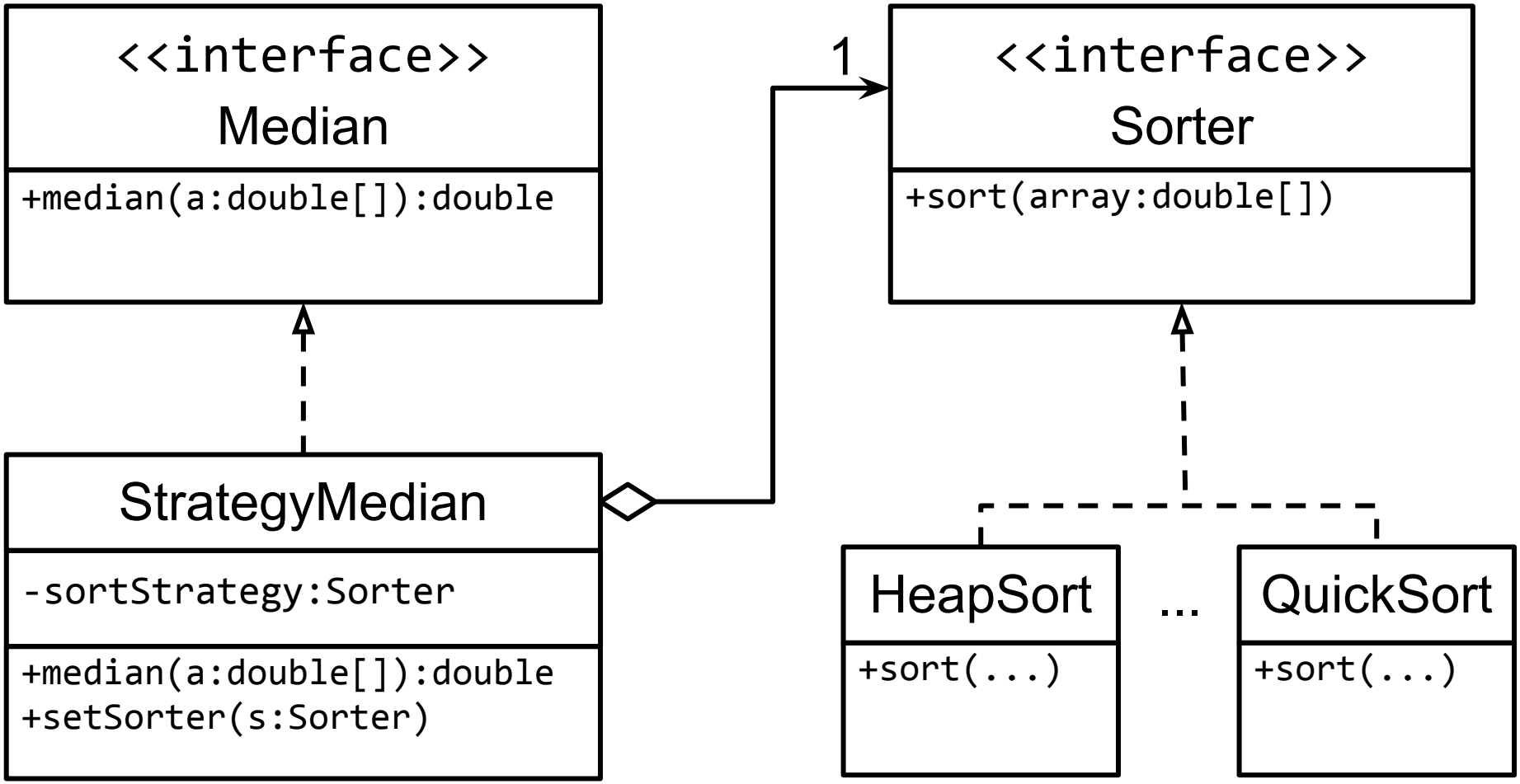
See code example (online)

- strategy
  - Interface *Sorter* for sorting strategies that defines the method *sort*.
  - Two implementations of this interface (*BubbleSort* and *QuickSort*).
  - *StrategyMedian* delegates the sorting to a sorting strategy, which can be configured and changed at run time.

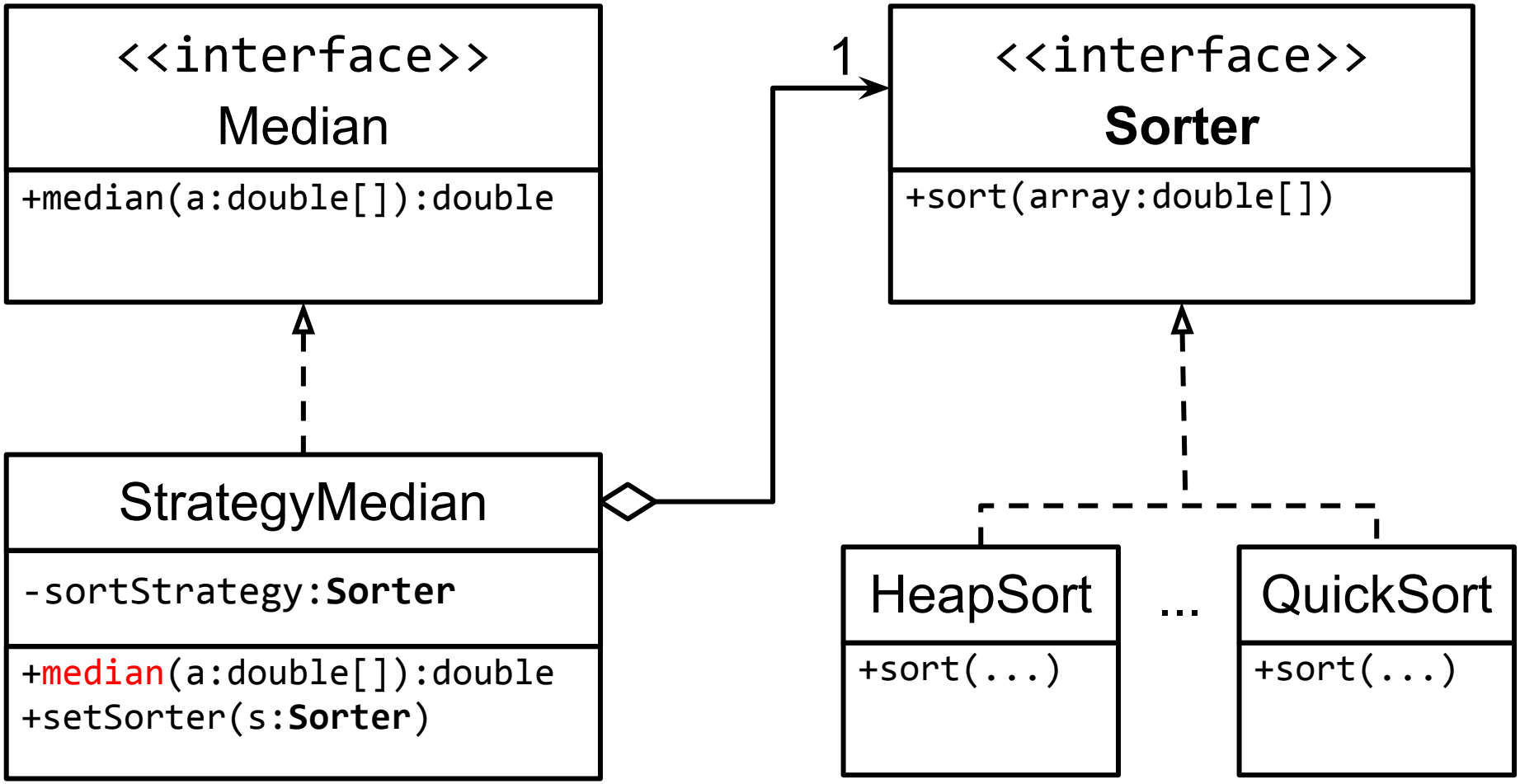
Use `ant` to compile and test the code:

```
$ant -p          => list all targets
$ant compile     => compile the code
$ant test        => run all tests (note how testSwapSorter in
                    StrategyMedianTest changes the sorter at run time!)
```

Another possible solution: strategy pattern



Another possible solution: strategy pattern



"median" delegates the sorting of the array to a "sortStrategy", which can be configured and changed at run time.

Template method pattern vs. strategy pattern

Two solutions to the same problem

Template method

- Behavior selected at compile time.
- Template method is usually final.

Strategy

- Behavior selected at runtime.
- Composition/aggregation over inheritance.