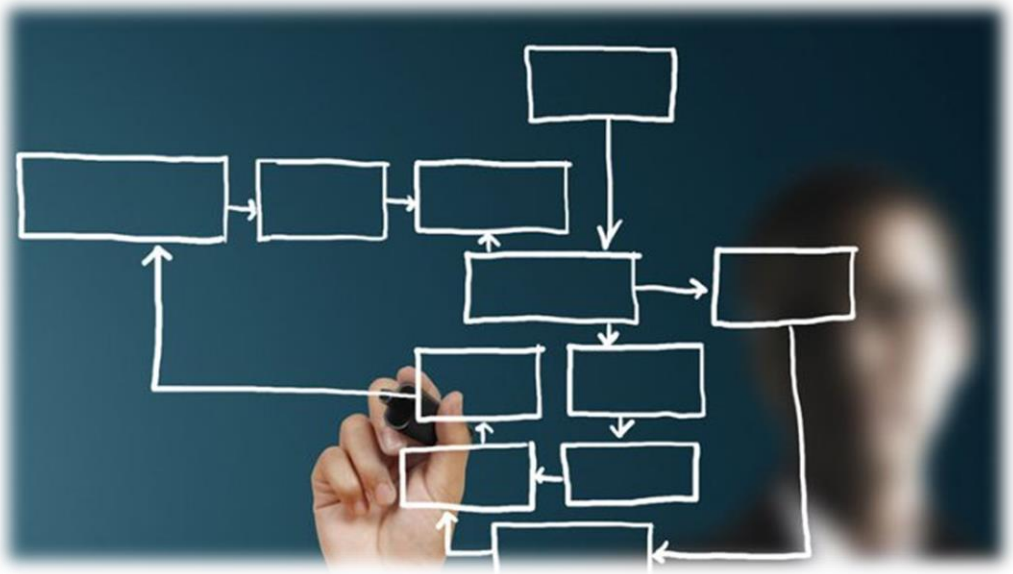
Advanced Programming 2

DR. ELIAHU KHALASTCHI

2016

Agenda

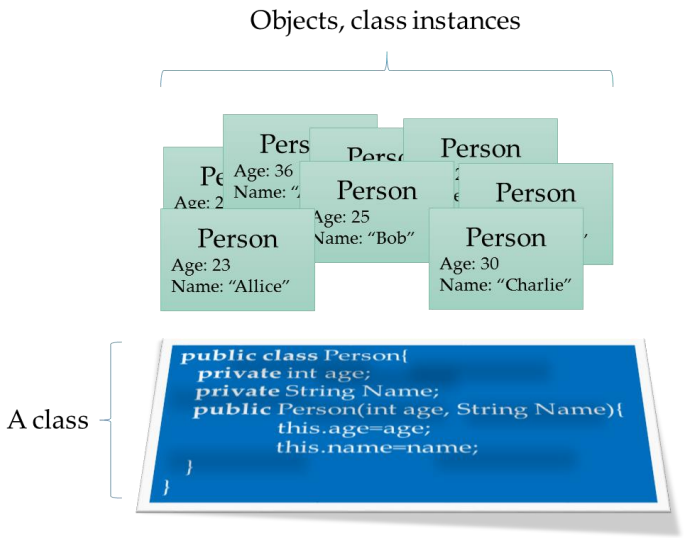
- Object Oriented Design Principles
- From pseudo code to OOP

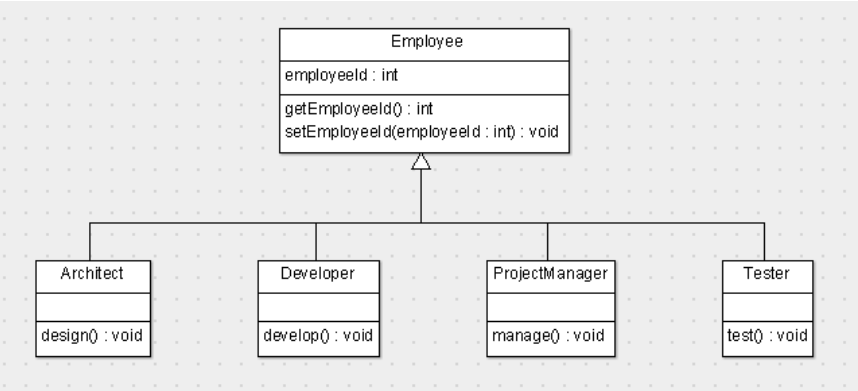


Object Oriented Design Principles

Design Principles

- Abstraction — Of types…
- Encapsulation — Of mechanisms… — Private members, Iterators, etc.
- Inheritance — Class hierarchy to prevent duplicated code…
- Polymorphism — Generalization of code (general algorithms & containers)





General Principles

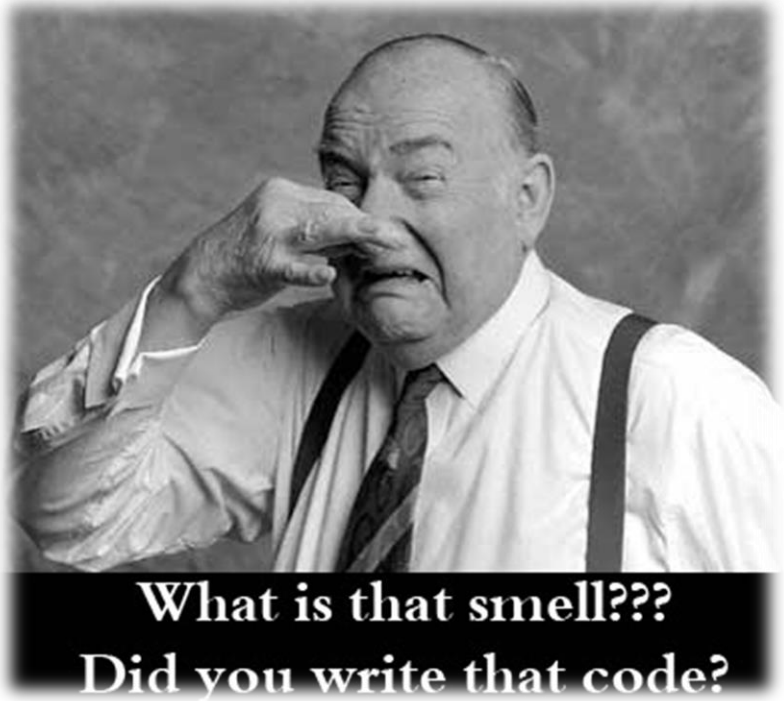
- DRY: don't repeat yourself
  - Obvious: we do not copy and paste blocks of code
  - Not obvious: also avoid duplications in
    - The data base schemers, diagrams and documentation…
  - "A single source of truth"

- YAGNI: you ain't gonna need it
  - Don't write speculative code, solve the problem you know to exist

- Avoid code smells…

Pointless comments
Inconsistent names
Excessively short / long identifiers
Same name, different meaning
Lack of comments
Excessive return of data
Too many none-public methods
Strikingly similar subclasses
Middle Man
Multiple inheritance
Data class
Conditional complexity
Downcasting
Long method
A Class with too many variables
Inappropriate intimacy
Feature Envy
Trivial modules or layers
Freeloader
The GOD class
Duplicated code
Contrived complexity
Too many parameters

Code Smells



YOU MUST AVOID THEM!

SOLID Principles

BY ROBERT MARTIN, A.K.A “UNCLE BOB”

Solid principles of OO design

- S
  - Single Responsibility Principle
- O
  - Open / Closed Principle
- L
  - Liskov Substitution Principle
- I
  - Interface Segregation Principle
- D
  - Dependency Inversion Principle

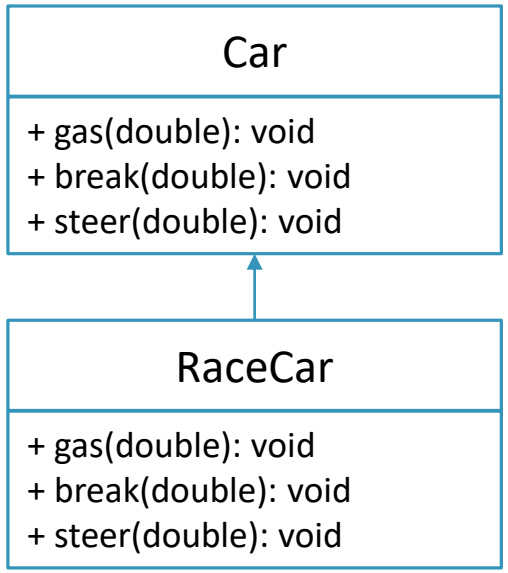
Solid principles of OO design

- S
  - **Single Responsibility Principle**
- O
  - Open / Closed Principle
- L
  - Liskov Substitution Principle
- I
  - Interface Segregation Principle
- D
  - Dependency Inversion Principle

Car
+ gas(double): void + break(double): void + steer(double): void + planPath(Destination):Plan

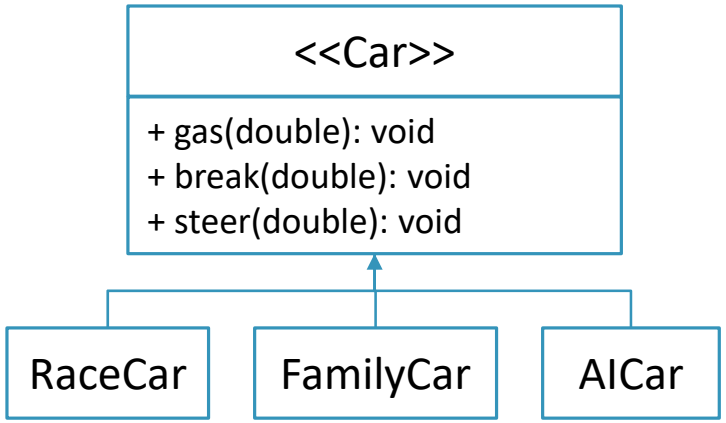
Solid principles of OO design

- S
  - Single Responsibility Principle
- O
  - **Open / Closed Principle**
- L
  - Liskov Substitution Principle
- I
  - Interface Segregation Principle
- D
  - Dependency Inversion Principle



Solid principles of OO design

- S
  - Single Responsibility Principle
- O
  - Open / Closed Principle
- L
  - **Liskov Substitution Principle**
- I
  - Interface Segregation Principle
- D
  - Dependency Inversion Principle

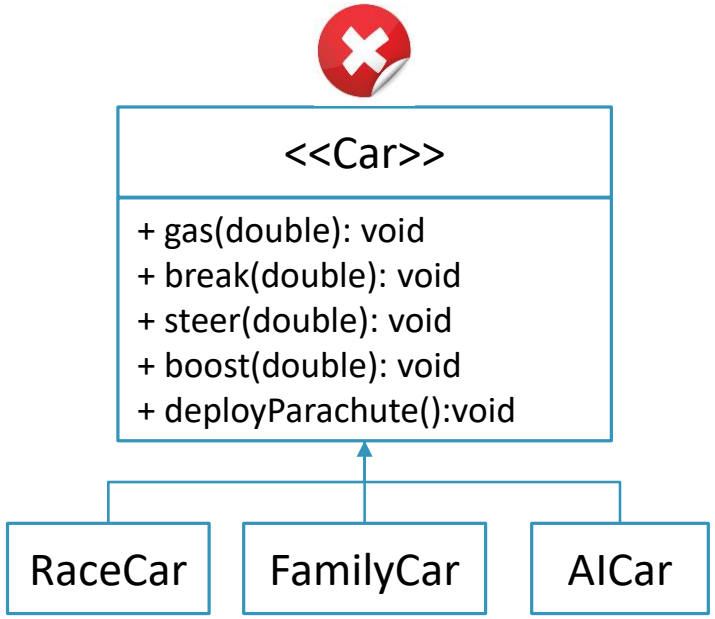



```
void drive(Car c){
        …
        c.gas(40);
        …
        c.break(20);
        c.steer(25);
        …
}
```

```
void drive(Car c){
        …
        c.gas(40);
        …
        ((AICar)c) .autopilot();
        …
}
```

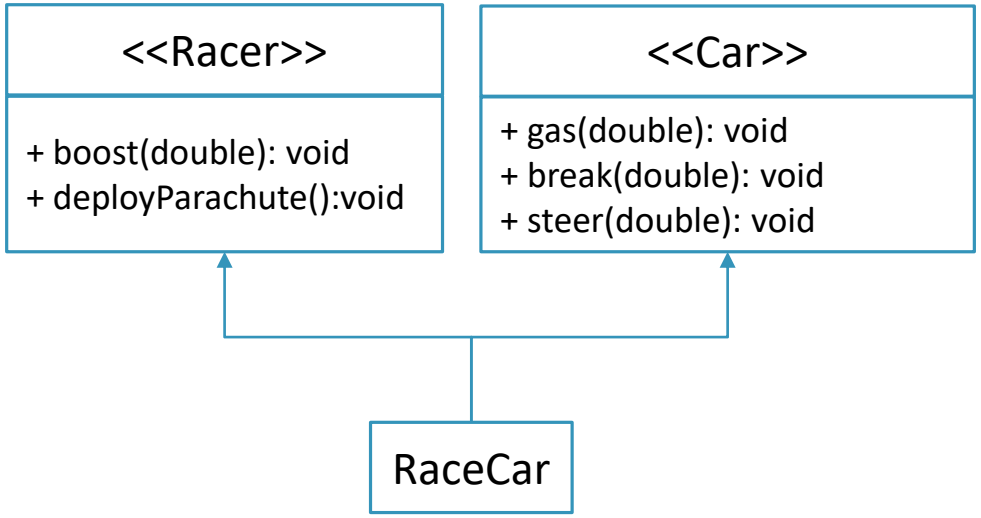
Solid principles of OO design

- S
  - Single Responsibility Principle
- O
  - Open / Closed Principle
- L
  - Liskov Substitution Principle
- I
  - **Interface Segregation Principle**
- D
  - Dependency Inversion Principle



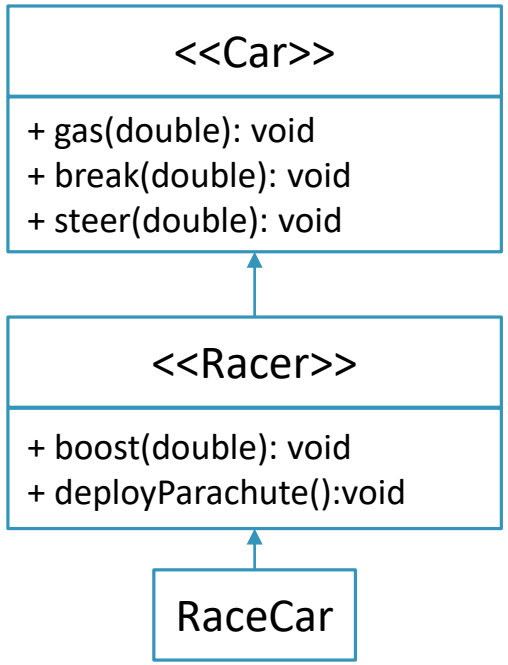
Solid principles of OO design

- S
  - Single Responsibility Principle
- O
  - Open / Closed Principle
- L
  - Liskov Substitution Principle
- I
  - **Interface Segregation Principle**
- D
  - Dependency Inversion Principle



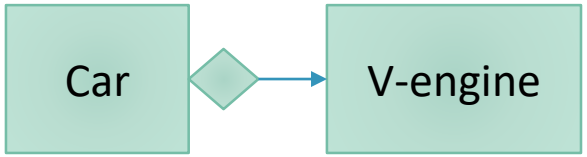
Solid principles of OO design

- S
  - Single Responsibility Principle
- O
  - Open / Closed Principle
- L
  - Liskov Substitution Principle
- I
  - **Interface Segregation Principle**
- D
  - Dependency Inversion Principle



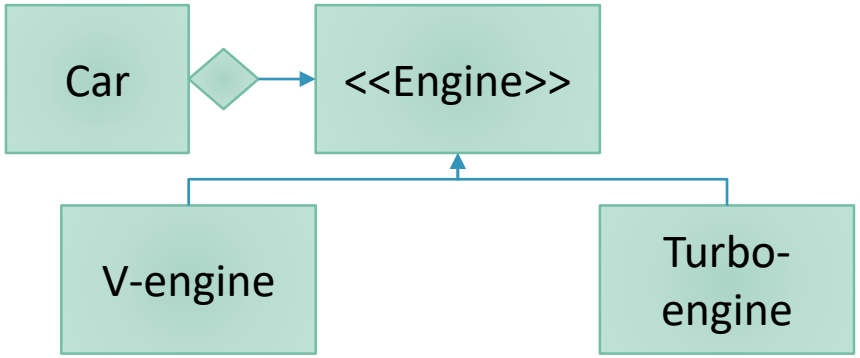
Solid principles of OO design

- S
  - Single Responsibility Principle
- O
  - Open / Closed Principle
- L
  - Liskov Substitution Principle
- I
  - Interface Segregation Principle
- D
  - **Dependency Inversion Principle**



Solid principles of OO design

- S
  - Single Responsibility Principle
- O
  - Open / Closed Principle
- L
  - Liskov Substitution Principle
- I
  - Interface Segregation Principle
- D
  - **Dependency Inversion Principle**



GRASP Principles

GRASP

- **G**eneral **R**esponsibility **A**ssignment **S**oftware **P**atterns
- SOLID and GRASP are not in conflict
  - GRASP puts the focus on responsibility

- There are 9 ideas in GRASP:
  - Creator, Controller, Pure Fabrication
  - Information Expert, High Cohesion, Indirection
  - Low Coupling, Polymorphism, Protected Variations

Information Expert

- Assign a responsibility to the class that has the information to fulfill it

- Which class should calculate the number of unread emails?

User

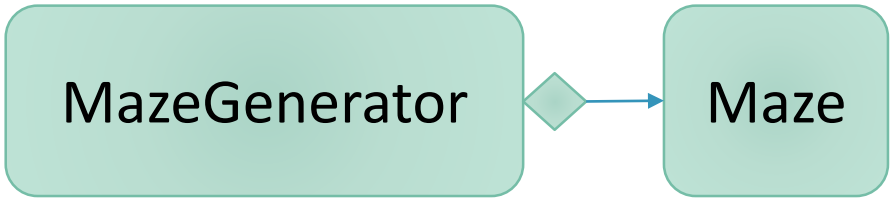
Mailbox

Email

Creator

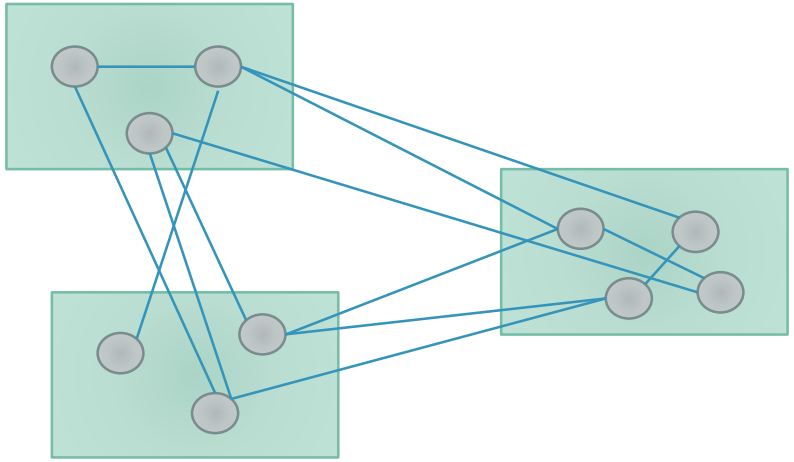
- Who is responsible for creating an object?

- To assign a creator we need to answer these questions:
  - Does the creator contain another object?
  - Does the creator closely use another object?
  - Does the creator know enough to create an object?

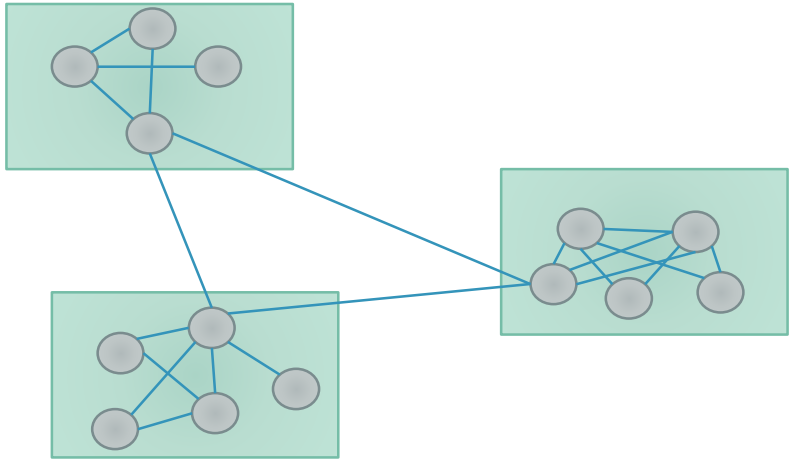


Low Coupling / High Cohesion

- **Coupling:** the level of dependencies between objects
- **Decoupling:** the process of reducing these dependencies
- **Cohesion:** how focused is a class around a single responsibility
- We want high cohesion & low coupling



Low Cohesion, High Coupling



High Cohesion, Low Coupling

Controller

- Example: Decouple *UI* class from a Business class

- MVC – Model View Controller
- Is an example of using this idea as an architecture

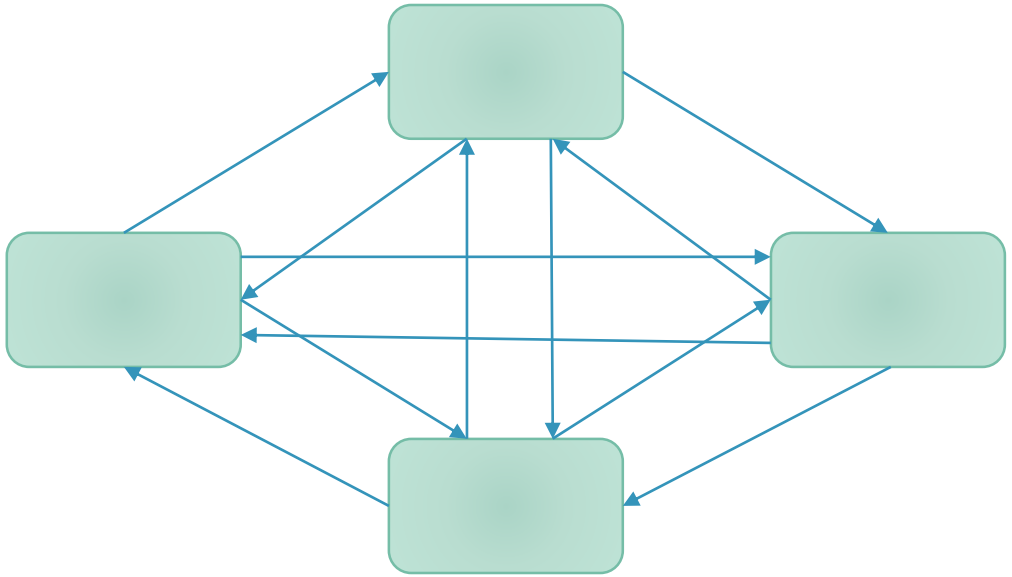
Pure Fabrication

- If a behavior does not belong anywhere - Put it in a new class
- Instead of forcing it to another class
  - And thus reduce its cohesion

- It is OK to have a class that represents pure functionality
  - As long as you know why you are doing it

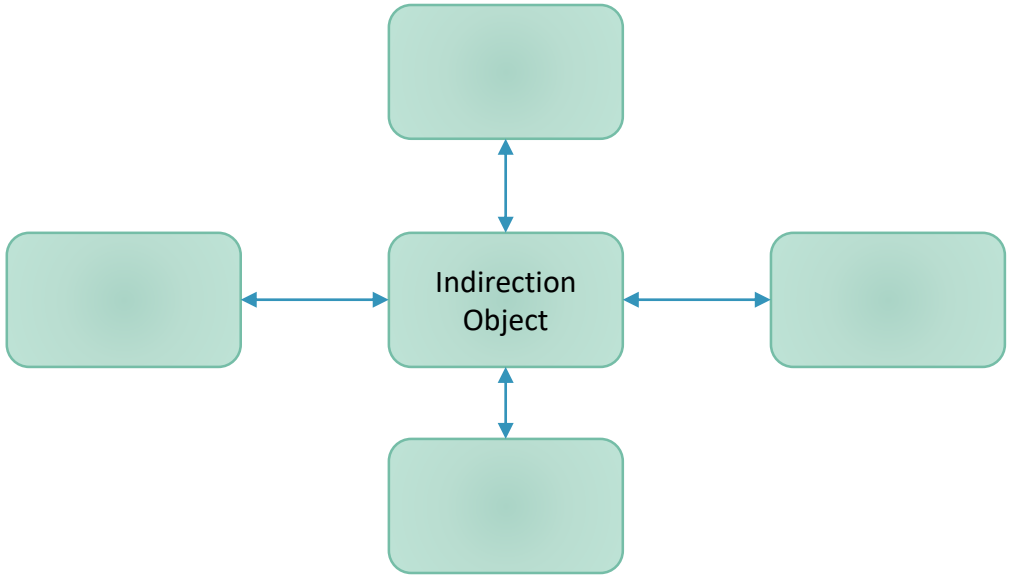
Indirection

- To reduce coupling, introduce an intermediate object



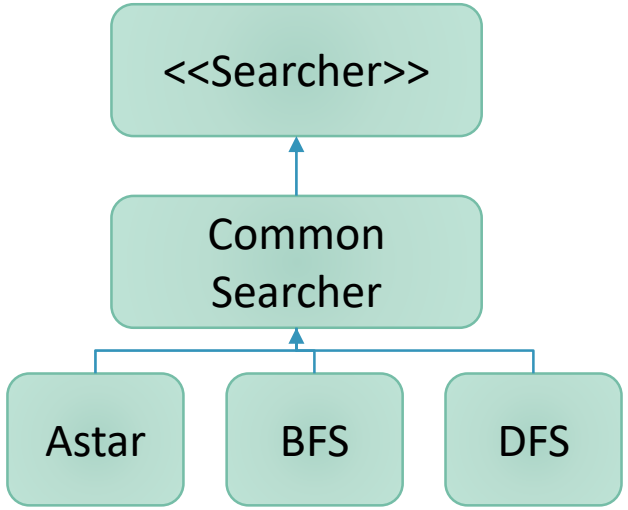
Indirection

- To reduce coupling, introduce an intermediate object



Polymorphism

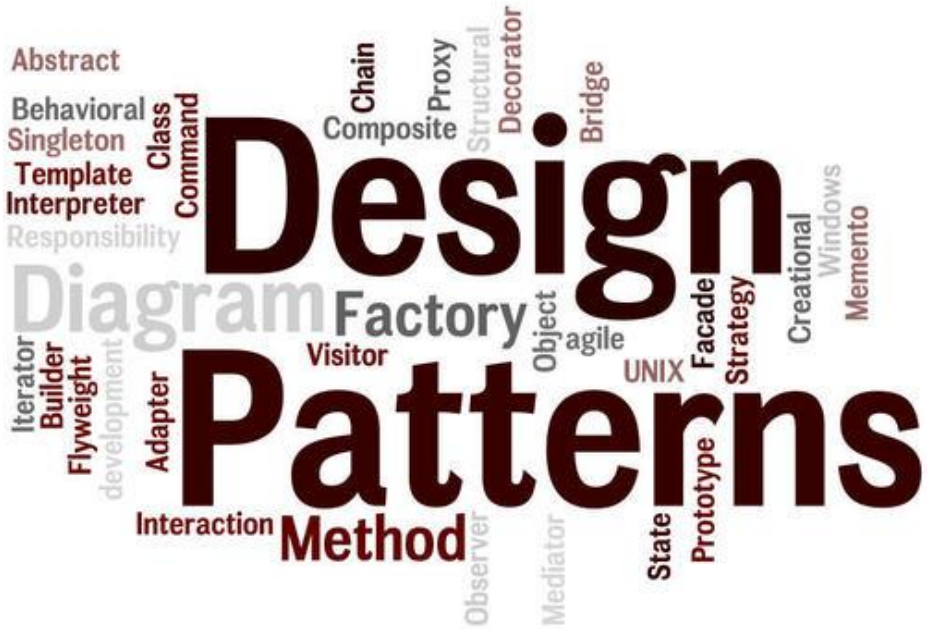
- Automatically correct behavior based on type
- As opposed to typing
  - Checking the type of a runtime-object
- Example:
  - Searcher s = new Astar(); // or…
  - Searcher s = new BFS();
  - The rest of the code is unchanged
  - It applies the methods of Searcher



Protected Variations

- Protect the project from changes and variations
- Identify the most likely points of change
- And use what we have learned
  - Encapsulation
  - Interfaces
  - Polymorphism / Liskov substitution principle
  - Open / closed principle
  - Etc.

- These OO principles allow us to write *readable, maintainable,* and *flexible* code.

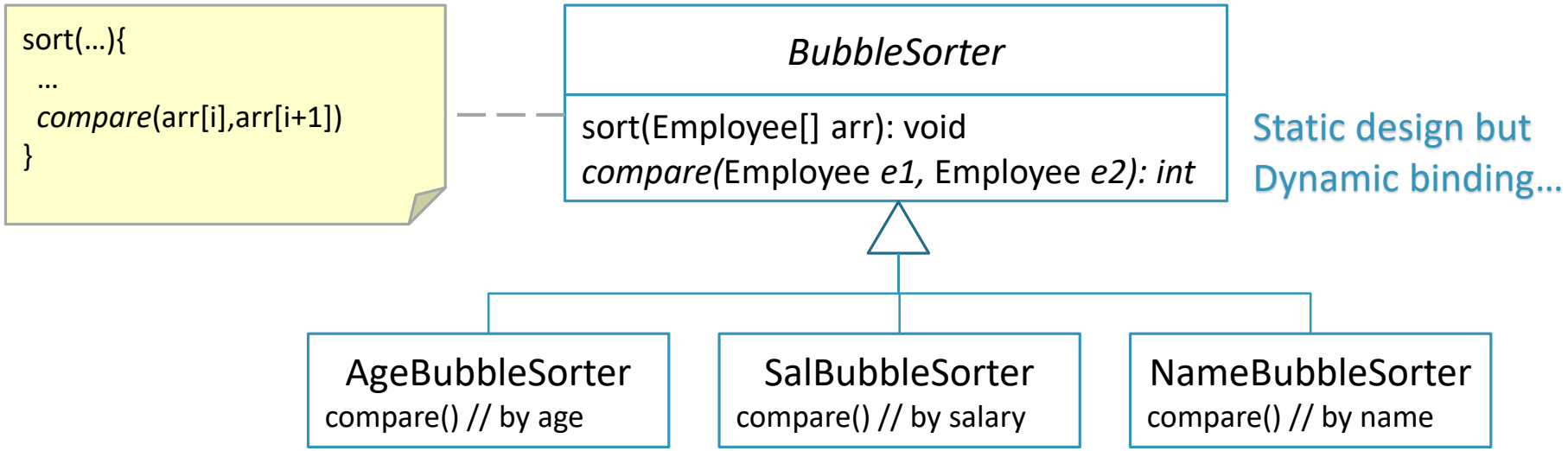


The Design Patterns you have learned help you to achieve these design principles and avoid code smells…

Static vs. Dynamic Design

AN EXAMPLE

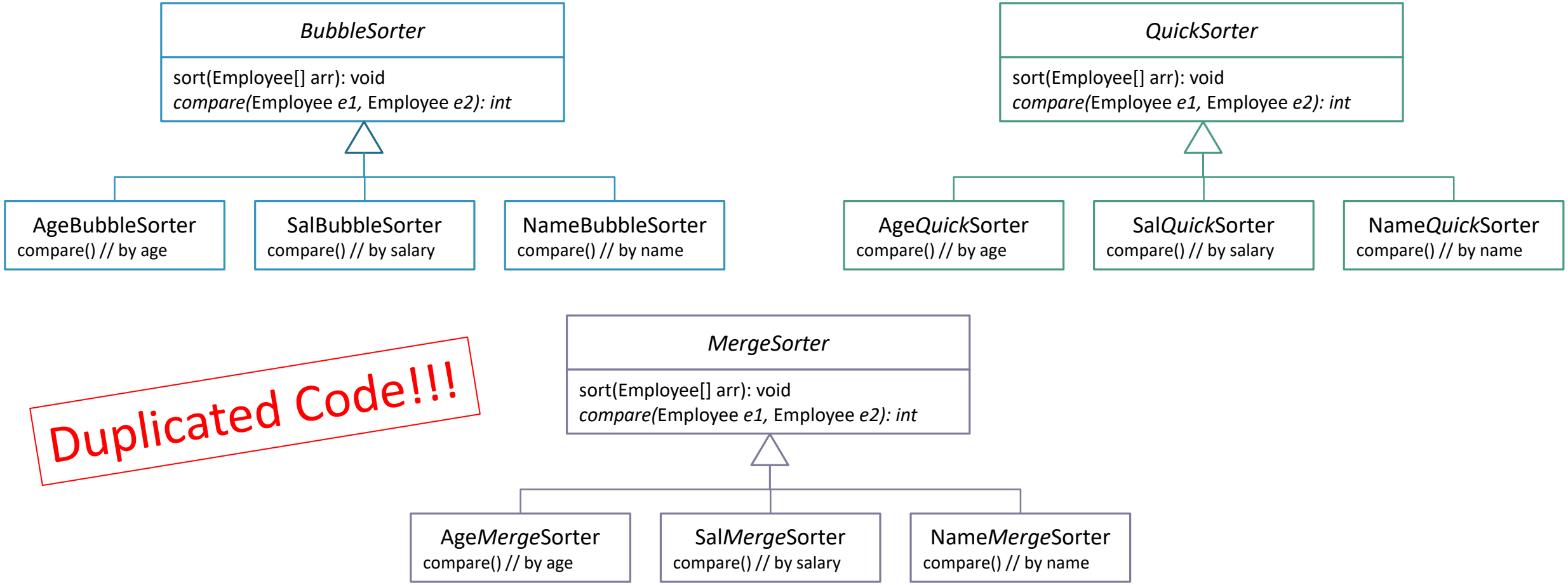
Static Design – uses inheritance




```
BubbleSorter s = new AgeBubbleSorter();
s.sort(employees);
```

```
BubbleSorter s = new NameBubbleSorter();
s.sort(employees);
```

Static Design – problem…



Static Design – problem…

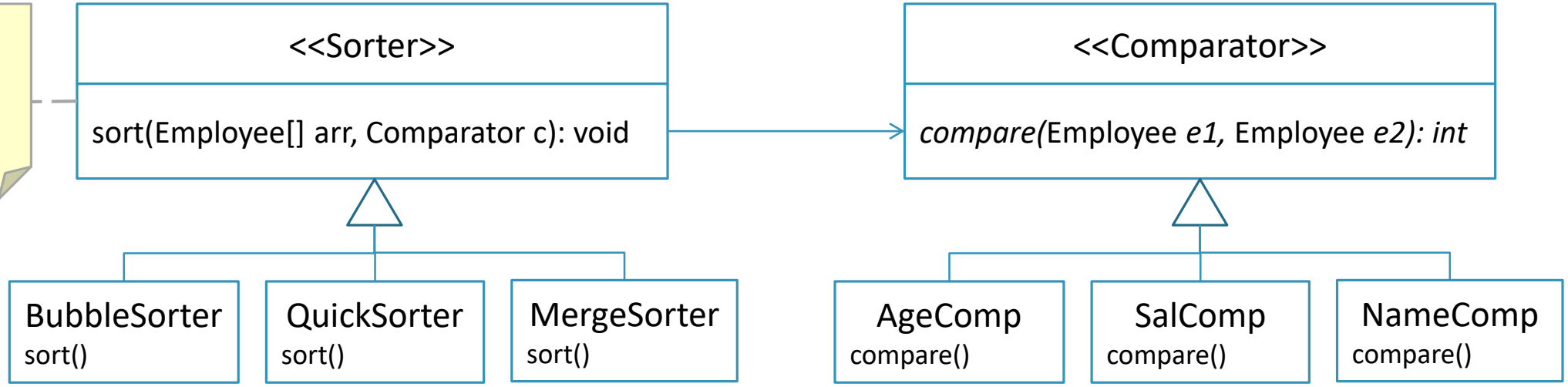
```
class AgeBubbleSorter extends BubbleSorter{
    int compare(Employee e1, Employee e2){
        return e1.Age – e2.Age;
    }
}
```

```
class AgeQuickSorter extends QuickSorter{
    int compare(Employee e1, Employee e2){
        return e1.Age – e2.Age;
    }
}
```

Duplicated Code!!!

Dynamic Design – uses composition

```
sort(…){
  …
  c.compare(arr[i],arr[i+1])
}
```

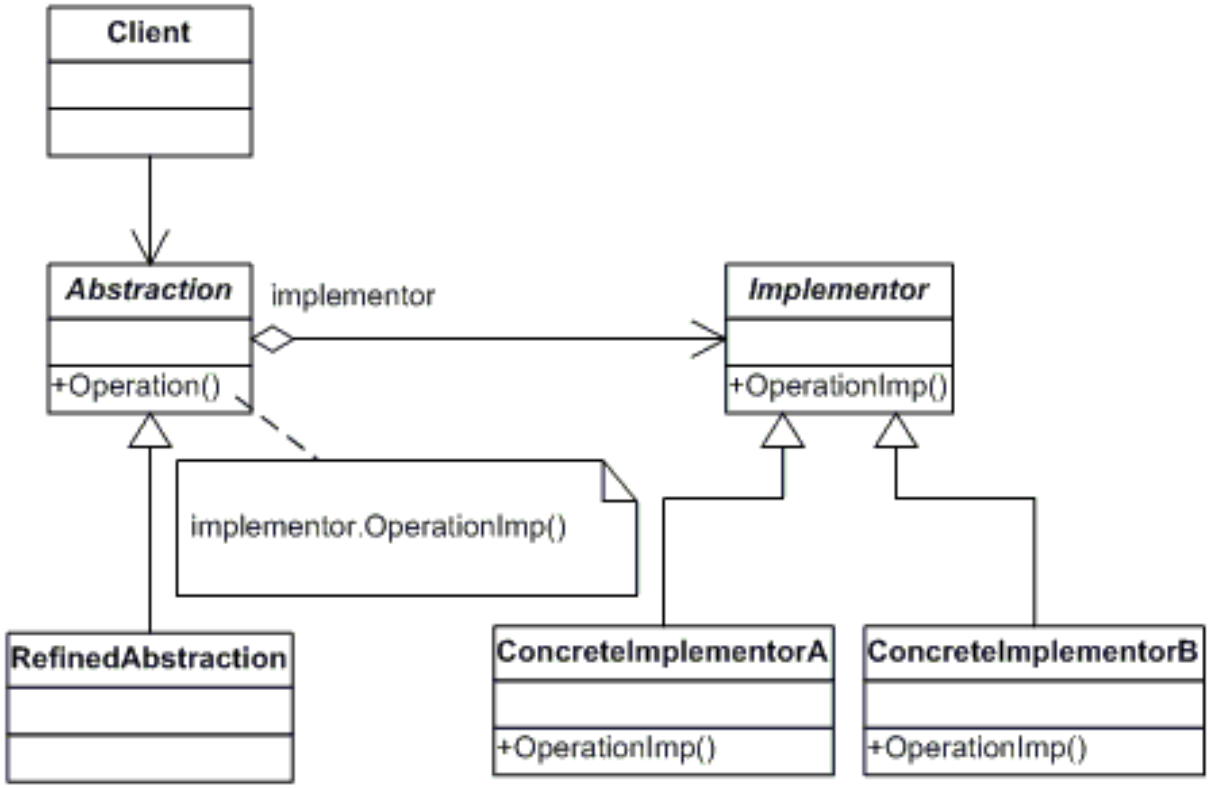


- ☑ S
- ☑ O
- ☑ L
- ☑ I
- ☑ D

```
Sorter s = new BubbleSorter();
s.sort(employees, new AgeComp());
s.sort(employees, new SalComp());
```

```
Sorter s = new QuickSorter();
s.sort(employees, new AgeComp());
s.sort(employees, new SalComp());
```

The Bridge Design Pattern



From pseudo-code to OOP

Let’s say we want to solve these problems

- We refer to each problem setting as a domain

Unscramble Domain:

LI #oCe v

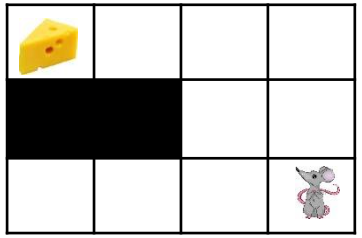
I Love C#

- A state represents the current string
- The cost to switch two letters can be derived from their distance
- Which switches do we need?

Parking Lot Domain:

- A state represents current car positions
- Each car may have a different “move” value
- How can we get the black car out?

Maze Domain:



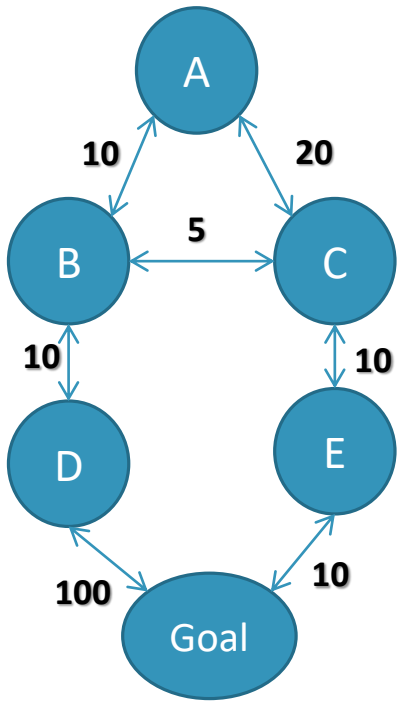
- A state may represent the position of the mouse
- The cost to move diagonally can be 15
- The cost to move directly can be 10
- What is the chipset path?

We can use the **Best First Search** algorithm

Best First Search:

OPEN = [initial state] // a **priority queue** of states to be evaluated
CLOSED = [] // a **set** of states already evaluated
while OPEN is not empty
do

1. **n** ← dequeue(OPEN) // Remove the best node from OPEN
2. add(n,CLOSED) // so we won't check n again
3. If **n** is the goal state,
   backtrace path to **n** (through recorded parents) and return path.
4. Create **n**'s successors.
5. For each successor **s** do:
   a. If **s** is not in CLOSED and **s** is not in OPEN:
      update that we came to **s** from **n**
      add(**s**,OPEN)
   b. Otherwise, if this new path is better than previous one
      i. If it is not in OPEN add it to OPEN.
      ii. Otherwise, adjust its priority in OPEN done



OPEN

A {0,null}

B {10,A}

~~C {20,B}~~

D {20,B}

E {25,C}

G {35,E}

CLOSED

Goal ← E ← C ← B ← A

How can we do it in the OOP way?

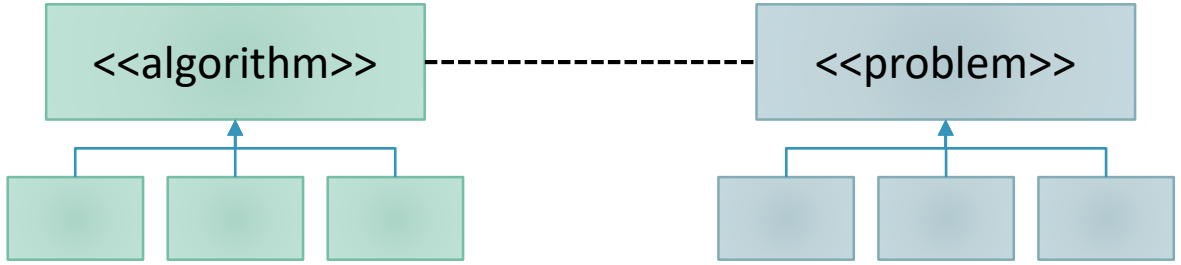
We **do not** just implement it inside one class…

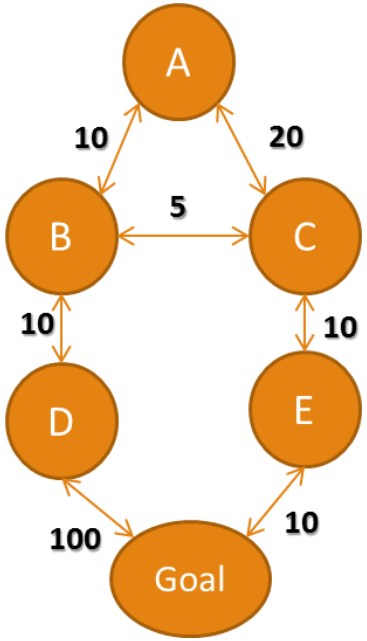
- We need to think about the **design!!!**
- We need to think how this algorithm is going to be used
  - In different projects
  - For different problems

- Where to start?
- It is probably a good idea to suppurate items which are
  - Domain Specific - specific to a certain domain e.g., the cost to move the mouse
  - Domain Independent - not specific to a certain domain
- We want to create a **domain independent** implementation of the BFS algorithm

Decouple the algorithm from the problem it solves!

- We need to start by defining what we expect from
  - A general search problem
  - A general search algorithm
- This expectation is the functionality of these entities
- In other words, we should define their interfaces
- Later, each entity can have different independent implementations





Defining the Problem

We can start with a general **State** class

```
public class State
{
    private string state;    // the state represented by a string
    private double cost;     // cost to reach this state (set by a setter)
    private State cameFrom;  // the state we came from to this state (setter)

    public State(string state)    // CTOR
    {
        this.state = state;
    }

    public override bool Equals(object obj) // we override Object's Equals method
    {
        return state.Equals((obj as State).state);
    }
     // ...
}
```

Example of Use:

```
State a, b, goal;
a = new State("A");
b = new State("B");
goal=new State("B");

Console.WriteLine(b.Equals(goal));
// true
```

We can start with a general **State** class

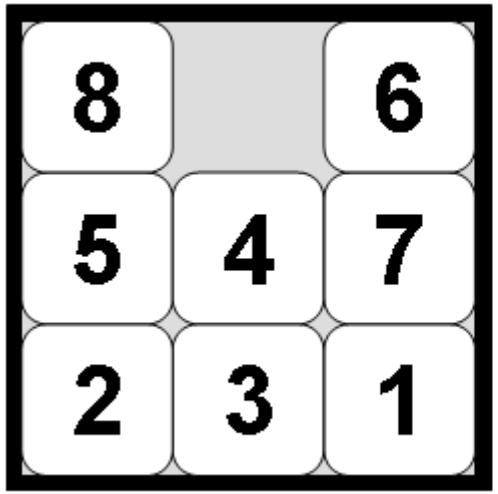
```
public class State<T>
{
    private T state;          // the state represented by a string
    private double cost;      // cost to reach this state (set by a setter)
    private State<T> cameFrom;  // the state we came from to this state (setter)

    public State(T state)     // CTOR
    {
        this.state = state;
    }

    public bool Equals(State<T> s) // we overload Object's Equals method
    {
        return state.Equals(s.state);
    }
     // ...
}
```

Example of Use:

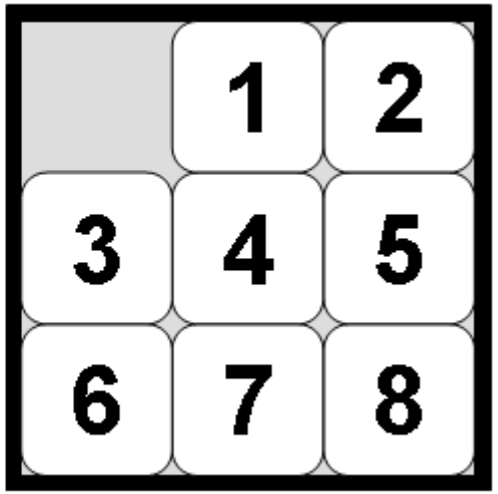
```
State<string> a, b, goal;
a = new State<string>("A");
b = new State<string>("B");
goal=new State<string>("B");

Console.WriteLine(b.Equals(goal));
// true
```

Example of Representing the state as a string

string state="8 6547231";



string state=" 12345678";



What do we expect from a search problem?

Best First Search:

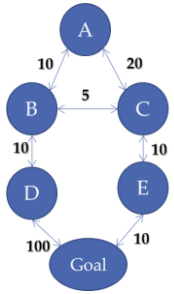
```
OPEN = [initial state]   // a priority queue of states to be evaluated
CLOSED = []        // a set of states already evaluated
while OPEN is not empty
do
  1. n ← dequeue(OPEN) // Remove the best node from OPEN
  2. add(n,CLOSED)          // so we won't check n again
  3. If n is the goal state,
       backtrace path to n (through recorded parents) and return path.
  4. Create n's successors.
  5. For each successor s do:
       a. If s is not in CLOSED and s is not in OPEN:
              i.  update that we came to s from n
              ii. add(s,OPEN)
       b. Otherwise, if this new path is better than previous one
                     i.  If it is not in OPEN add it to OPEN.
              ii. Otherwise, adjust its priority in OPEN
done
```

What do we need from the search domain?

- To give us the ***start state*** and the ***goal state***
- Given a state, what are the possible states we can get to?
  - This is actually a part of the graph…

```
public interface ISearchable {
     State getInitialState();
     State getGoalState();
     List<State> getAllPossibleStates(State s);
}
```

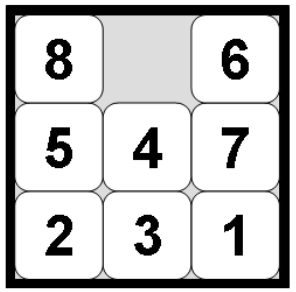
State



EightPuzzle

getAllPossibleStates of

"8 6547231"



Returns:

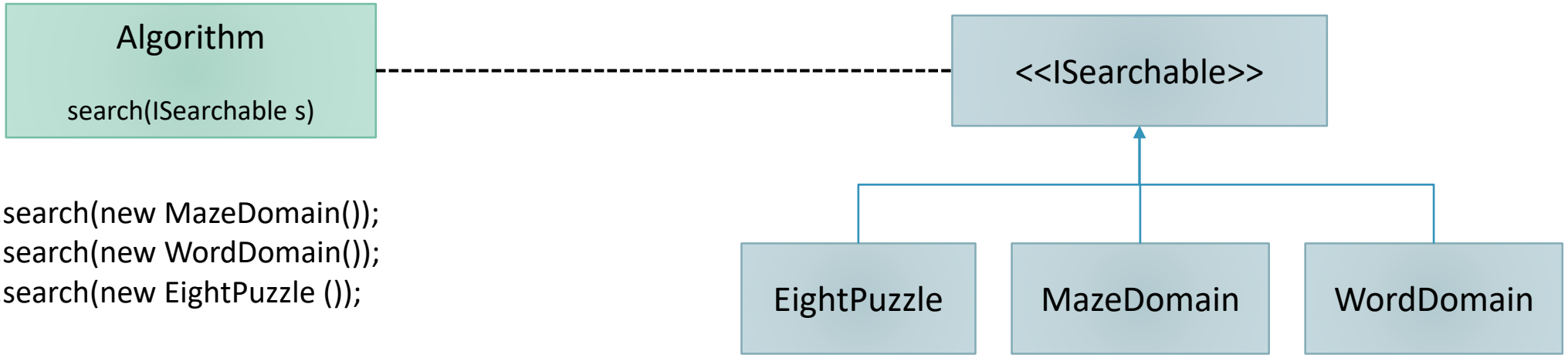
" 86547231"

"8465 7231"

"86 547231"

What did we gain?

- The algorithm just knows ISearchable (and State)
- It does not know any specific searchable domains (and states)
- ➔ we can switch searchable domains independently from the algorithm!!!




```
alg.search(new MazeDomain());
alg.search(new WordDomain());
alg.search(new EightPuzzle ());
```

We call this design ***dependency injection***

```
Best First Search:
OPEN = [initial state]        // a priority queue of states to be evaluated
CLOSED = []        // a set of states already evaluated
while OPEN is not empty
do
  1. n ← dequeue(OPEN) // Remove the best node from OPEN
  2. add(n,CLOSED)              // so we won't check n again
  3. If n is the goal state,
       backtrace path to n (through recorded parents) and return path.
  4. Create n's successors.
  5. For each successor s do:
          a. If s is not in CLOSED and s is not in OPEN:
                  i.  update that we came to s from n
                  ii. add(s,OPEN)
          b. Otherwise, if this new path is better than previous one
                  i.  If it is not in OPEN add it to OPEN.
                  ii. Otherwise, adjust its priority in OPEN
done
```

Defining the Algorithm

THE SEARCH ALGORITHM

By now, we are able to switch searchable domains independently from the searching algorithm.

We want to be able to switch searching algorithms as well.

What more do we need to think about?

- The BFS is a **type of** "searcher"
- We may want to implement **other** searching algorithms in the **future** as well
  - E.g., beam search, A*, hill climbing, etc.
- We want our system to work with any type of "searcher"
  - I.e., we can replace the searching algorithm without changing the system's code
- For that we need to define an Interface!

```
public interface ISearcher
{
    // the search method
     Solution search (ISearchable searchable);
    // get how many nodes were evaluated by the algorithm
     int getNumberOfNodesEvaluated();
}
```

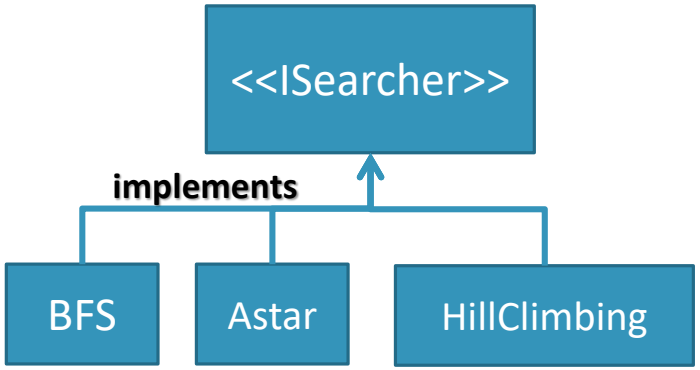
The expected common functionality to any searcher

The client only needs to know the interface

- The “client” is another programmer, that may use the searching algorithms

```
public void testSearcher(ISearcher searcher, ISearcheable searchable) {
     Solution solution=searcher.search(searchable);
     int n = searcher.getNumberOfNodesEvaluated();
}
```

Dependency Injection!



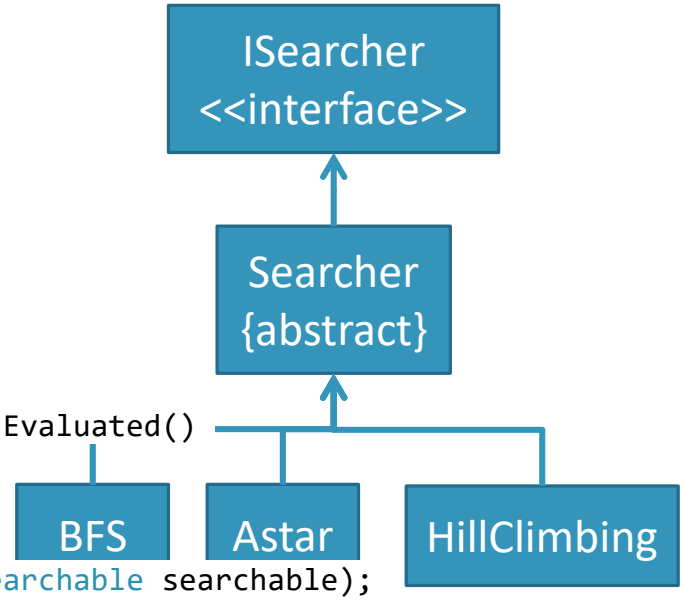
Examples:
tester.testSearcher(new BFS(), new EightPuzzle());
tester.testSreacher(new Astar(), new EightPuzzle());
tester.testSearcher(new HillClimbing(), new MazeDomain());

What all searchers have in common?

- Let's assume that all searching algorithms has a priority queue of states
- It will be wasteful to implement it over and over again with each algorithm…
- We can use an abstract class!

```
public abstract class Searcher : ISearcher
{
    private MyPriorityQueue<State> openList;
    private int evaluatedNodes;

    public Searcher(){
        openList=new MyPriorityQueue<State>();
        evaluatedNodes=0;
    }

    protected State popOpenList()
    {
        evaluatedNodes++;
        return openList.poll();
    }
```

```
// a property of openList
public int OpenListSize
{ // it is a read-only property :)
    get { return openList.Count; }
}

// ISearcher's methods:

public virtual int getNumberOfNodesEvaluated()
{
        return evaluatedNodes;
}
public abstract Solution search(ISearchable searchable);
}
```



BFS – best first search

Best First Search:

```
OPEN = [initial state]   // a priority queue of states to be evaluated
CLOSED = []        // a set of states already evaluated
while OPEN is not empty
do
  1. n ← dequeue(OPEN) // Remove the best node from OPEN
  2. add(n,CLOSED)          // so we won't check n again
  3. If n is the goal state,
       backtrace path to n (through recorded parents) and return path.
  4. Create n's successors.
  5. For each successor s do:
       a. If s is not in CLOSED and s is not in OPEN:
              i.  update that we came to s from n
              ii. add(s,OPEN)
       b. Otherwise, if this new path is better than previous one
                     i.  If it is not in OPEN add it to OPEN.
              ii. Otherwise, adjust its priority in OPEN
done
```

The BFS search algorithm

```
public override Solution search(Isearchable searchable) { // Searcher's abstract method overriding
    addToOpenList(searchable.getInitialState()); // inherited from Searcher
    HashSet<State> closed = new HashSet<State>();
    while (OpenListSize > 0) {
            State n = popOpenList();  // inherited from Searcher, removes the best state
            closed.Add(n);
            if (n.Equals(searchable.getIGoallState()))
              return backTrace(); // private method, back traces through the parents
            // calling the delegated method, returns a list of states with n as a parent
            List<State> succerssors = searchable.getAllPossibleStates(n);
            foreach (State s in succerssors)
            {
                if (!closed.Contains(s) && !openContaines(s))
                {
                    // s.setCameFrom(n);  // already done by getSuccessors
                    addToOpenList(s);
                }
                else
                {
                    //...
```

Our BFS implementation is domain independent! ☺