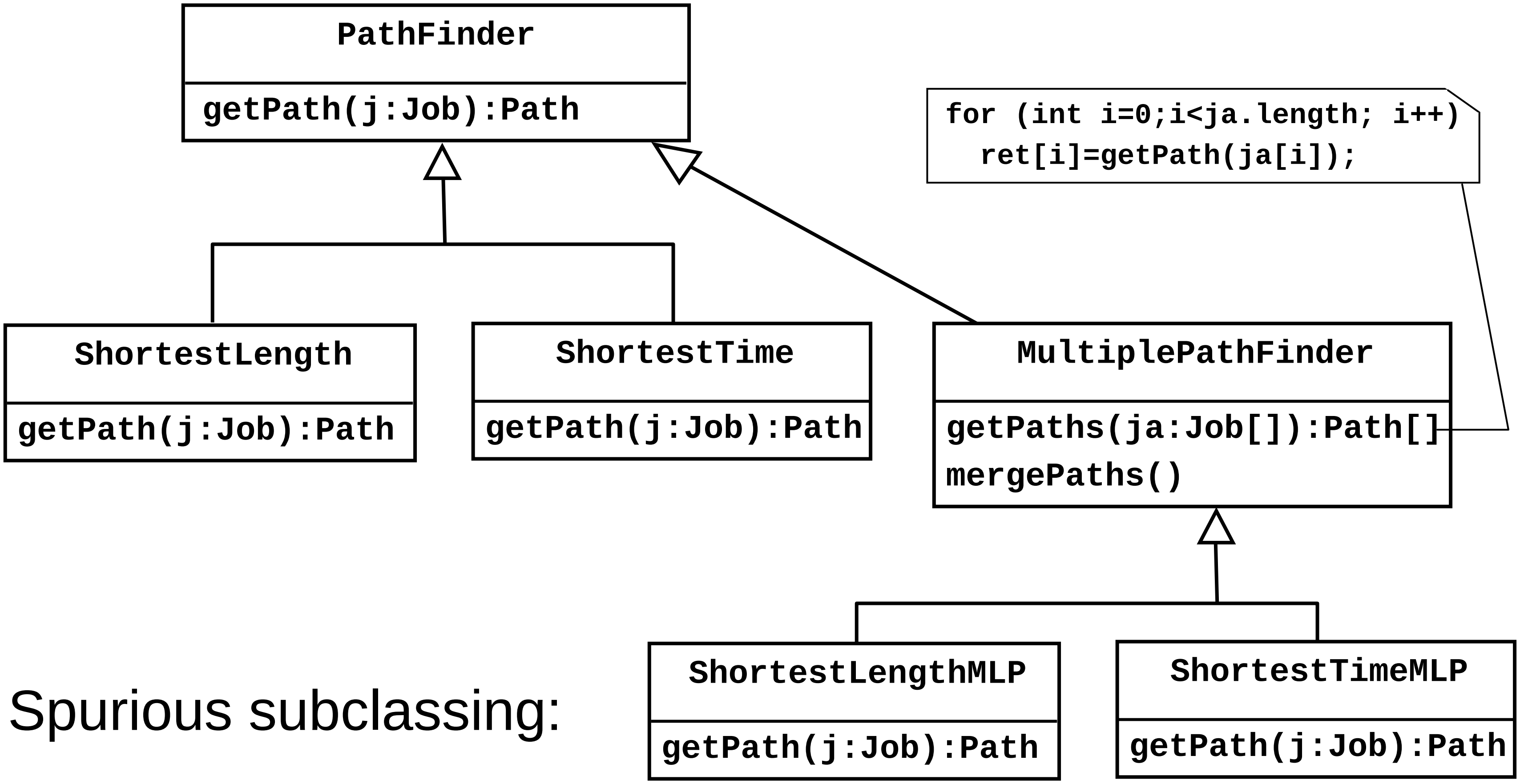
Design Patterns (III)

- Bridge
- Builder
- Observer

UNIVERSITÄT SALZBURG

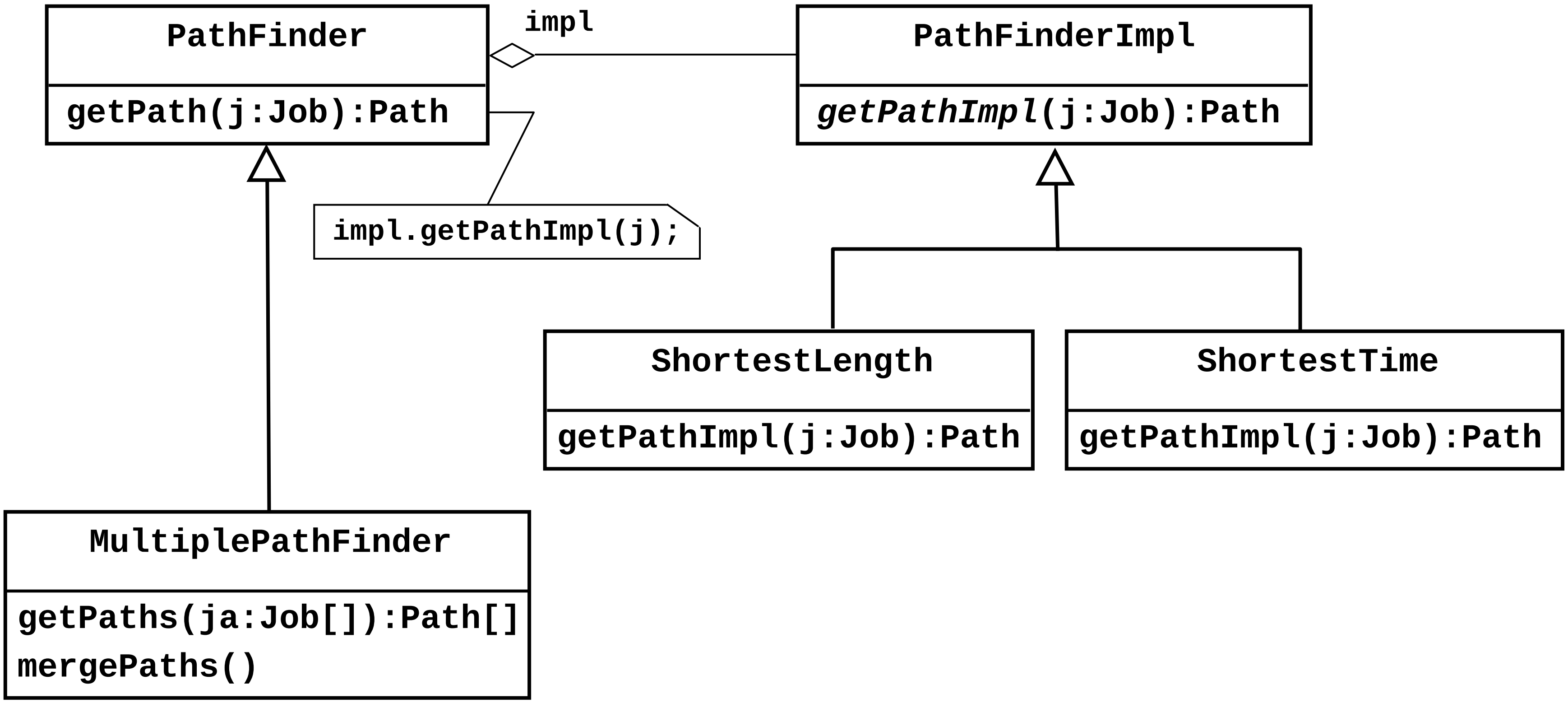
The Bridge Design Pattern: Example (I)



```
for (int i=0;i<ja.length; i++)
  ret[i]=getPath(ja[i]);
```

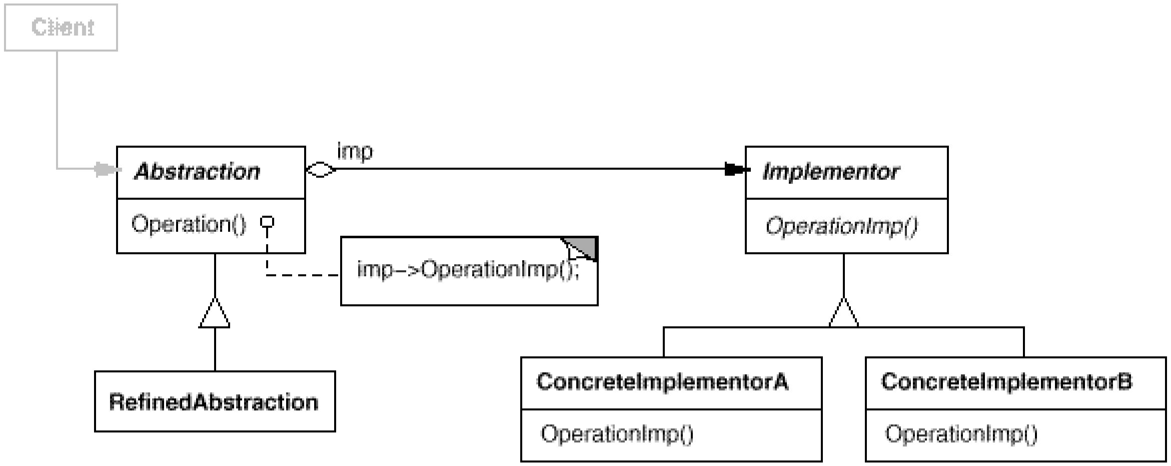
Spurious subclassing:

The Bridge Design Pattern: Example (II)



UNIVERSITÄT SALZBURG

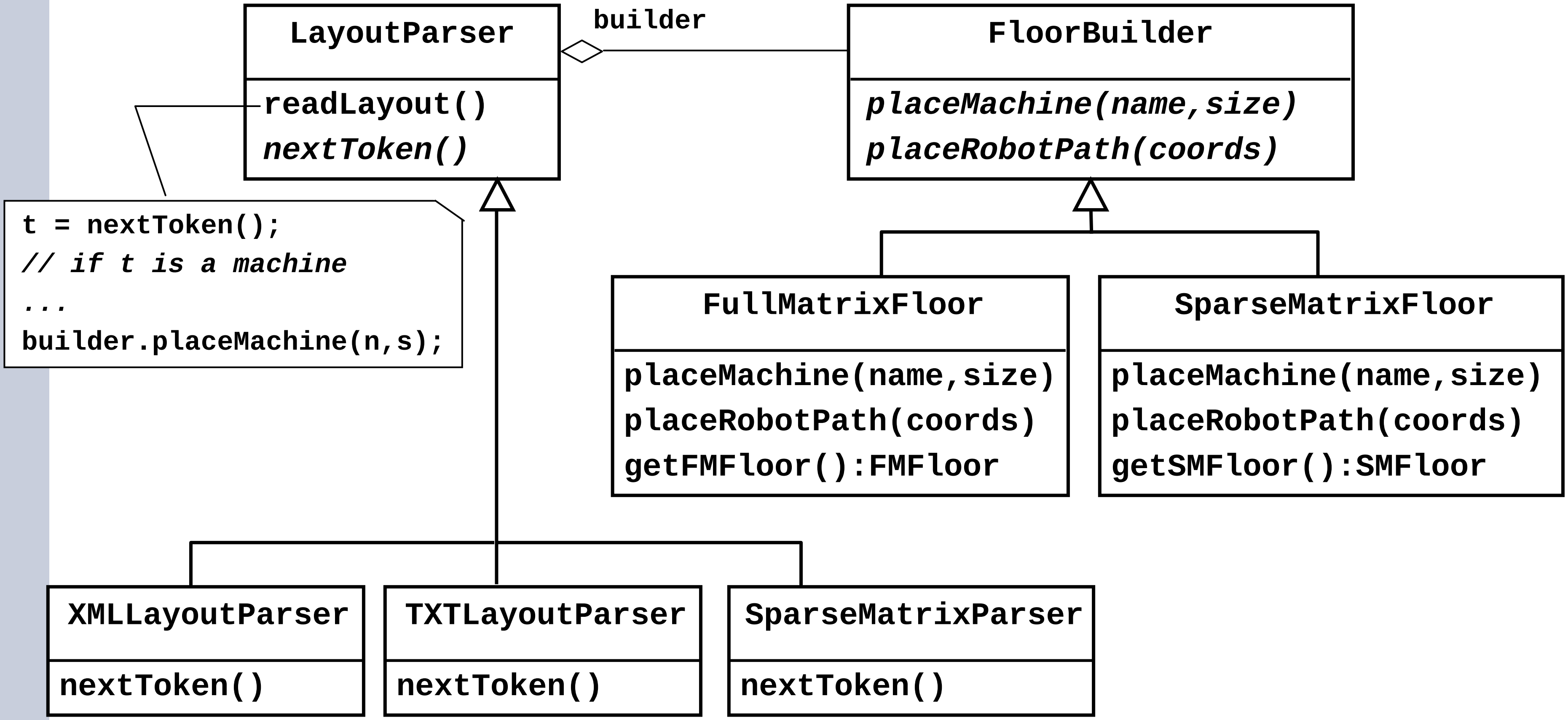
The Bridge Design Pattern: Structure



Used when:

- An abstraction should be decoupled from its implementation
- The abstraction and its implementation should be (independently) extensible by subclassing

The Builder Design Pattern: Example (I)



LayoutParser

readLayout()

nextToken()

builder

FloorBuilder

placeMachine(name,size)

placeRobotPath(coords)

```
t = nextToken();
// if t is a machine
...
builder.placeMachine(n,s);
```

FullMatrixFloor

placeMachine(name,size)

placeRobotPath(coords)

getFMFloor():FMFloor

SparseMatrixFloor

placeMachine(name,size)

placeRobotPath(coords)

getSMFloor():SMFloor

XMLLayoutParser

nextToken()

TXTLayoutParser

nextToken()

SparseMatrixParser

nextToken()

Hook Method and Hook Object with the same template method

The Builder Design Pattern: Example (II)

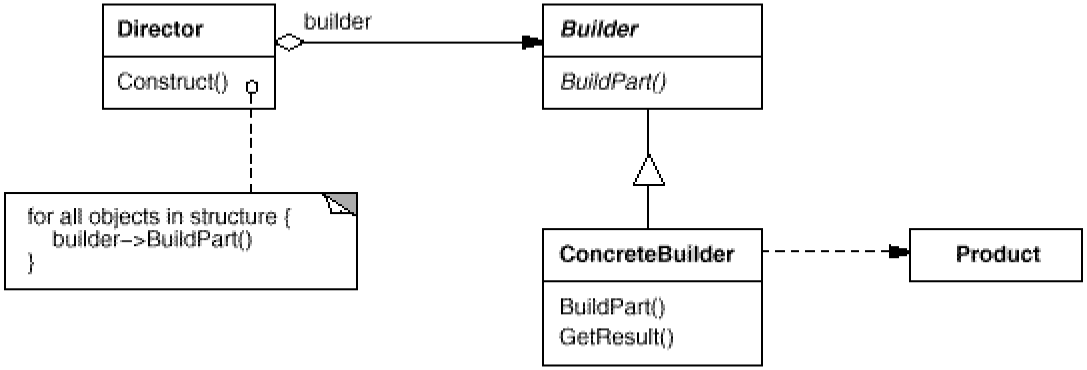
- Usage:

```
SMFloor sparse_floor;
FloorBuilder sm_builder = new SparseMatrixFloor();
LayoutParser layIn = new XMLLayoutParser(sm_builder, xml_file);
layIn.readLayout();
sparse_floor=sm_builder.getSMFloor();
```

- Change of built object at runtime:

```
FMFloor full_floor;
FloorBuilder fm_builder = new FullMatrixFloor();
LayoutParser layIn = new SparseMatrixParser(fm_builder,
                                                   sparse_floor);
layIn.readLayout();
full_floor = fm_builder.getFMFloor();
```

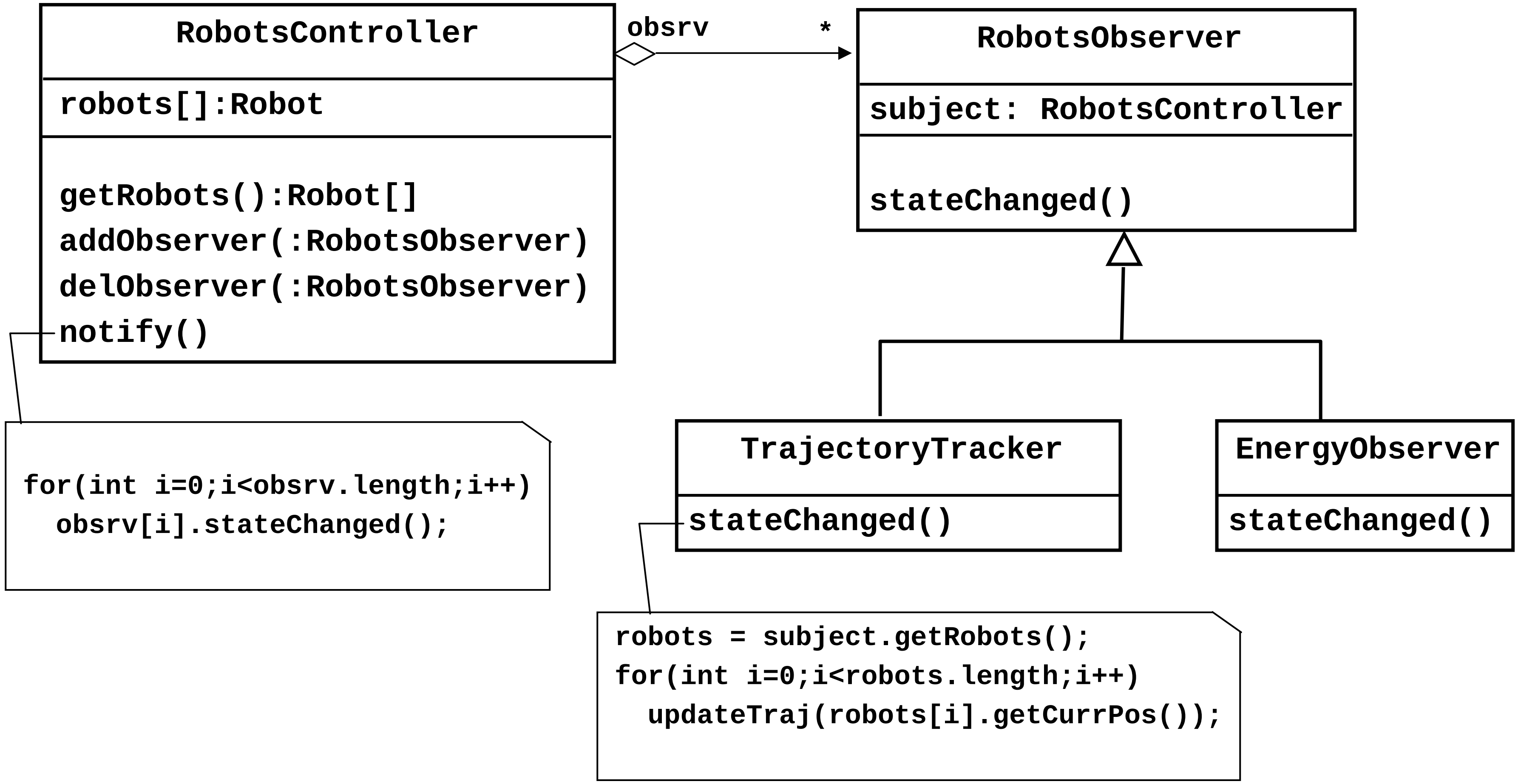
The Builder Design Pattern: Structure



Used when

- The construction of a product is independent of the parts
- Different representations for the product can be employed

The Observer Design Pattern: Example(I)



RobotsController

robots[]:Robot

getRobots():Robot[]
addObserver(:RobotsObserver)
delObserver(:RobotsObserver)
notify()

obsrv

*

RobotsObserver

subject: RobotsController

stateChanged()

TrajectoryTracker

stateChanged()

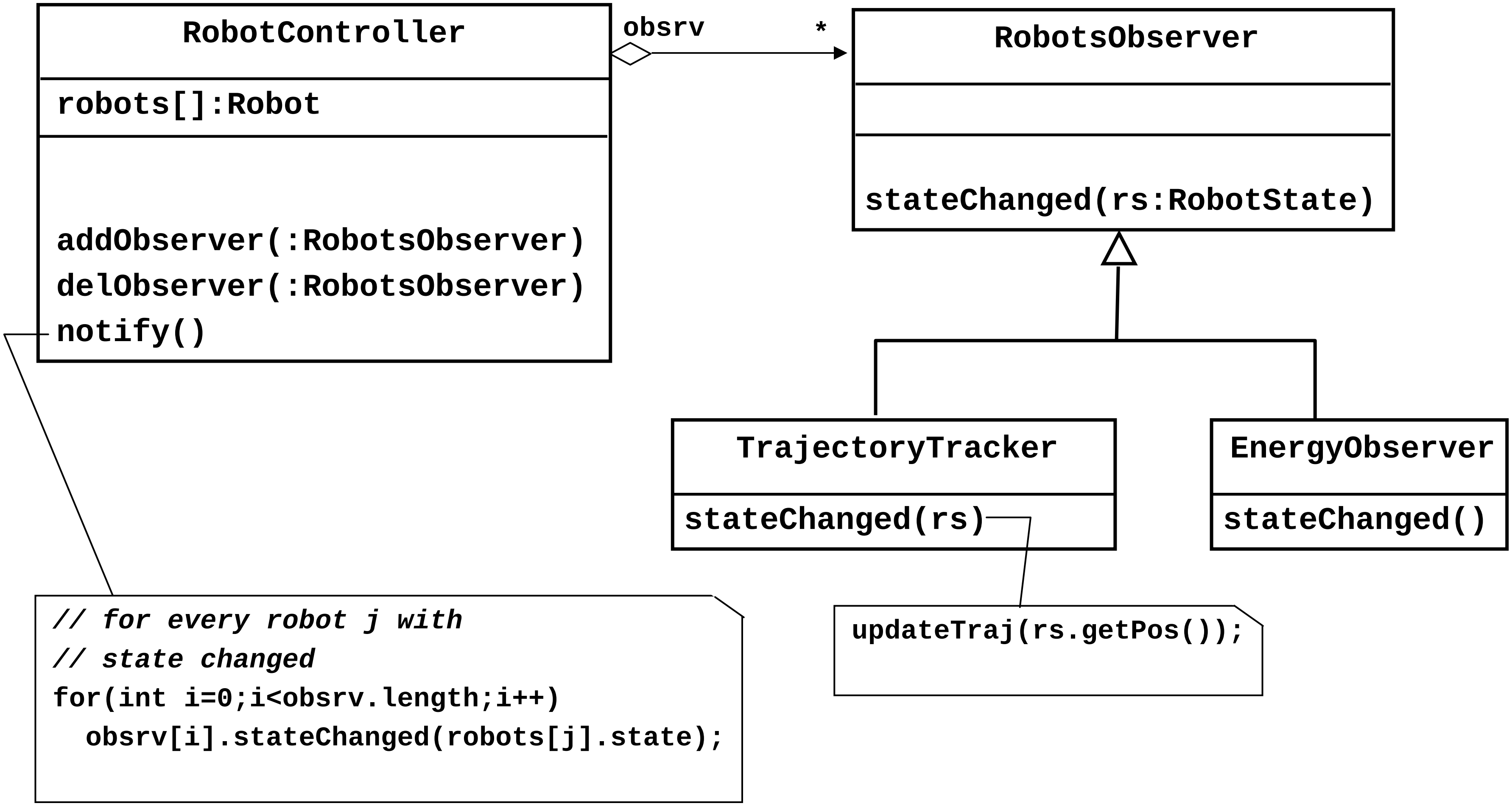
EnergyObserver

stateChanged()

```
for(int i=0;i<obsrv.length;i++)
  obsrv[i].stateChanged();
```

```
robots = subject.getRobots();
for(int i=0;i<robots.length;i++)
  updateTraj(robots[i].getCurrPos());
```

The Observer Design Pattern: Example(II)



RobotController

robots[]:Robot

addObserver(:RobotsObserver)

delObserver(:RobotsObserver)

notify()

obsrv

*

RobotsObserver

stateChanged(rs:RobotState)

TrajectoryTracker

stateChanged(rs)

EnergyObserver

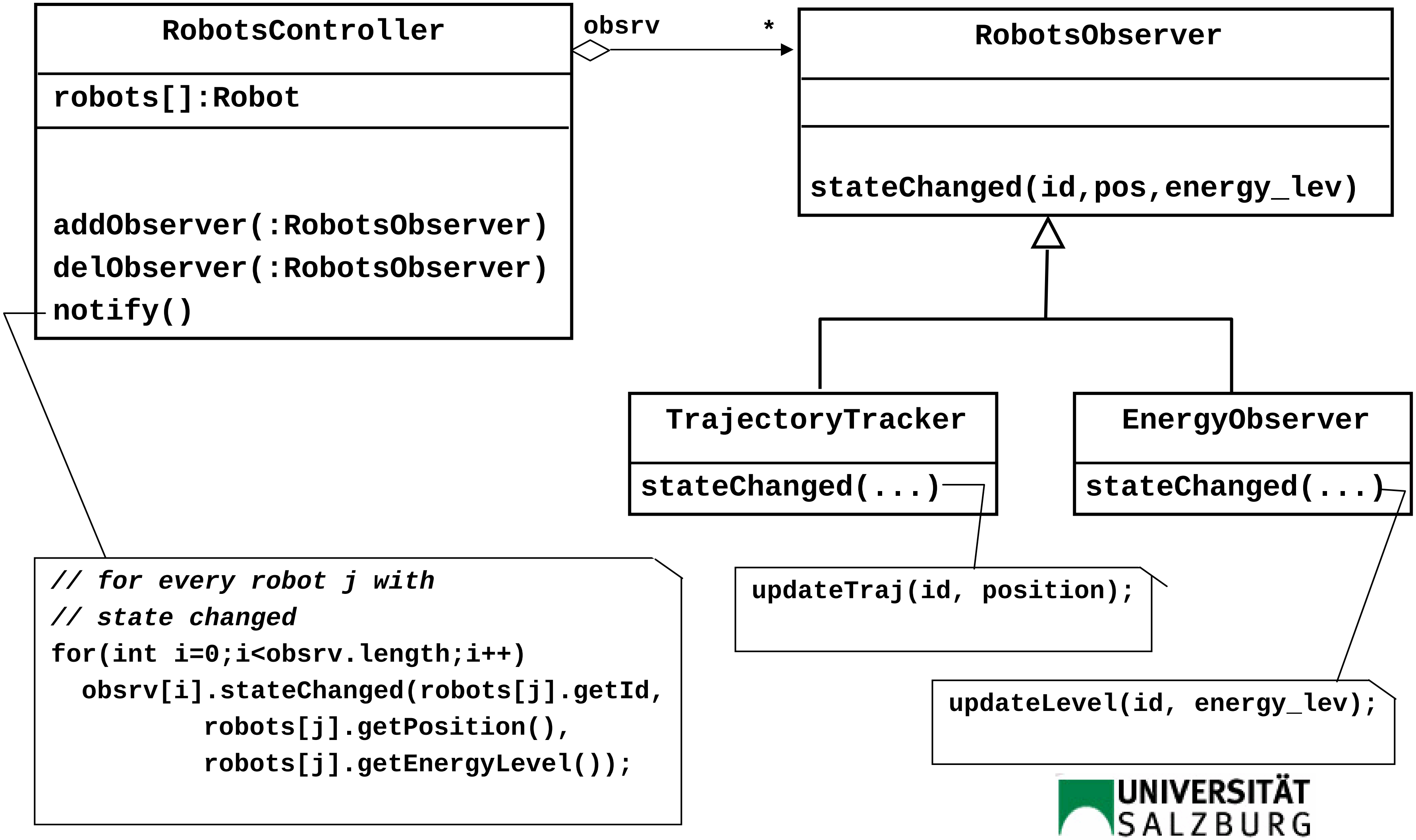
stateChanged()

```
// for every robot j with
// state changed
for(int i=0;i<obsrv.length;i++)
  obsrv[i].stateChanged(robots[j].state);
```

```
updateTraj(rs.getPos());
```

UNIVERSITÄT SALZBURG

The Observer Design Pattern:Example(III)



RobotsController

robots[]:Robot

addObserver(:RobotsObserver)
delObserver(:RobotsObserver)
notify()

obsrv *

RobotsObserver

stateChanged(id,pos,energy_lev)

TrajectoryTracker

stateChanged(...)

EnergyObserver

stateChanged(...)

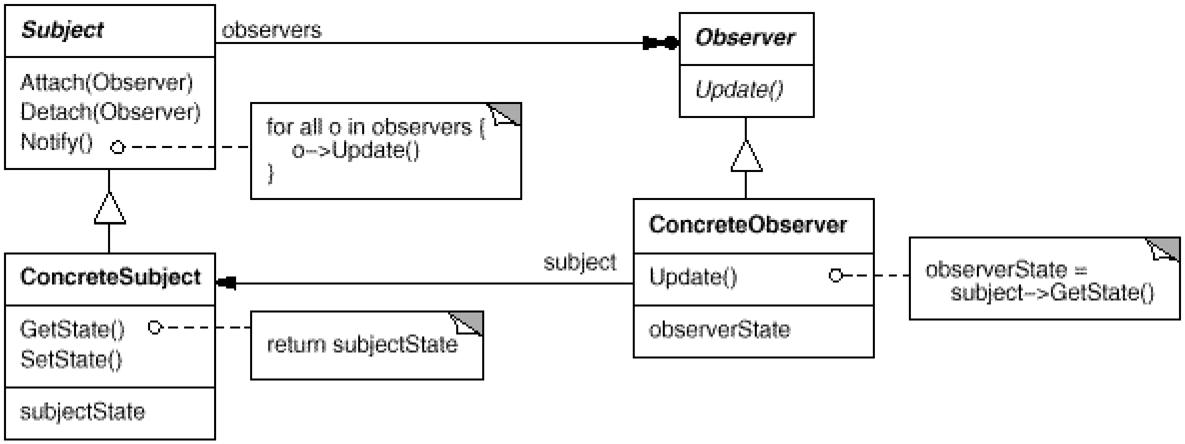
```
// for every robot j with
// state changed
for(int i=0;i<obsrv.length;i++)
  obsrv[i].stateChanged(robots[j].getId,
          robots[j].getPosition(),
          robots[j].getEnergyLevel());
```

```
updateTraj(id, position);
```

```
updateLevel(id, energy_lev);
```

UNIVERSITÄT
SALZBURG

The Observer Design Pattern: Structure



Use:

- To deal with different and dependent aspects of the same state
- When a change in an object must be announced to an unspecified number of objects
- When the receivers of the change must be decoupled