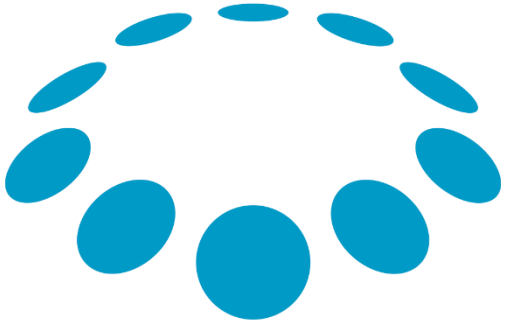
Object Oriented Design

Dr. Eliahu Khalastchi

2017



המסלול האקדמי
המכללה למינהל

Analysis

Understand our problem & required functionality

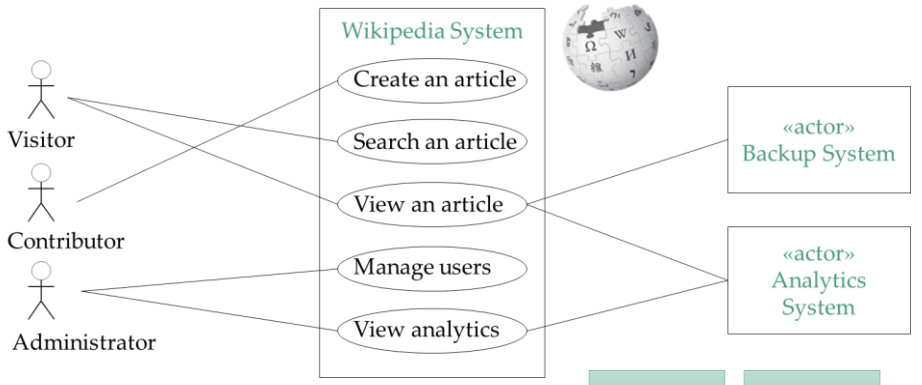
Object Oriented Design

Plan a solution that meets these requirements

Implement the design with an Object Oriented Programming Language

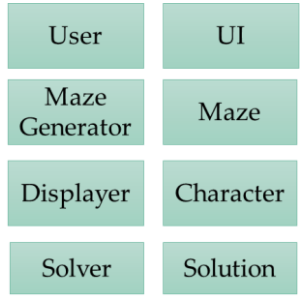
Object Oriented Analysis – the steps

1. Gather **requirements**

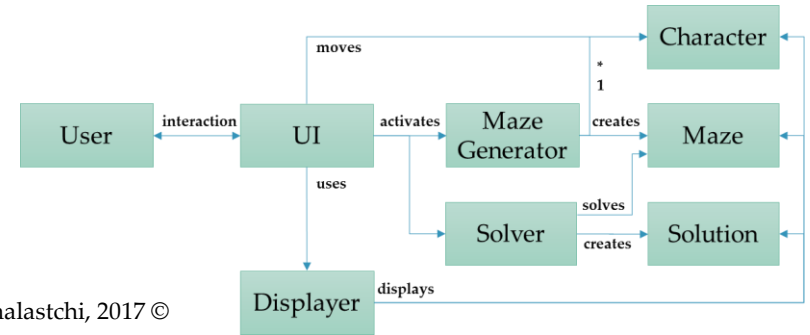


2. **Describe** the application

3. **Identify** the main objects



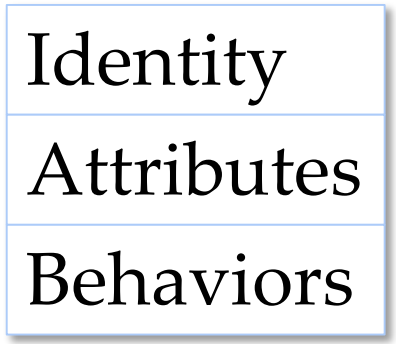
4. Describe the **interaction** between these objects



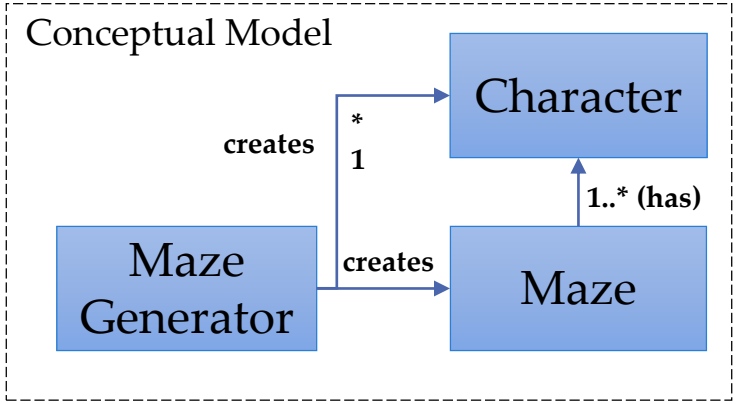
5. Create a **class diagram**

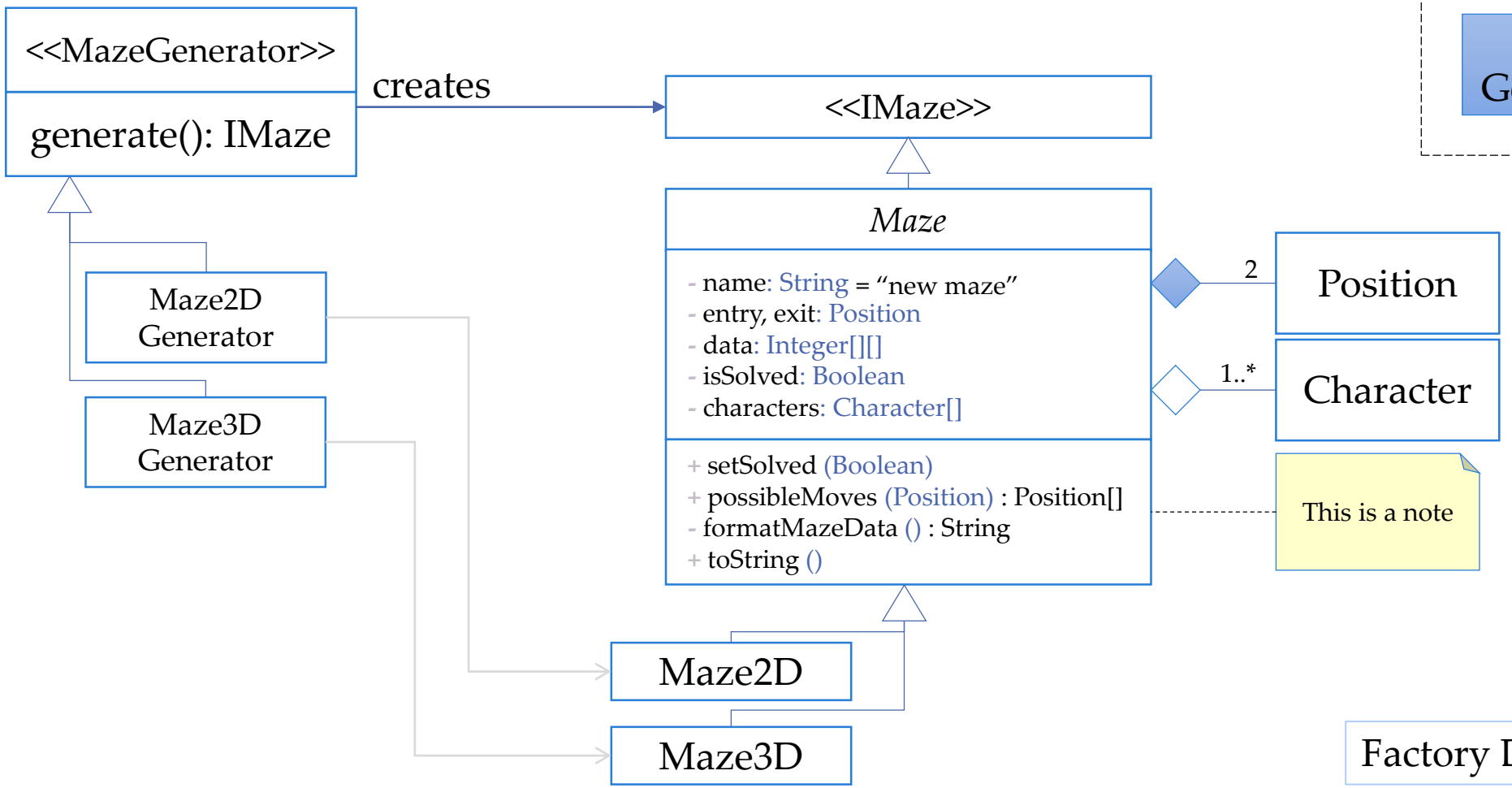
Creating a class diagram

- After the conceptual model is created, we can identify responsibilities
- We can **assign responsibilities** to classes
  - **One, and only one**, responsibility to a class
  - Each object is responsible of **its own attributes**
- These responsibilities define the class behaviors / methods



Class Diagram





Factory Design Pattern!

Object Oriented Programming

Analysis

Understand our problem & required functionality

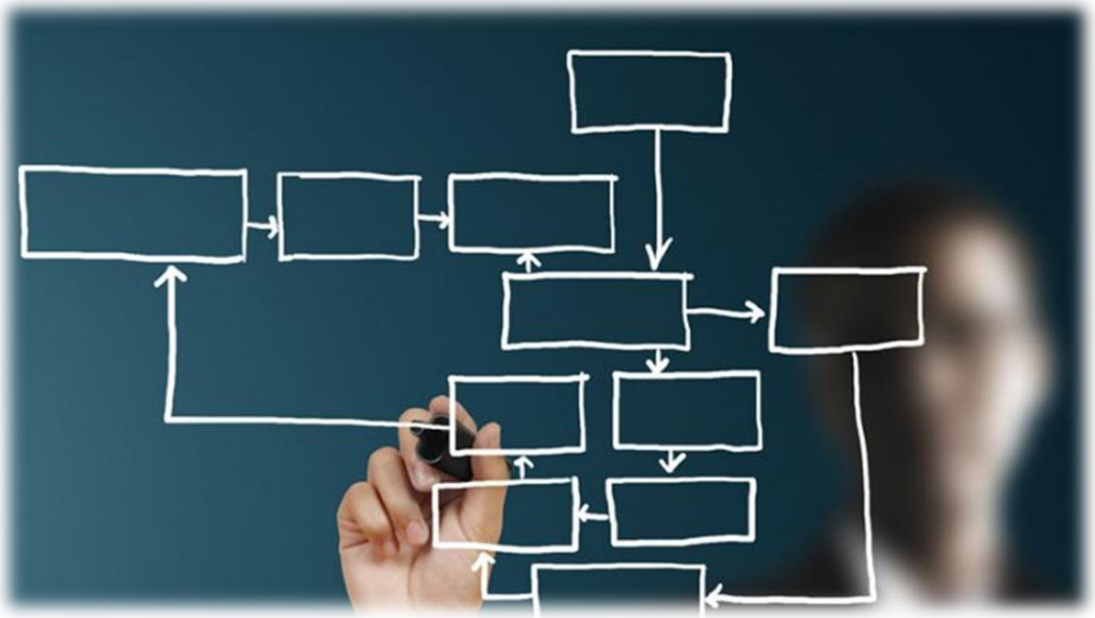
Design

Plan a solution that meets these requirements

Implement the design with an Object Oriented Programming Language

Design Principles

Object Oriented Design Principles



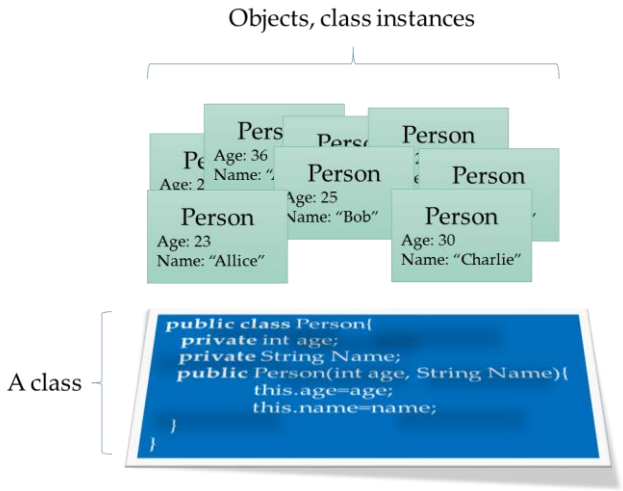
Agenda

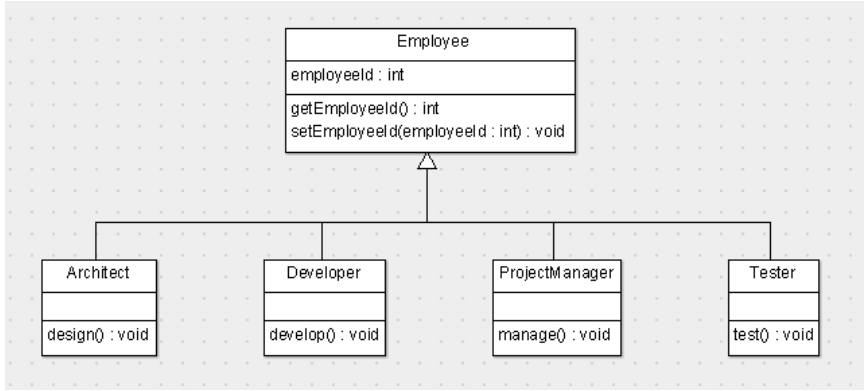
- General principles
- Code smells
- SOLID principle
- GRASP principle

General Principles

In previous lessons…

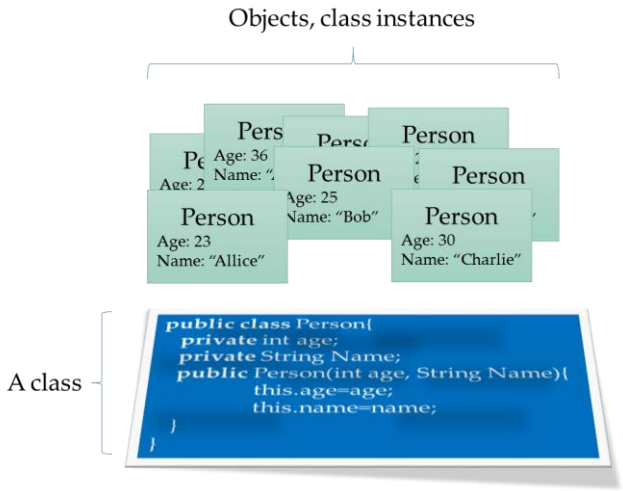
- Abstraction — Of types…
- Encapsulation — Of mechanisms… — Private members, Iterators, etc.
- Inheritance — Class hierarchy to prevent duplicated code…
- Polymorphism — Generalization of code (general algorithms & containers)

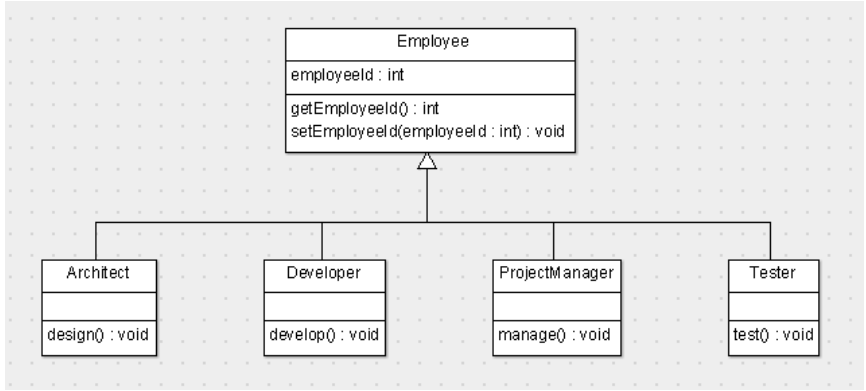




How to define classes?

- Abstraction
  - Define a **TYPE**, not a specific instance
  - Define only what is important to our application
- Encapsulation
  - An object should hide its contents (as much as possible)
  - It should only reveal the necessary services it provides
  - Other programmers cannot misuse the objects
  - Dependencies are reduced
    - A change in one place does not affect other places in the code
- Inheritance
- Polymorphism





No encapsulation, bugs are introduced

Struct Student

Name
ID
Age
…

Moshe
12345789
-3
…

Somewhere in the project:

Function A():
X← user input
Moshe.age=X

Function B():
X← read from file
Moshe.age=X

Fixing the bug – all over the entire project

Struct Student

Name
ID
Age
…

Moshe
12345789
-3
…

Somewhere in the project:

```
Function A():
X← user input
If X>0 Then
   Moshe.age=X
```

```
Function B():
X← read from file
If X>0 Then
   Moshe.age=X
```

Using encapsulation, only one place to fix the bug

Class Student

```
class Student{
  private:
    float _age;
     …

  public:
     void setAge(float age){
       _age=age;
     }
};
```

```
Student moshe, yossi;
moshe._age=-3; // error! _age is private

we have to use the setter method all over the project
moshe.setAge(-3);
…
yossi.setAge(-5);

there is only one place in the project to fix the bug:
the student class
```

Using encapsulation, only one place to fix the bug

Class Student

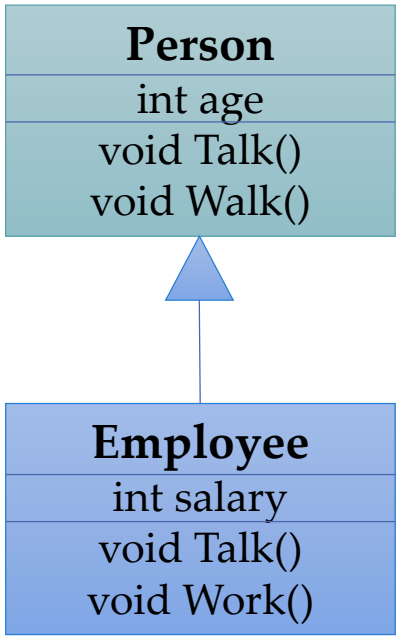
```
class Student{
  private:
    float _age;
     …

  public:
     void setAge(float age){
       if(age>0)
          _age=age;
     }
};
```

```
Student moshe, yossi;
moshe._age=-3; // error! _age is private

we have to use the setter method all over the project
moshe.setAge(-3);
…
yossi.setAge(-5);

there is only one place in the project to fix the bug:
the student class
```

Inheritance = less code to write

```
class Person
{
        int age;
    public:
        Person() {…} // constructor
        void Talk() {…}
        void Walk() {…}
};

class Employee: public Person
{
        int salary;
     public:
        Employee() {…} // constructor
        void Talk() {…} // override
        void Work(){…}
};
```

Class representation:

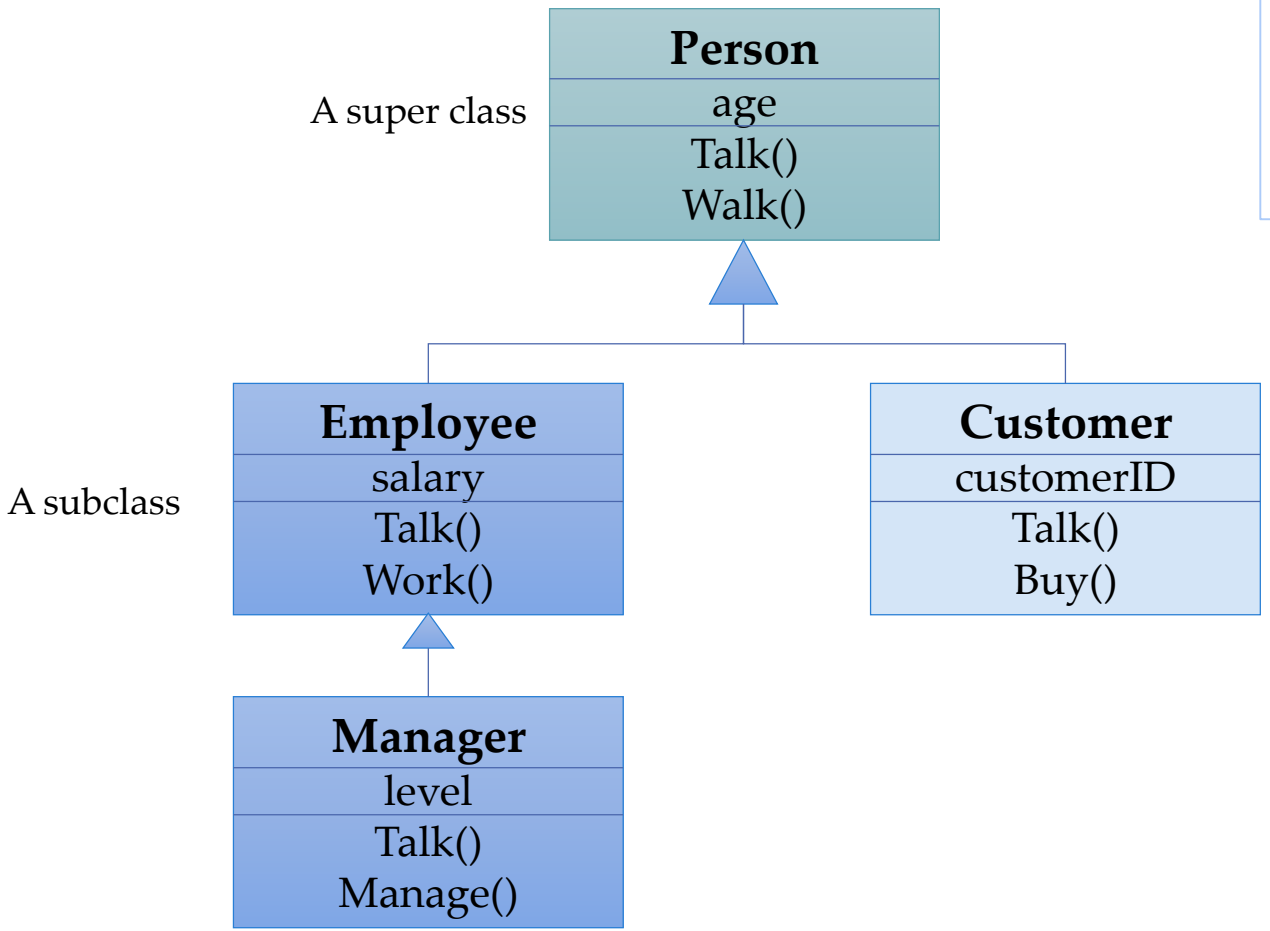


Object representation:

Person allice;

Employee bob;

Inheritance

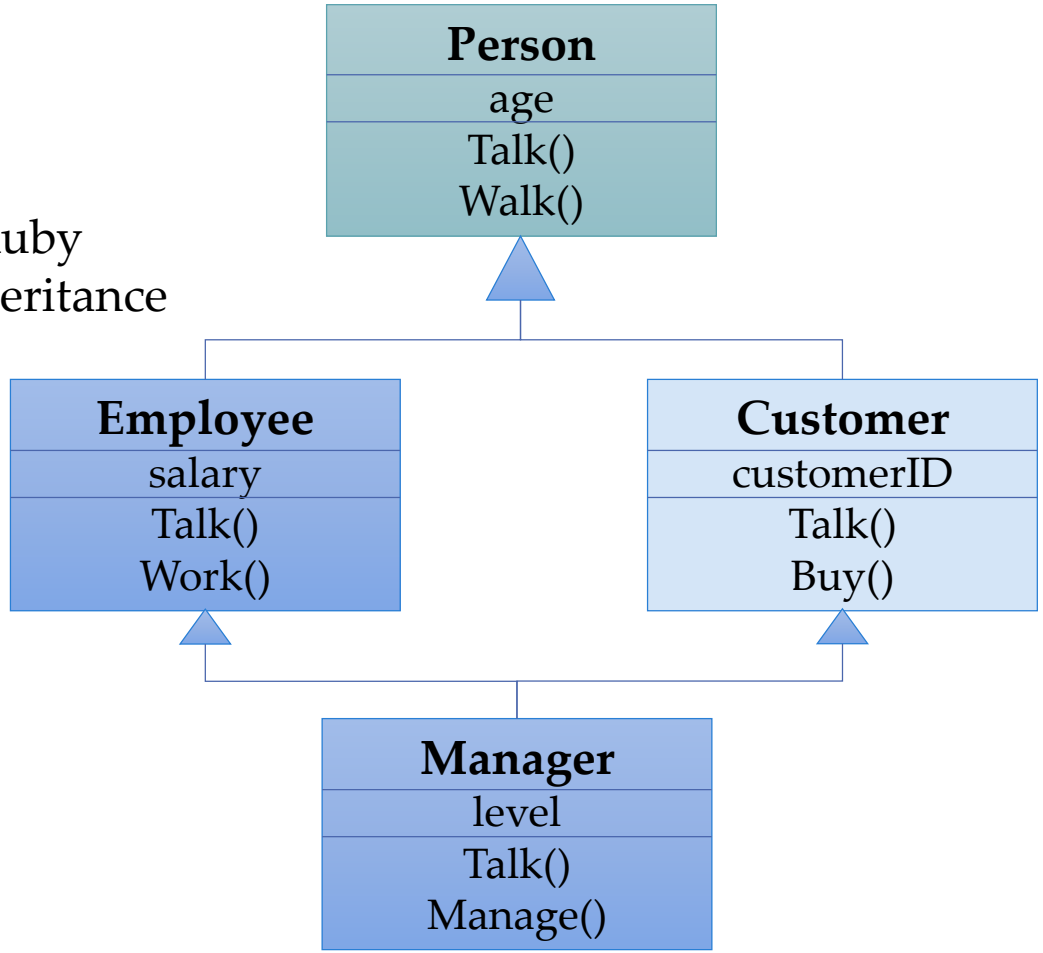


Inheritance:
- Saves time
- Enable reuse of code
- Enables polymorphism

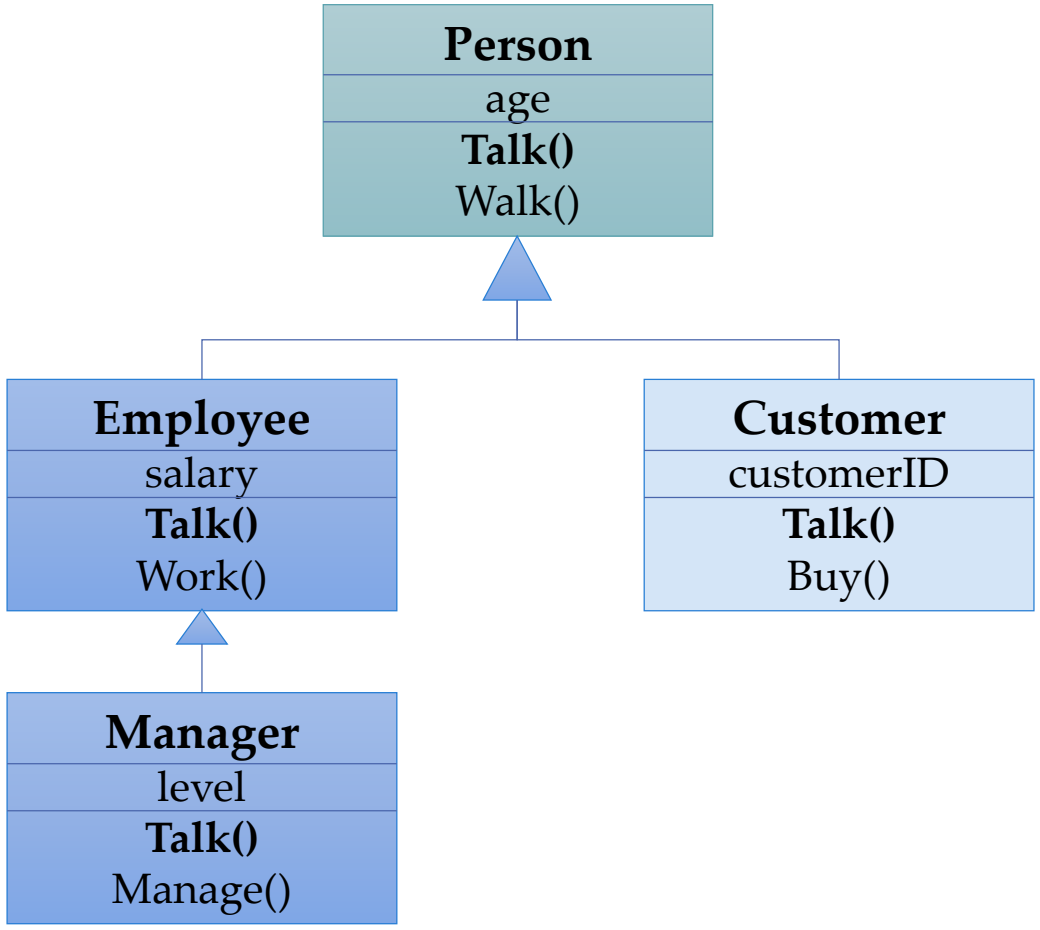
Multiple Inheritance

Multiple Inheritance:

- In C++
- Can be confusing
- Other languages
  - Java, C#, Objective C, Ruby
  - Allow only a single inheritance
  - A better idea…



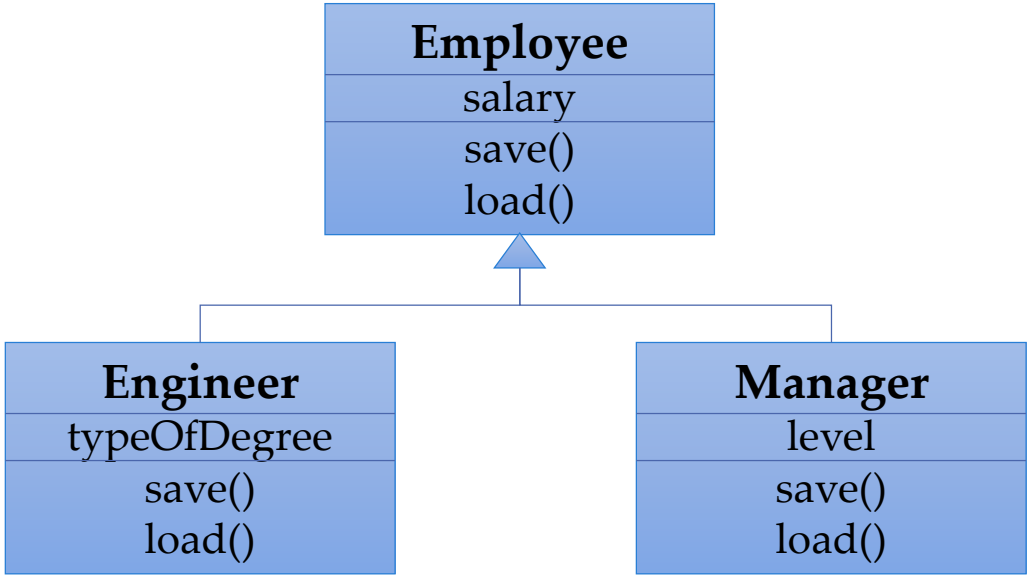
Polymorphism



- These are polymorphic persons
- Each has overridden the Talk() method
  - We can apply Talk() from any person
  - Without knowing which person it actually is

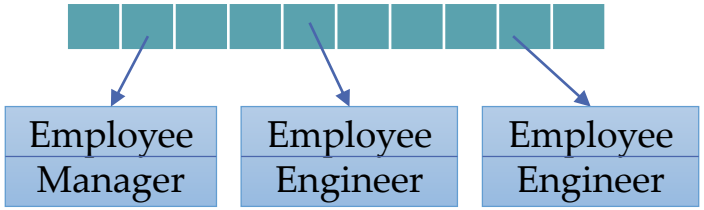
```
Person p;
if (isCustomer)
        p=new Customer();
else
        p=new Employee();
p.Talk();
```

Polymorphism enables General Containers



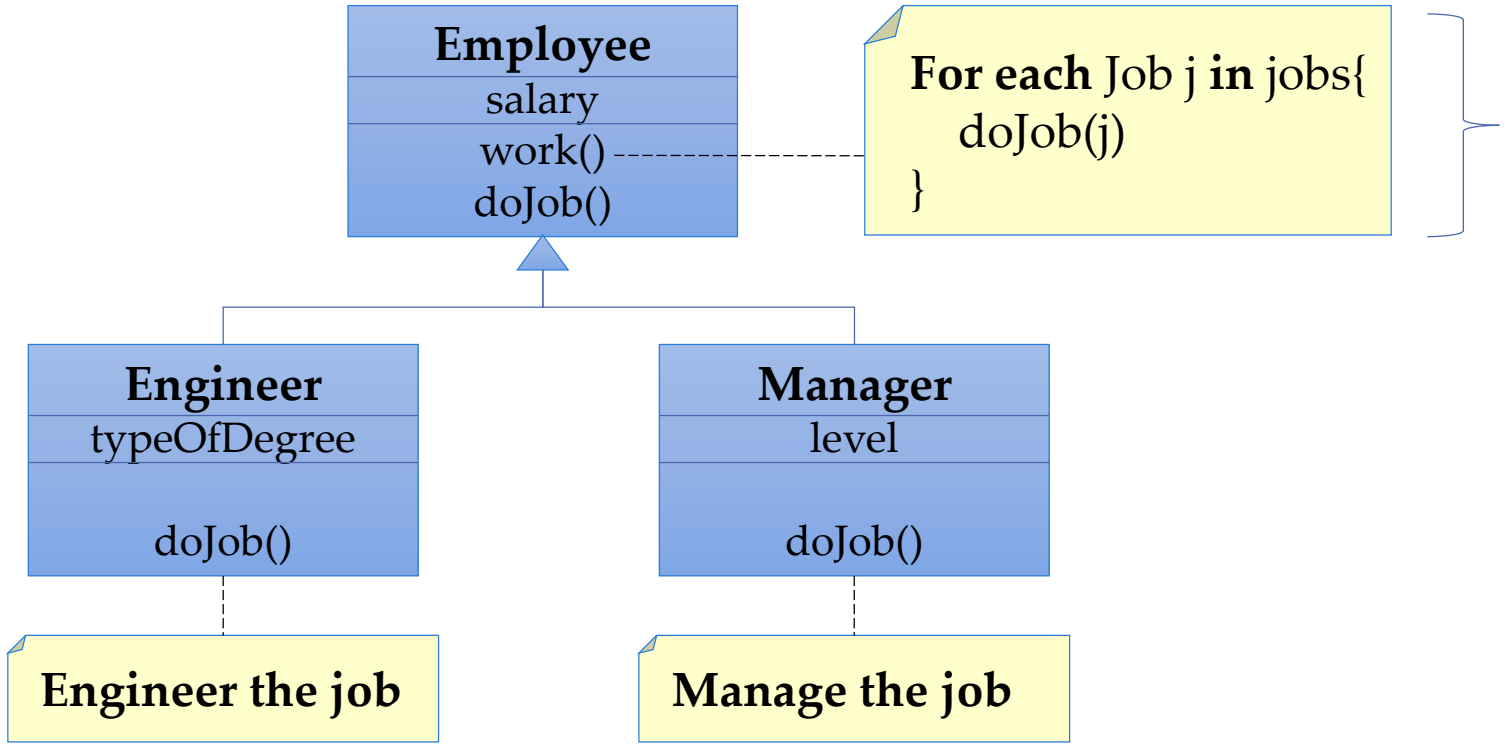
General Container:

```
Employee** employees;
```



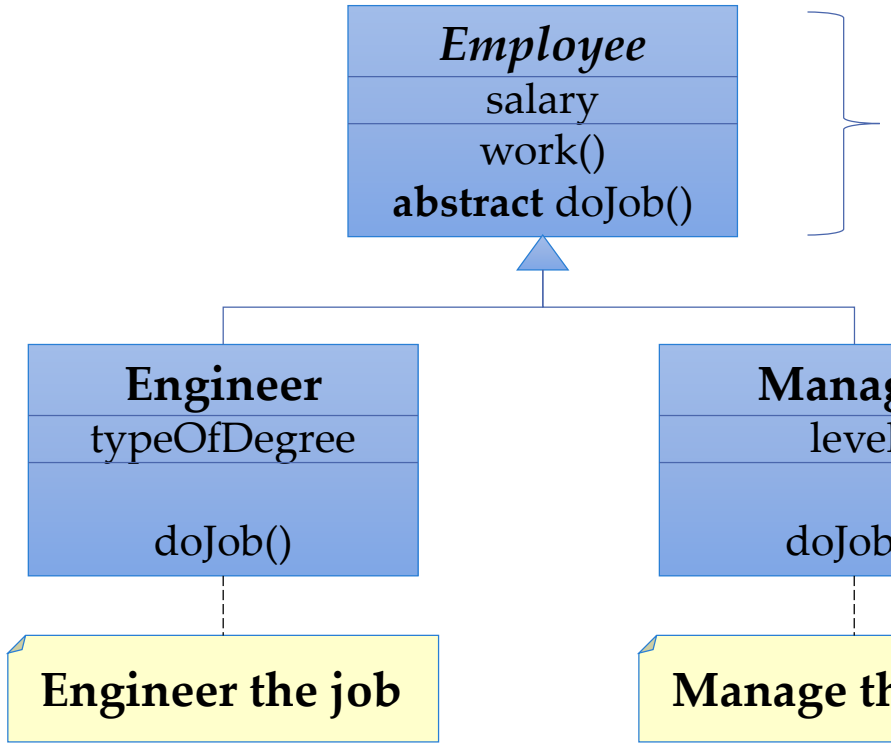

```
// save all employees to file
for (int i = 0; i < size; i++)
   employees[i]->save(/*...*/);
```

Polymorphism enables generic algorithms



This is a general algorithm. Changing doJob() will change the behavior of work(), without changing the source code of work()

Abstract Class



- Employee is abstract
- We cannot instantiate an Employee
- We can instantiate a subclass of an Employee
  - If it is not an abstract class

These classes MUST implement the inherited abstract methods

General Principles

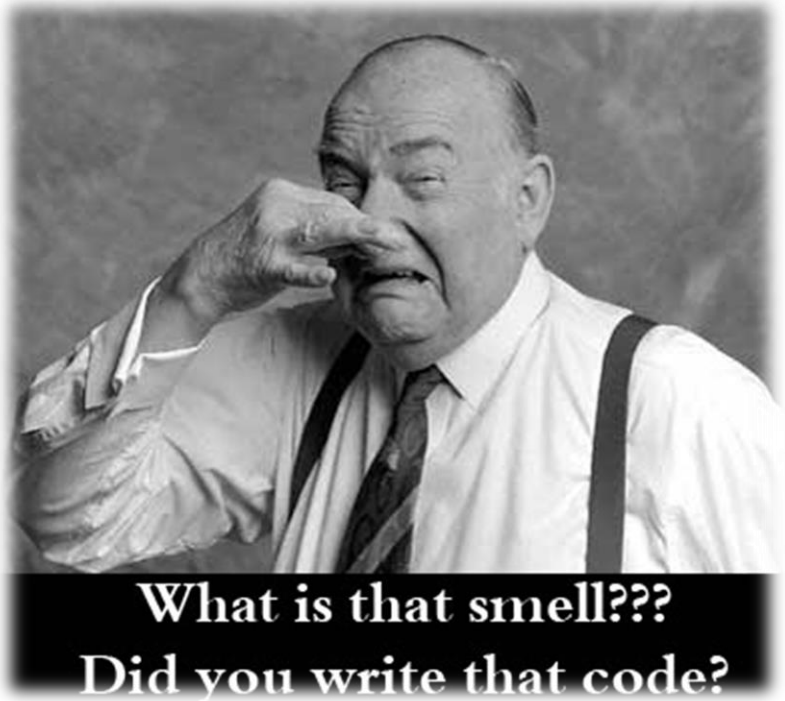
- DRY: don't repeat yourself
  - Obvious: we do not copy and paste blocks of code
  - Not obvious: also avoid duplications in
    - The data base schemers, diagrams and documentation…
  - "A single source of truth"

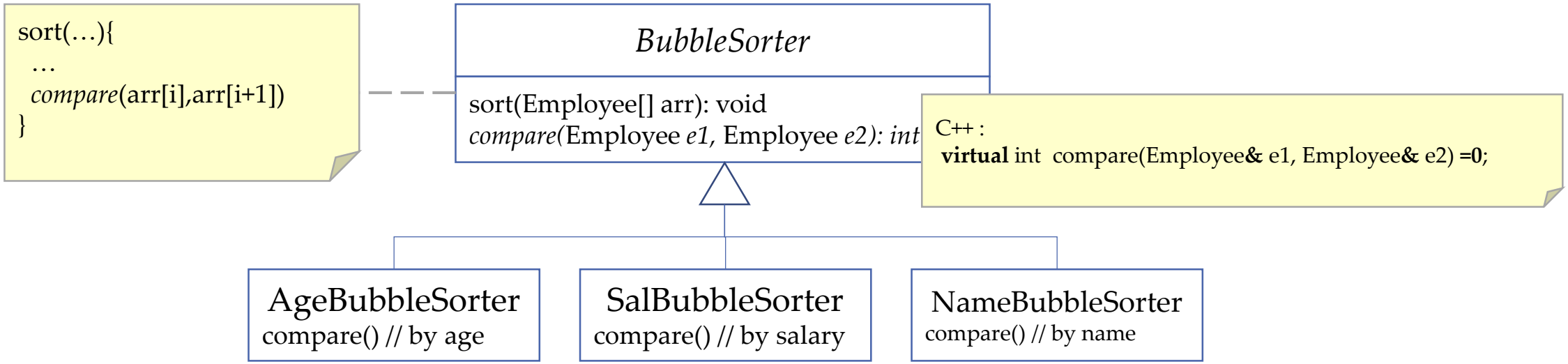
- YAGNI: you ain't gonna need it
  - Don't write speculative code, solve the problem you know to exist

- Avoid code smells…

Code Smells

You must avoid them!

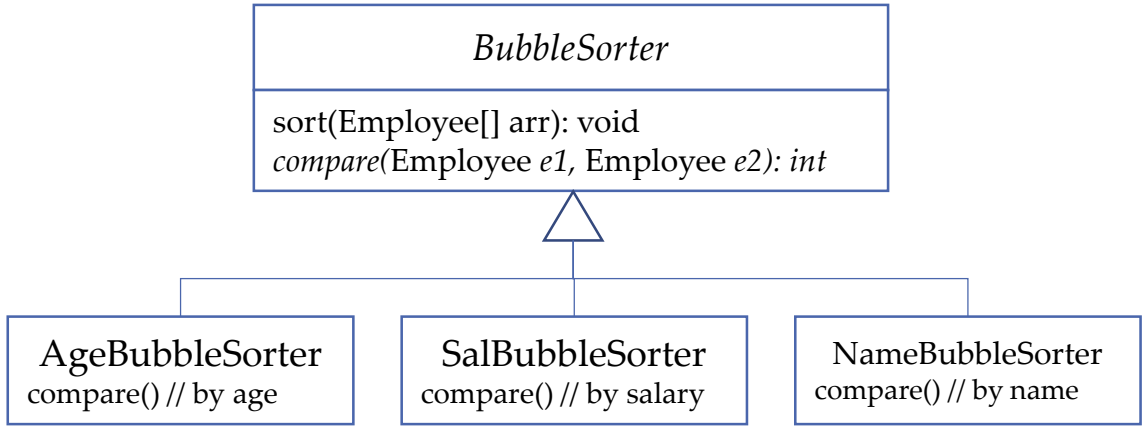


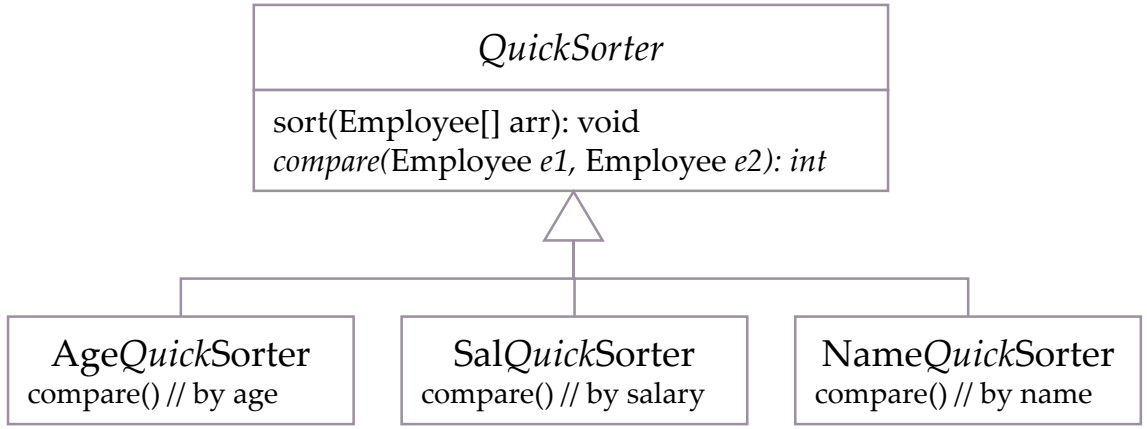



```
BubbleSorter* s = new AgeBubbleSorter();
s->sort(employees);
```

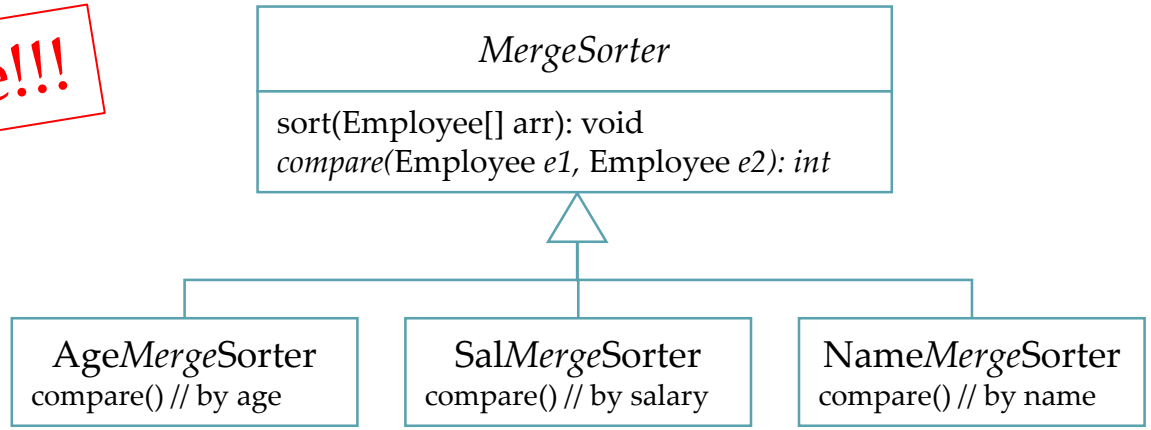
```
BubbleSorter* s = new NameBubbleSorter();
s->sort(employees);
```

Where is the code smell???





Duplicated Code!!!



```
class AgeBubbleSorter : BubbleSorter{
  int compare(Employee& e1, Employee& e2){
        return e1.getAge() – e2.getAge();
  }
}
```

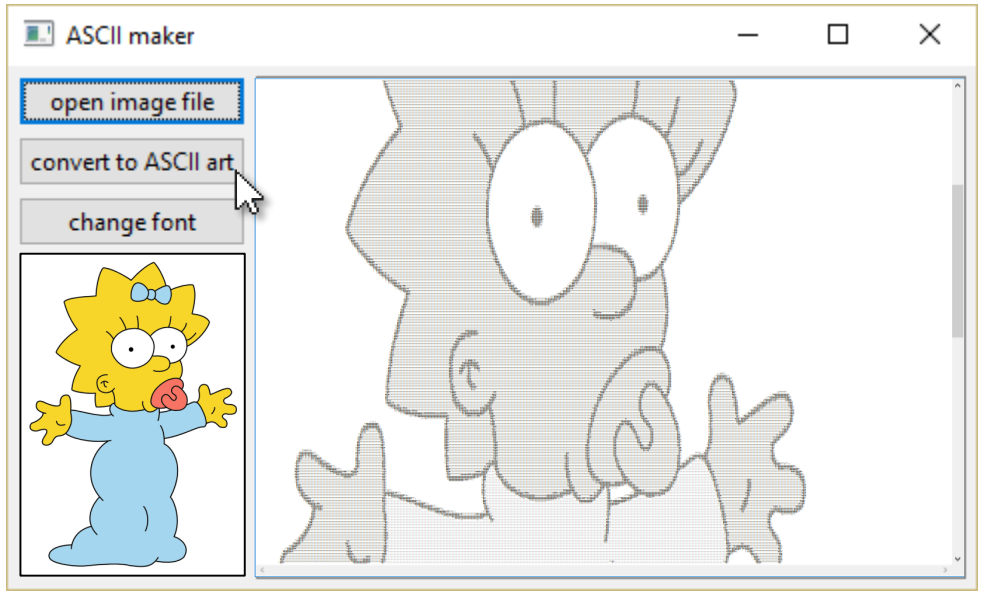
```
class AgeQuickSorter : QuickSorter{
  int compare(Employee& e1, Employee& e2){
        return e1.getAge() – e2.getAge();
  }
}
```

Duplicated Code!!!

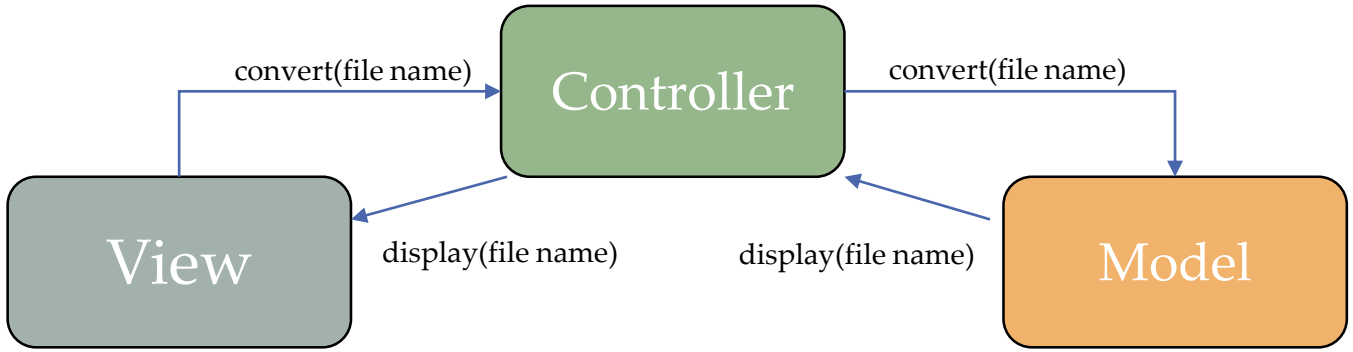
Application-Level Code Smells…

- Duplicated code
  - Identical or very similar code exists in more than one location
- Contrived complexity
  - Forced usage of complicated design patterns
  - where simpler design would suffice
- Trivial modules or layers
  - When they only call the next layer down

Where is the code smell???



Trivial modules or layers



Convert file to ASCII art

Class-Level Code Smells…

- The GOD class
  - A master class that tries to do everything…
  - Has very different responsibilities, which have nothing to do with each other
  - Has too many instance variables (data members)
  - Common for a procedural programmer that has leaned only the syntax
- Feature Envy
  - Does very little except use all the methods of another class
- Inappropriate intimacy
  - A class that has dependencies on implementation details of another class

Class-Level Code Smells…

- Freeloader
  - A class that does too little
- Conditional complexity
  - Too many branches or loops
  - This may indicate a function needs to be broken up into smaller functions
  - Or that it has potential for simplification
- Downcasting
  - A type cast which breaks the abstraction model
  - Example, java containers before java generics

Class-Level Code Smells…

- A Class with too many variables
  - Consider divide the responsibilities with another class
- Strikingly similar subclasses
  - Example, two subclasses that handle different inputs in the same way
- Multiple inheritance
  - The diamond of death

Class-Level Code Smells…

- Too many none-public methods
  - Harder to test…
- Data class
  - Avoid classes that passively store data
  - Classes should contain data and methods
- Middle Man
  - A class that delegates all of its work – cut the middle man
  - Wrappers should manipulate something

Method-Level Code Smells…

- Too many parameters
  - Harder to read, call and test
  - The purpose of the function is ill-conceived
  - Refactor the responsibilities
- Long method
  - A method that has grown too large
  - Typically, written by a procedural programmer
- Excessively short / long identifiers
  - The name of a variable should reflect its function unless the function is obvious
- Excessive return of data
  - A method that returns more than what each of its callers needs

General Code Smells…

- Same name, different meaning
- Inconsistent names
  - E.g., if you have open() then you should have close()
- Pointless comments
  - The code should explain itself
- Lack of comments
  - Where it is needed

SOLID Principles

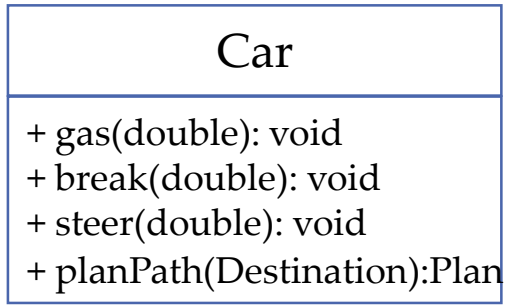
By Robert Martin, a.k.a “Uncle Bob”

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

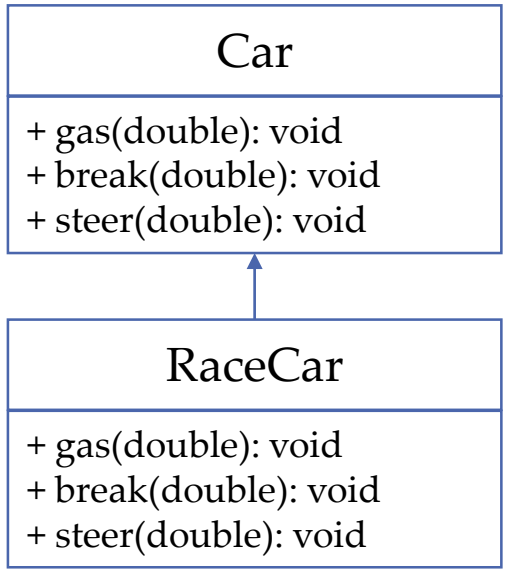
Solid principles of OO design

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



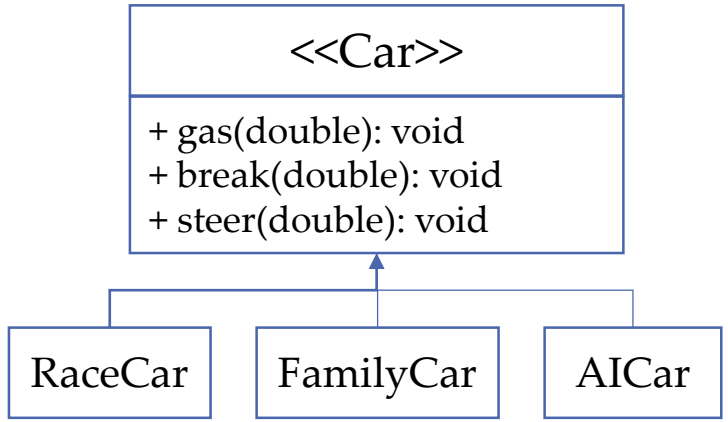
Solid principles of OO design

- 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

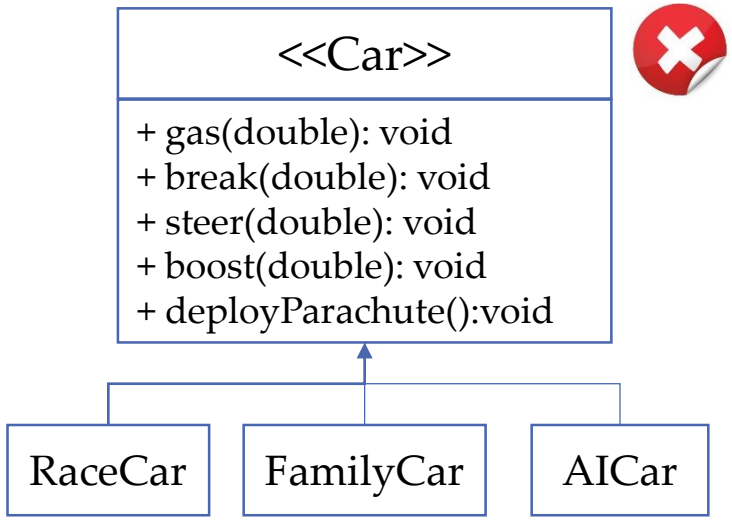



```
void drive(Car c){
  …
  c.gas(40);
  …
  c.break(20);
  c.steer(25);
  …
}
```

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

}
```

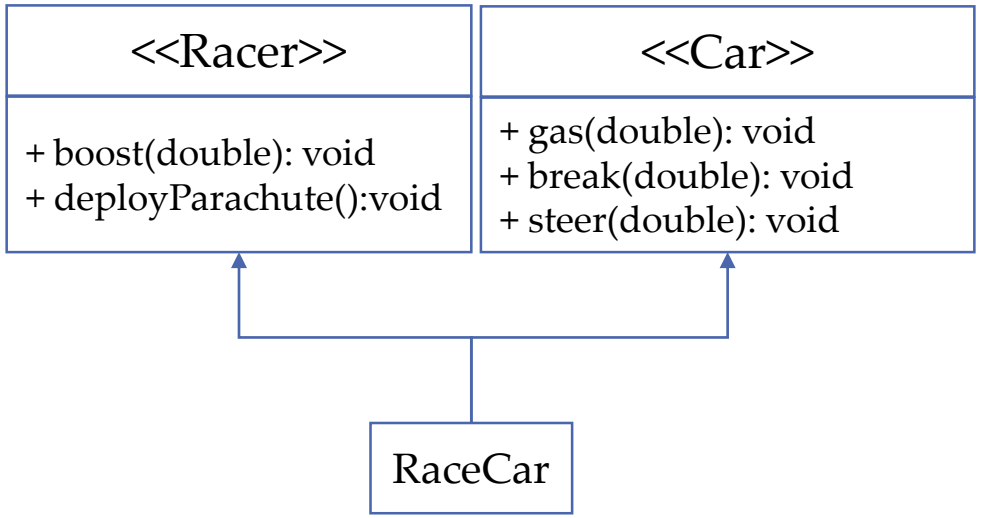
Solid principles of OO design

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



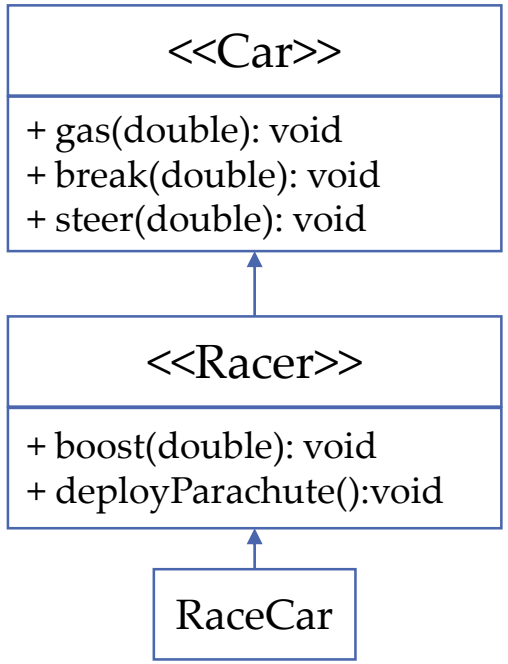
Solid principles of OO design

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



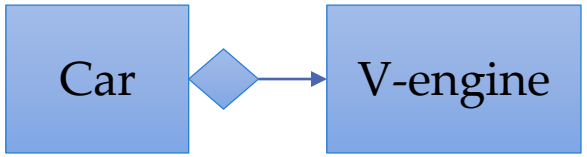
Solid principles of OO design

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



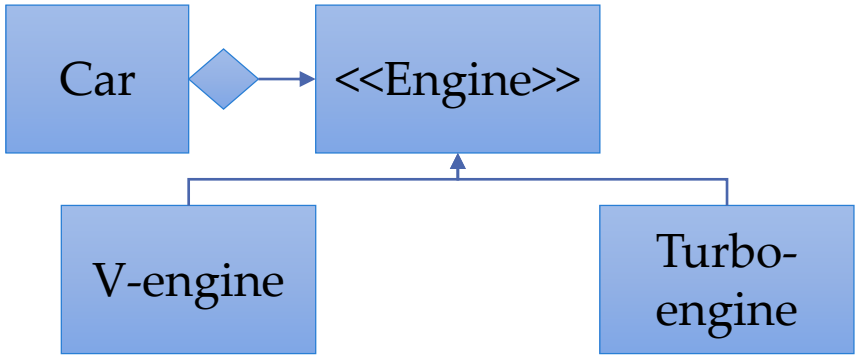
Solid principles of OO design

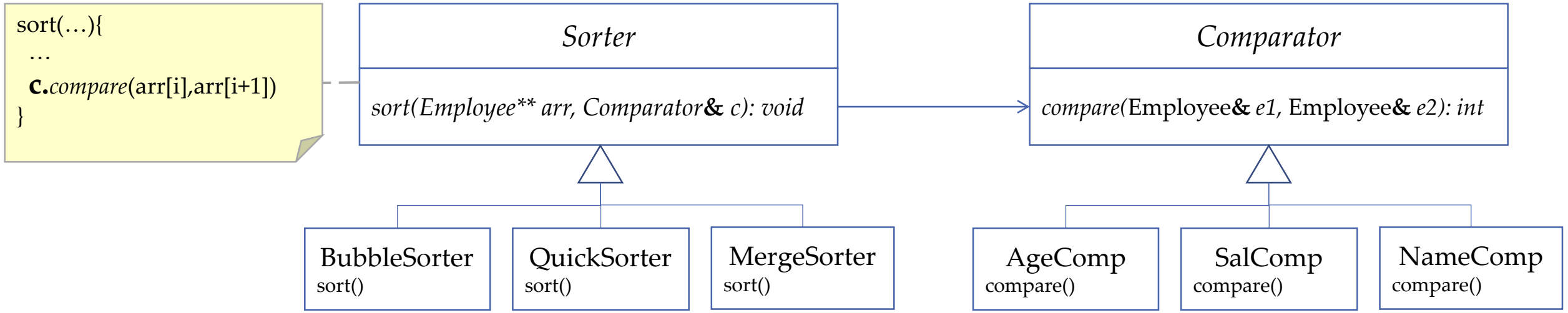
- S
  - Single Responsibility Principle
- O
  - Open / Closed Principle
- L
  - Liskov Substitution Principle
- I
  - Interface Segregation Principle
- D
  - **Dependency Inversion Principle**



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**





- ☑ 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());
```

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

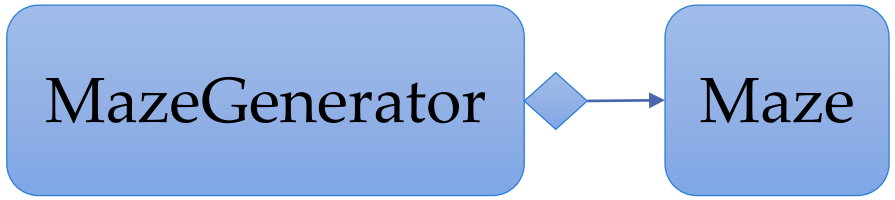
- Assign a responsibility to the class that has the information to fulfill it
- Which class should calculate the number of unread emails?

User | Mailbox | Email

Creator

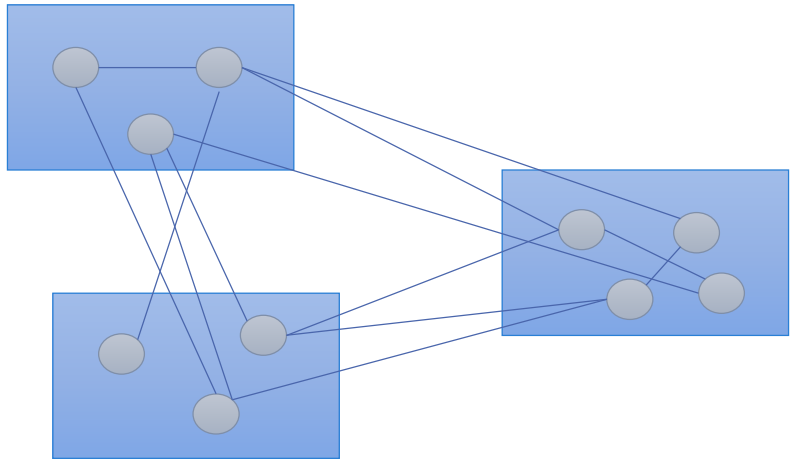
- Who is responsible for creating an object?

- 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?

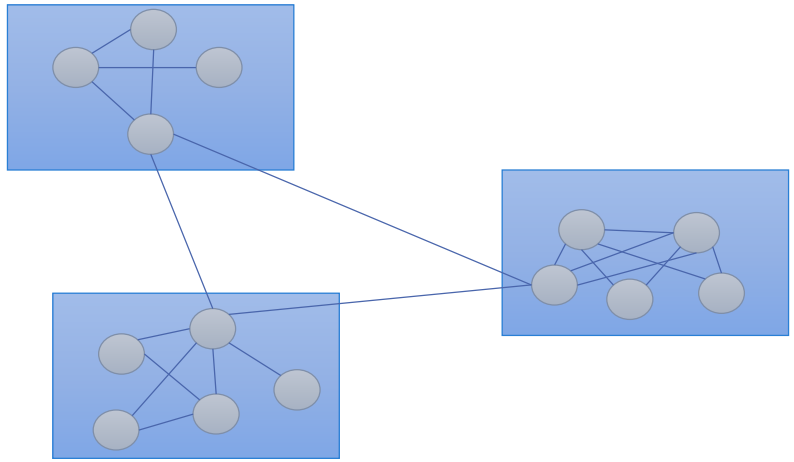


Low Coupling / High Cohesion

- **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

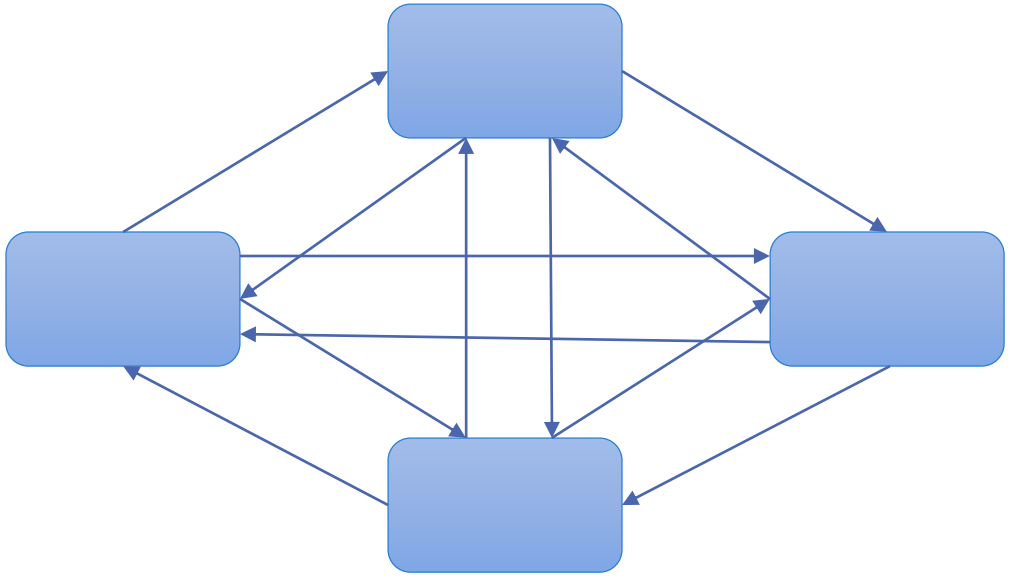
Pure Fabrication

- 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

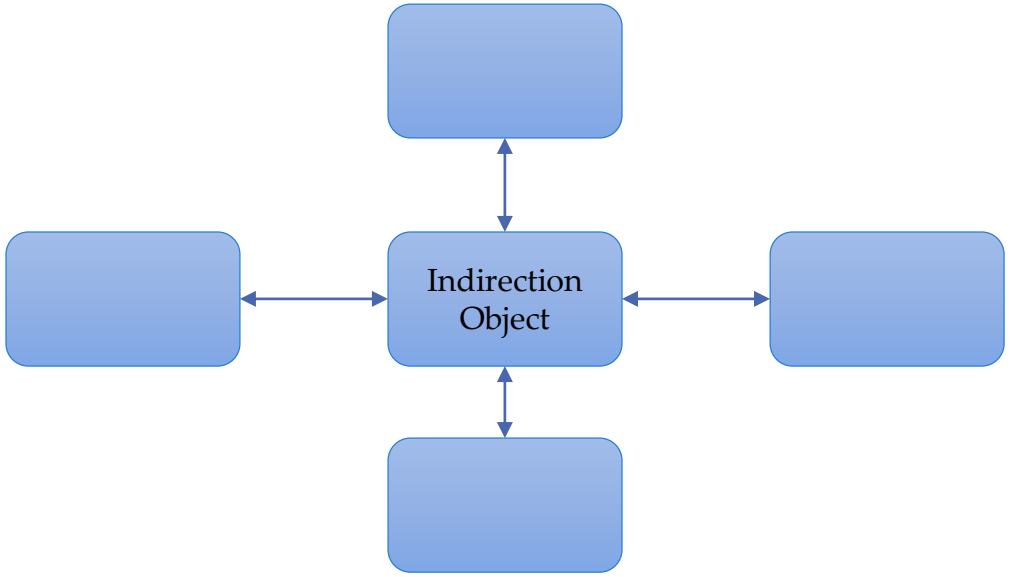
Indirection

- To reduce coupling, introduce an intermediate object



Indirection

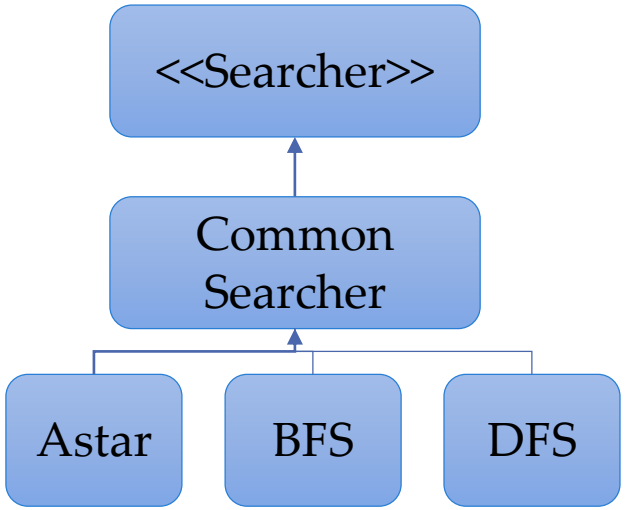
- To reduce coupling, introduce an intermediate object



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.