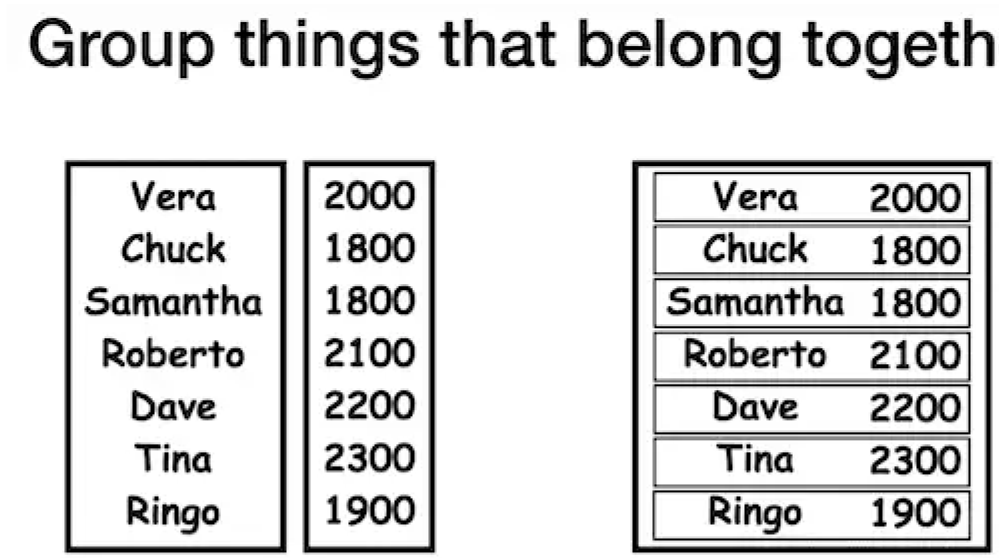
Abstraction and Inheritance

- In this topic, we solve issues that we experienced in the previous topics.
- We discuss how `Abstraction` and `Inheritance` can be used to refactor code smells.



Group things that belong together

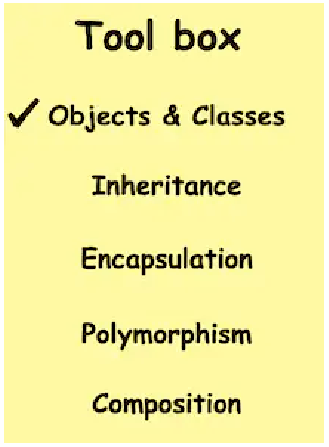
Vera	2000
Chuck	1800
Samantha	1800
Roberto	2100
Dave	2200
Tina	2300
Ringo	1900

Vera	2000
Chuck	1800
Samantha	1800
Roberto	2100
Dave	2200
Tina	2300
Ringo	1900

- The problems with our prototype software are from keeping two lists to track employees' salaries.

The Solution: Classes and Objects

- But we know how to combine them using classes and objects.



Tool box

✓ Objects & Classes

Inheritance

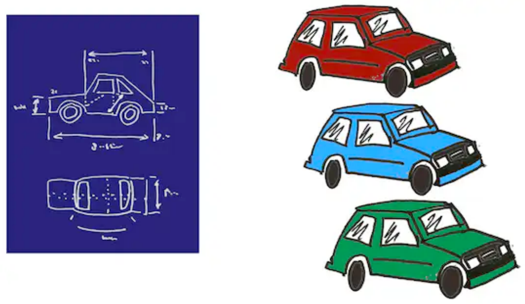
Encapsulation

Polymorphism

Composition

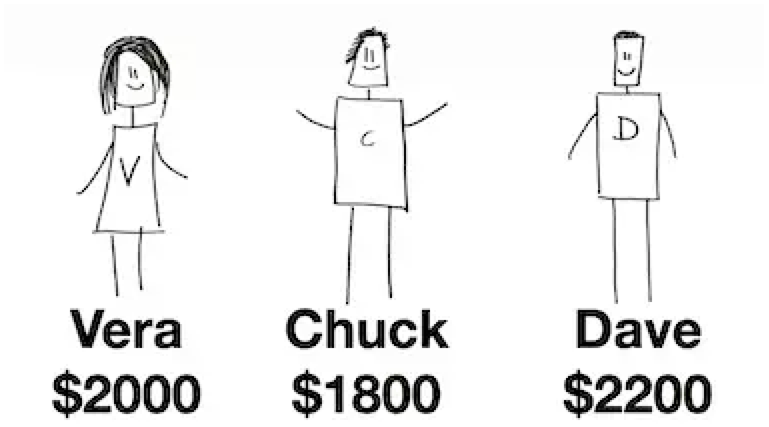
Class as Blueprint

- We can think of a class as a blueprint to make an object.



Creating the Employee Class

- We have a real-world object: an employee.
- We can make a class that abstracts an employee.
    - We need to abstract from the attributes of the object: name and salary.



```python
class Employee:
    def __init__(self, name, salary):
        self.name = name
        self.salary = salary
```

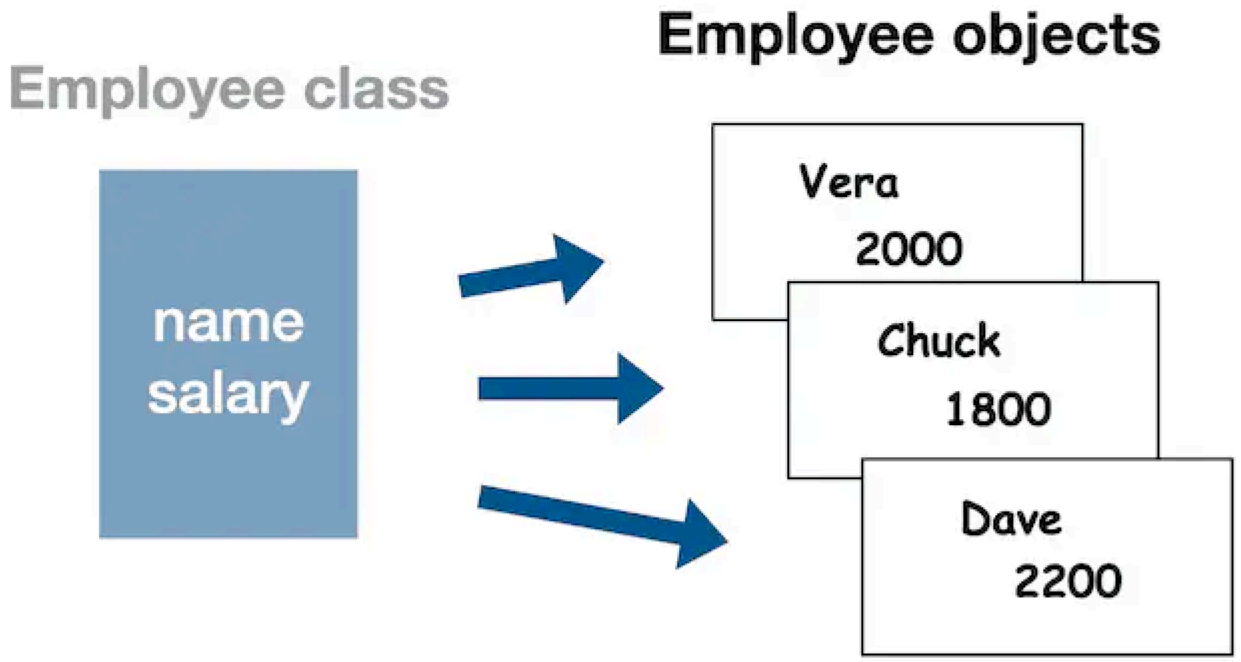
Java vs Python

- We can use any OOP language for abstraction.
- For example, we can use Java to express the same class.
- Java is a typed language, so we need to add data types.

```
public class Employee { // Java
    String name;
    int salary;
    public Employee(String name, int salary) {
        this.name = name;
        this.salary = salary;
    }
}
class Employee: # Python
    def __init__(self, name, salary):
        self.name = name
        self.salary = salary
```

Objects

- To use the class, we need to instantiate it in memory.
- The instantiated class is called an object.
- We store each object in a list in this example.



```
employees = [
    Employee("Vera", 2000),
    Employee("Chuck", 1800),
    Employee("Dave", 2200),
]
```

Java vs Python

- Dynamic languages such as Python or Ruby are easier to use than static languages such as Java or C#.

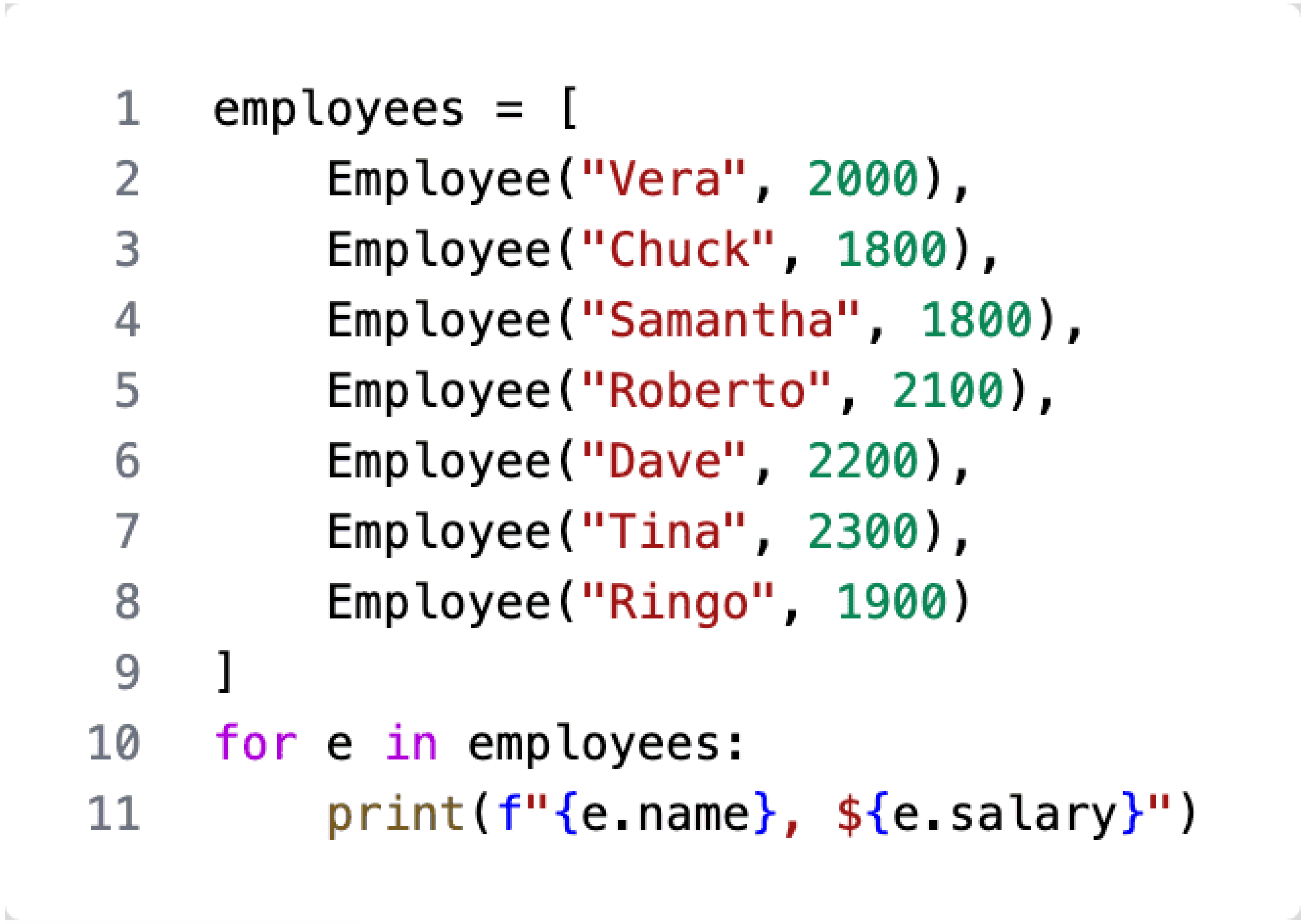
```java
List<Employee> employees = new ArrayList<Employee>();
employees.add(new Employee("Vera", 2000));
employees.add(new Employee("Chuck", 1800));
employees.add(new Employee("Dave", 2200));
```

Dynamic vs Static Languages

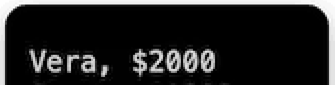
- It is easier to make bugs with dynamic languages in general.
- Dynamic language compilers do not check anything when they compile the code.
- So, dynamic languages are mainly used for prototypes.

- Static languages can find type-related bugs at compile time.
- The compiled code runs faster than a dynamic language in general.
- Static languages are mainly used for production.

Abstraction of Employee



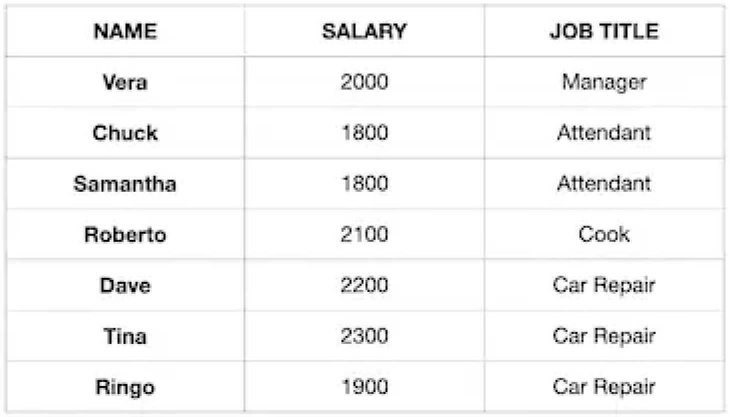
```python
employees = [
    Employee("Vera", 2000),
    Employee("Chuck", 1800),
    Employee("Samantha", 1800),
    Employee("Roberto", 2100),
    Employee("Dave", 2200),
    Employee("Tina", 2300),
    Employee("Ringo", 1900)
]
for e in employees:
    print(f"{e.name}, ${e.salary}")
```



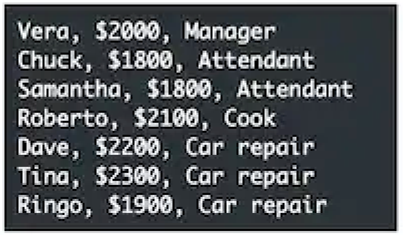
```
Vera, $2000
```

Change, Again

- Mr. Star likes the program, and he now wants to add the feature of printing job titles.



NAME	SALARY	JOB TITLE
Vera	2000	Manager
Chuck	1800	Attendant
Samantha	1800	Attendant
Roberto	2100	Cook
Dave	2200	Car Repair
Tina	2300	Car Repair
Ringo	1900	Car Repair



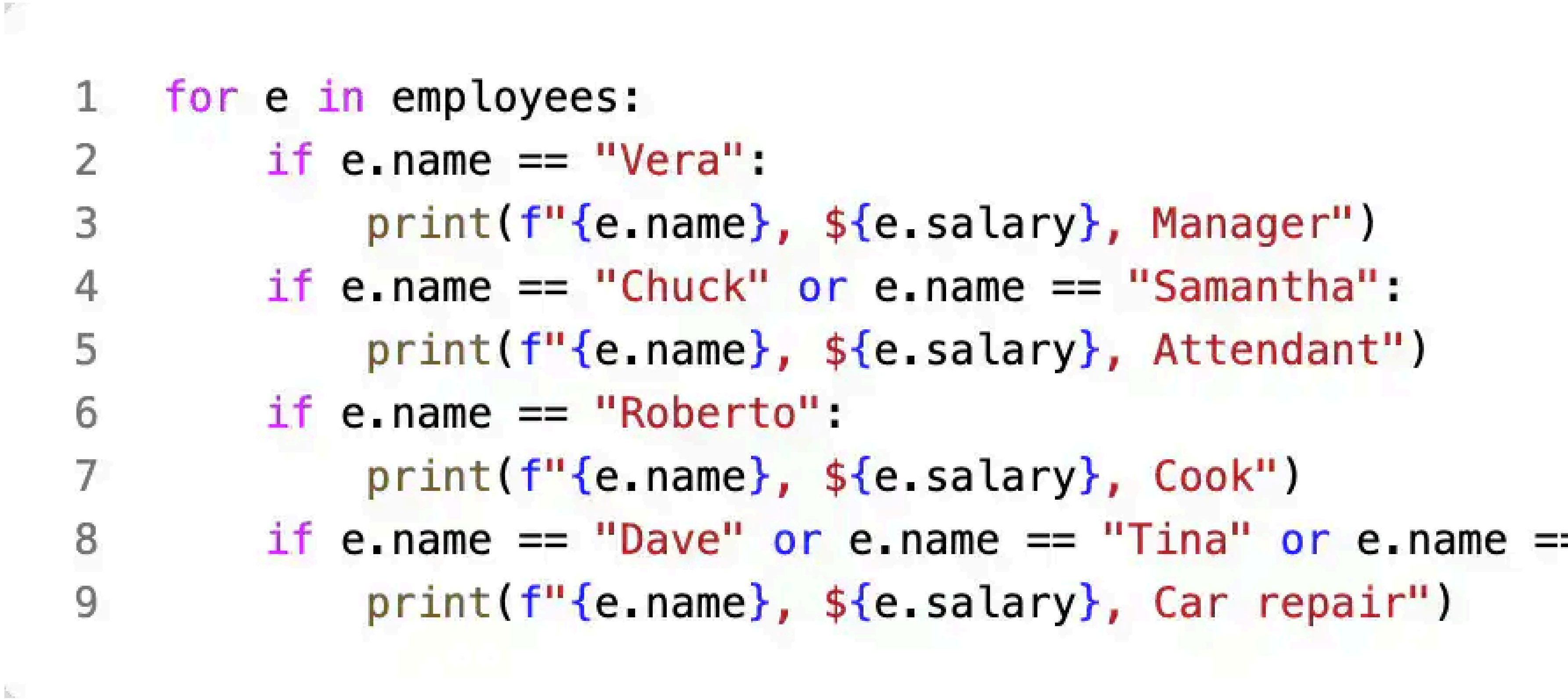
```
Vera, $2000, Manager
Chuck, $1800, Attendant
Samantha, $1800, Attendant
Roberto, $2100, Cook
Dave, $2200, Car repair
Tina, $2300, Car repair
Ringo, $1900, Car repair
```

Requirements Version 3

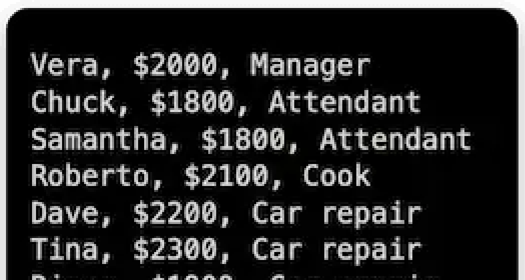
- After some discussion, we (software engineers) and Mr. Star (a client) agreed upon new requirements.
- We have a high-level requirement (epic requirement) with a sub-requirement.

```
Epic requirement:
As an "employer,"
I want to "see a list of the employees with their salaries,"
so that "I can manage my employees."

Sub-requirement 1: As a "manager,"
I want to "have the list including name, salary, and job title."
so that "I can track my employees effectively."
```

Quick Fix (Bad Solution)



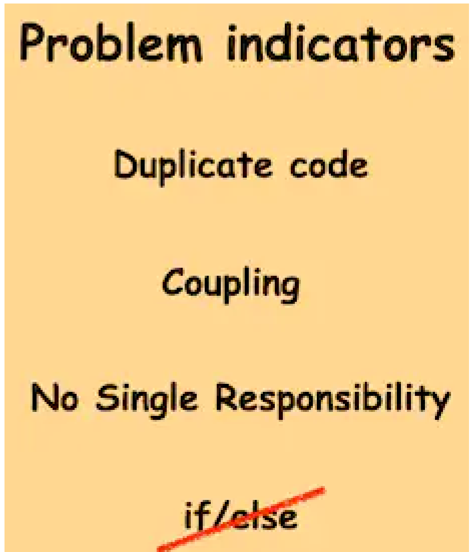
```
for e in employees:
    if e.name == "Vera":
        print(f"{e.name}, ${e.salary}, Manager")
    if e.name == "Chuck" or e.name == "Samantha":
        print(f"{e.name}, ${e.salary}, Attendant")
    if e.name == "Roberto":
        print(f"{e.name}, ${e.salary}, Cook")
    if e.name == "Dave" or e.name == "Tina" or e.name ==
        print(f"{e.name}, ${e.salary}, Car repair")
```



```
Vera, $2000, Manager
Chuck, $1800, Attendant
Samantha, $1800, Attendant
Roberto, $2100, Cook
Dave, $2200, Car repair
Tina, $2300, Car repair
```

Code Smell: if/else

- If/else code means we should change the code when we add new features.
- New changes can easily break this code.


Problem indicators

Duplicate code

Coupling

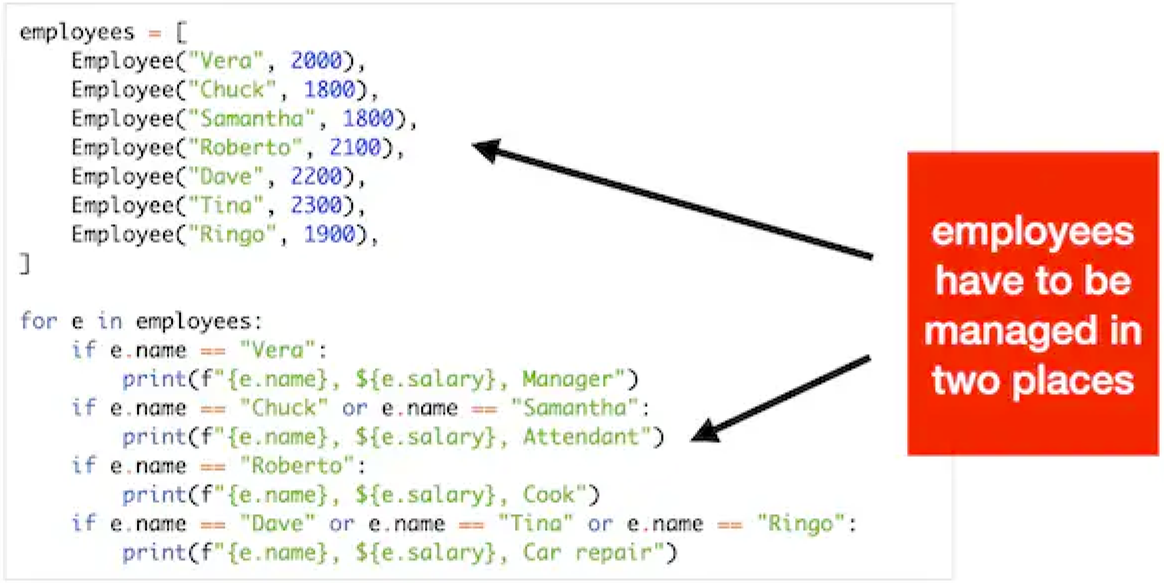
No Single Responsibility

~~if/else~~

The Two-Place Problem

- The employees have to be managed in **two places**.
- When there is a change, we should update the if/else statement and the list.

- We already made this mistake in the prototype development.

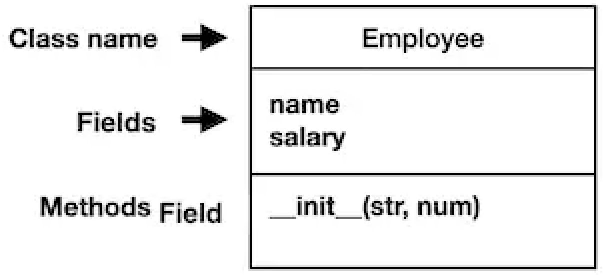


```
employees = [
    Employee("Vera", 2000),
    Employee("Chuck", 1800),
    Employee("Samantha", 1800),
    Employee("Roberto", 2100),
    Employee("Dave", 2200),
    Employee("Tina", 2300),
    Employee("Ringo", 1900),
]

for e in employees:
    if e.name == "Vera":
        print(f"{e.name}, ${e.salary}, Manager")
    if e.name == "Chuck" or e.name == "Samantha":
        print(f"{e.name}, ${e.salary}, Attendant")
    if e.name == "Roberto":
        print(f"{e.name}, ${e.salary}, Cook")
    if e.name == "Dave" or e.name == "Tina" or e.name == "Ringo":
        print(f"{e.name}, ${e.salary}, Car repair")
```

Abstraction as a Solution

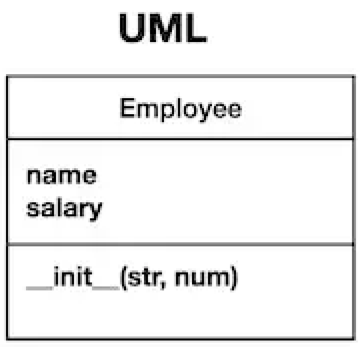
- We use the OOP abstraction technique (classes and objects) and can solve this issue.
- The following UML class diagram shows the abstraction of the Employee.

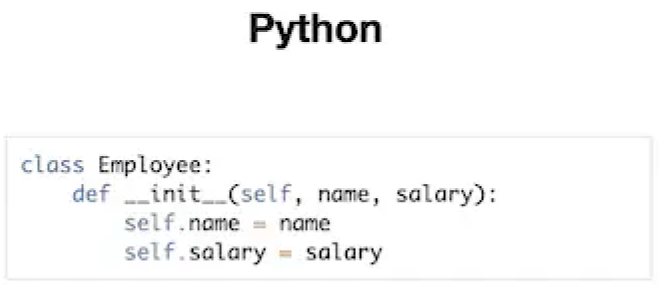
- The UML has three sections: name, fields (Some people use the terminology attributes), and methods.



UML To Python Code

- We can translate the UML class diagram into any OOP code, including Python.



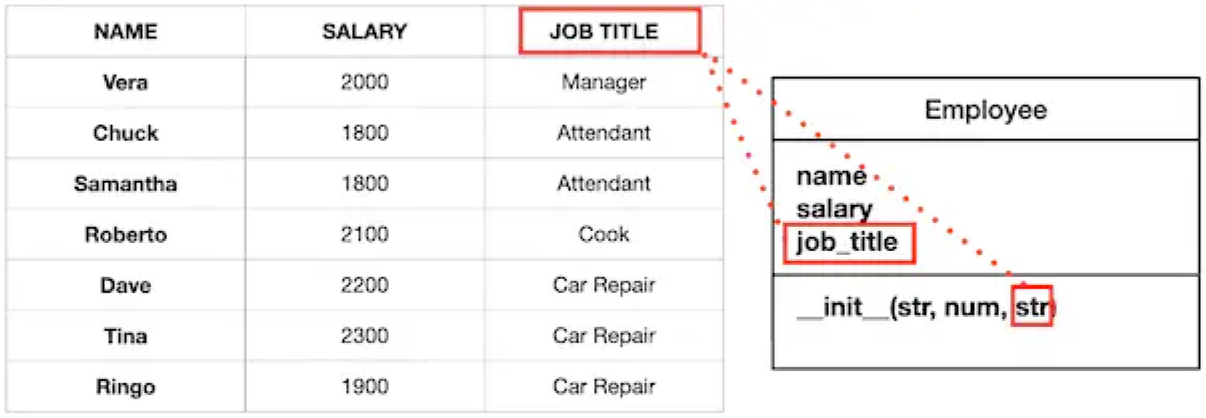


```
class Employee:
    def __init__(self, name, salary):
        self.name = name
        self.salary = salary
```

Refactor design first

- We can manage the complexity with abstractions.
- Instead of updating code directly to meet requirements, we refactor our design (in this case, the UML class diagram).

- By analyzing a new requirement and our current design, we can identify that we need to add a new field to our class design.



NAME	SALARY	JOB TITLE
Vera	2000	Manager
Chuck	1800	Attendant
Samantha	1800	Attendant
Roberto	2100	Cook
Dave	2200	Car Repair
Tina	2300	Car Repair
Ringo	1900	Car Repair

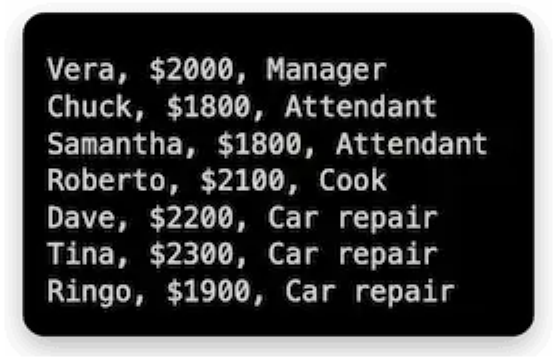
Then, refactor the Code

```python
class Employee:
    def __init__(self, name, salary, job_title):
        self.name = name
        self.salary = salary
        self.job_title = job_title

employees = [
    Employee("Vera", 2000, "Manager"),
    Employee("Chuck", 1800, "Attendant"),
    Employee("Samantha", 1800, "Attendant"),
    Employee("Roberto", 2100, "Cook"),
    Employee("Dave", 2200, "Car Repair"),
    Employee("Tina", 2300, "Car Repair"),
    Employee("Ringo", 1900, "Car Repair")
]
```

Clean Implementation

- With a slight modification, we can show the same results.

```python
for e in employees:
    print(f"{e.name}, ${e.salary}, {e.job_title}")
```

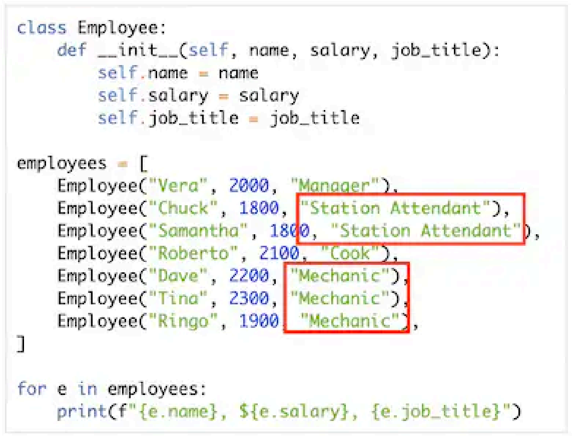


```
Vera, $2000, Manager
Chuck, $1800, Attendant
Samantha, $1800, Attendant
Roberto, $2100, Cook
Dave, $2200, Car repair
Tina, $2300, Car repair
Ringo, $1900, Car repair
```

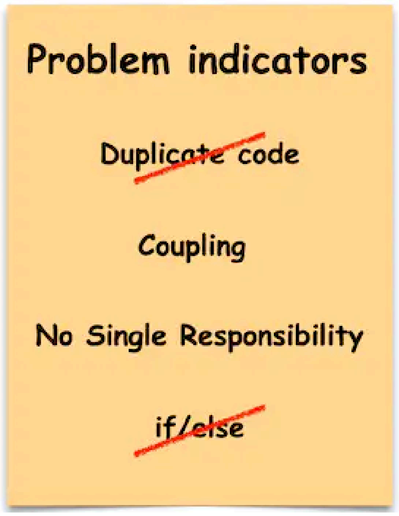
Code Smell - Duplicate Code

- However, we can sense another code smell.
- We find that job titles are duplicated in many places.

- This means we should modify multiple places with a single change.
- How can we solve this problem in an OOP way?



```python
class Employee:
    def __init__(self, name, salary, job_title):
        self.name = name
        self.salary = salary
        self.job_title = job_title

employees = [
    Employee("Vera", 2000, "Manager"),
    Employee("Chuck", 1800, "Station Attendant"),
    Employee("Samantha", 1800, "Station Attendant"),
    Employee("Roberto", 2100, "Cook"),
    Employee("Dave", 2200, "Mechanic"),
    Employee("Tina", 2300, "Mechanic"),
    Employee("Ringo", 1900, "Mechanic"),
]

for e in employees:
    print(f"{e.name}, ${e.salary}, {e.job_title}")
```



Problem indicators

~~Duplicate code~~

Coupling

No Single Responsibility

~~if/else~~

Lessons Learned

- OOP's abstraction allows us to manage complexity from changes.
- Whenever we need to change our code, we do not change it in a hurry.

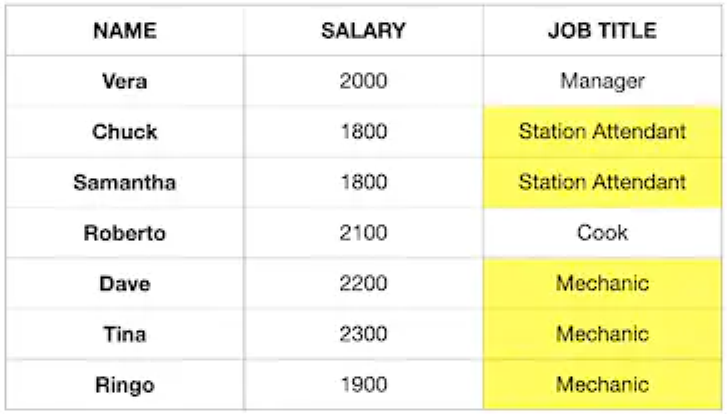
- Instead, we step back, consider the class design to accommodate the change, modify the class design, and then update the code accordingly.
- **Key Process**: Design → Code, not Code → Design
    - This process matches with vibe coding.

Duplication is a bad code smell.
Inheritance is a solution.

DRP Rule

- We have a software design rule, **DRP (Don't Repeat Principle)**.
- Duplication means multiple impacts from a change.

- If Mr. Star changes the job title, it will impact multiple places in the code.

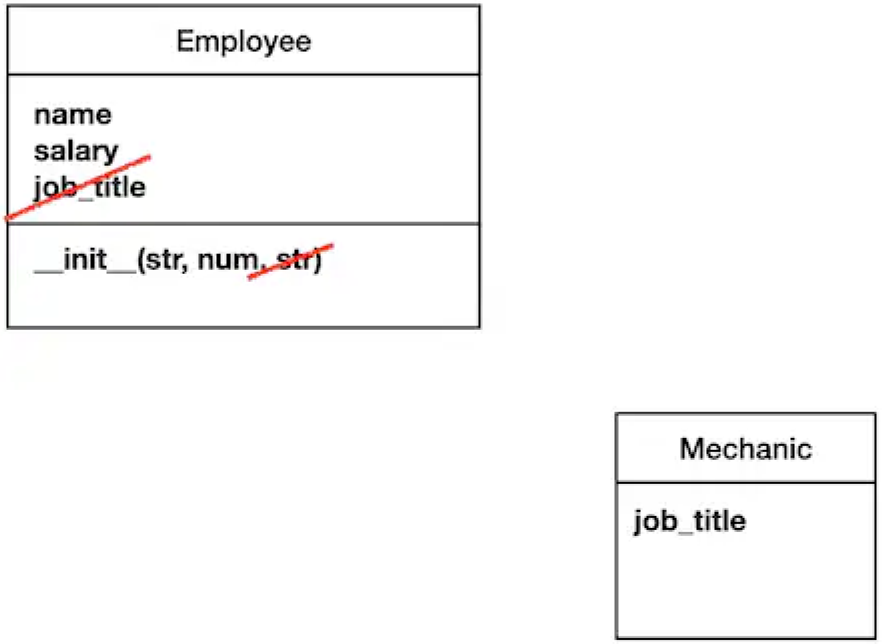


NAME	SALARY	JOB TITLE
Vera	2000	Manager
Chuck	1800	Station Attendant
Samantha	1800	Station Attendant
Roberto	2100	Cook
Dave	2200	Mechanic
Tina	2300	Mechanic
Ringo	1900	Mechanic

Identifying Duplication

- The first step to solving this issue is identifying the duplication.
- In this example, job_title is duplicated.

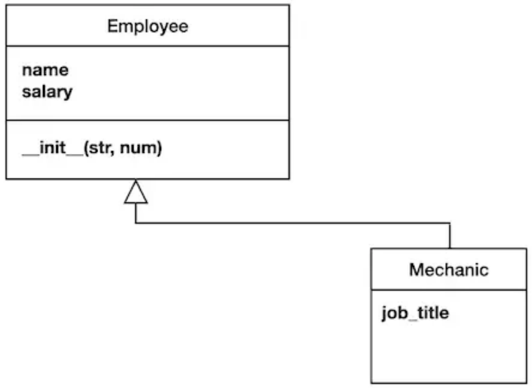
- We can solve this duplication issue by creating new job classes.

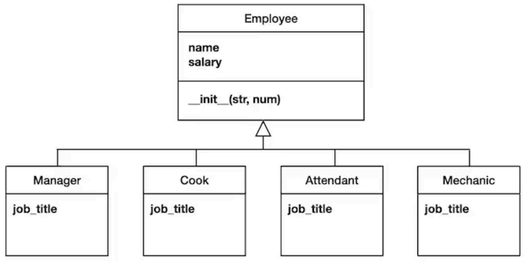


Inheritance as the solution

- The Mechanic class must use the Employee field's name and salary.
- The easiest way is to make the Mechanic class a subclass of the Employee class.

- We can extend the employee class to include other courses, such as manager, cook, or attendant.

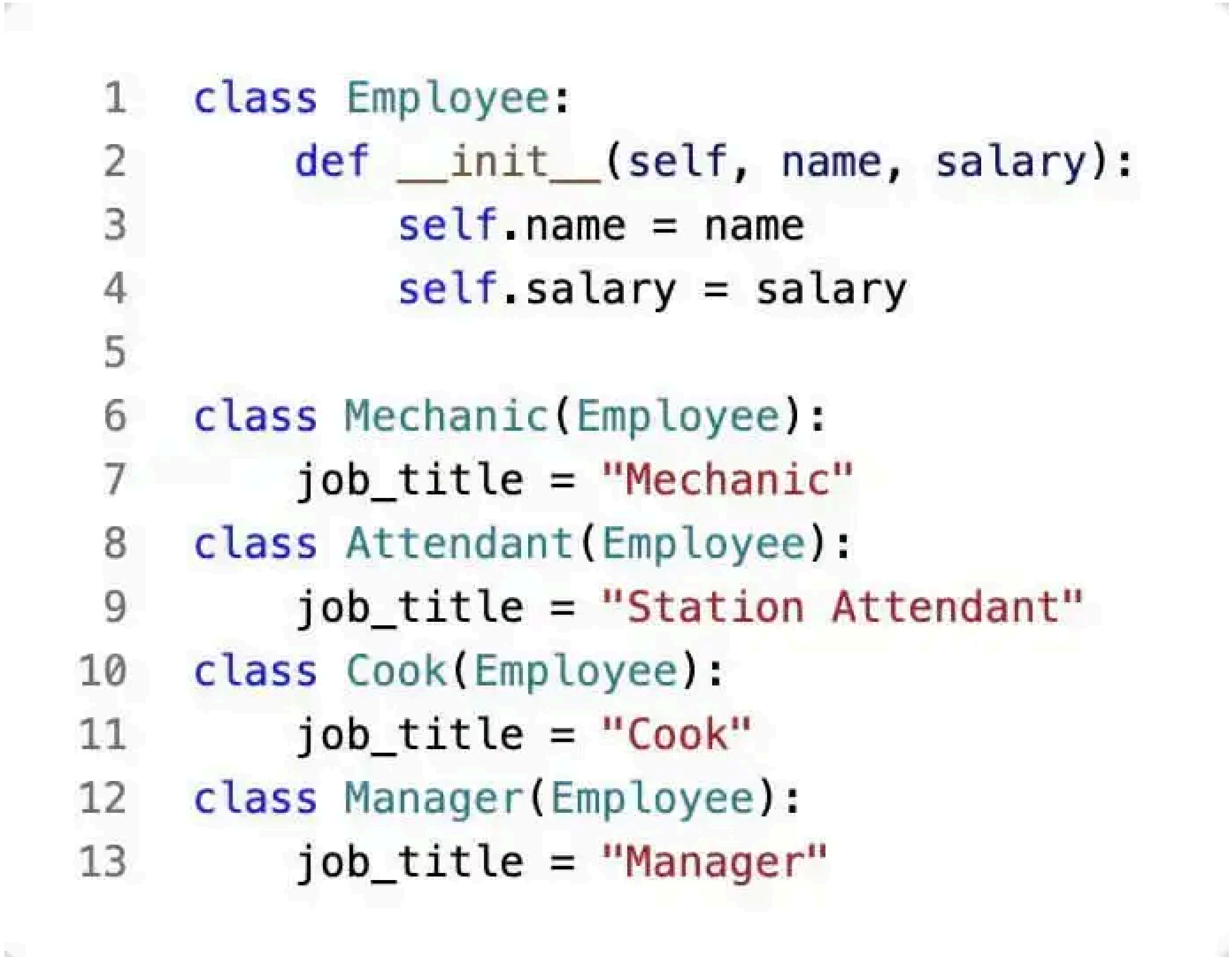




Static Fields

- The job title of the class is the same for all the objects.
- For example, the job title will always be "Mechanic" for the Mechanic class.
- We can make the job title a **static field**.

Implementation



```python
class Employee:
    def __init__(self, name, salary):
        self.name = name
        self.salary = salary

class Mechanic(Employee):
    job_title = "Mechanic"
class Attendant(Employee):
    job_title = "Station Attendant"
class Cook(Employee):
    job_title = "Cook"
class Manager(Employee):
    job_title = "Manager"
```

Inheritance as the solution to duplication

- We used the OOP technique **inheritance** to remove duplicate code smell.
- The `rule of three` applies here: when you see the duplication three times, it's a code smell to refactor using inheritance.

Benefits of using inheritance

- With inheritance, we can reuse existing code.
- We only extend (add) or override (revise) the missing features.

Tool box

✓ Objects & Classes

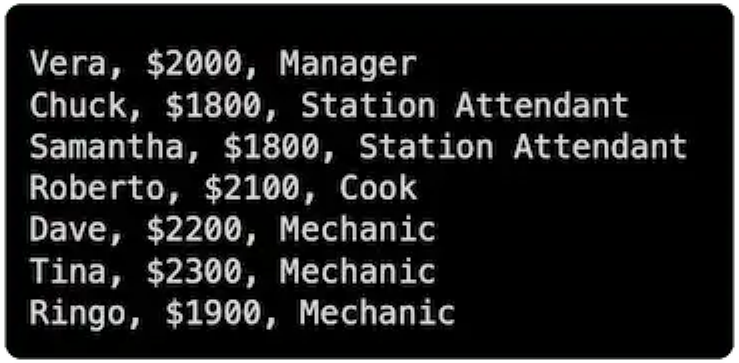
✓ Inheritance

Implementation

```
employees = [
    Manager("Vera", 2000),
    Attendant("Chuck", 1800),
    Attendant("Samantha", 1800),
    Cook("Roberto", 2100),
    Mechanic("Dave", 2200),
    Mechanic("Tina", 2300),
    Mechanic("Ringo", 1900),
]

for e in employees:
    print(f"{e.name}, ${e.salary}, {e.job_title}")
```

Problem solved

- **Success**: We can change **only one place** to change the job title!

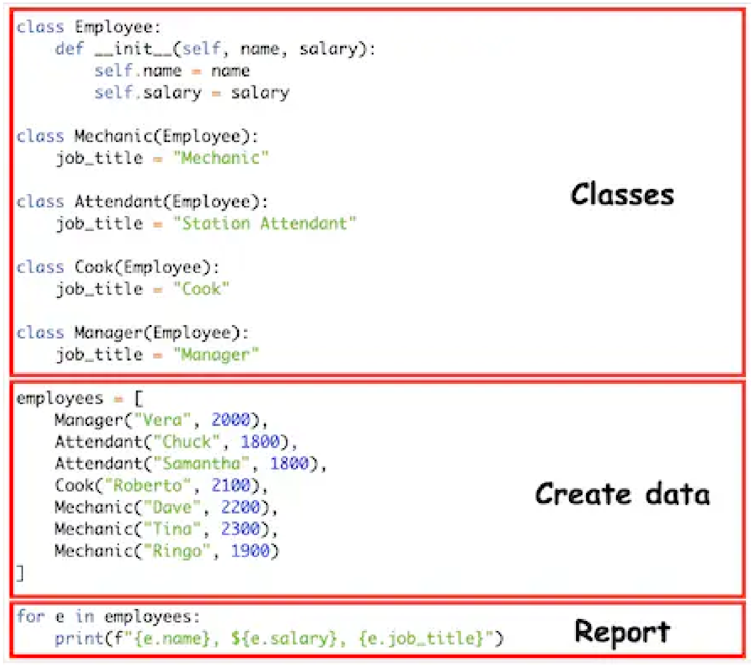


```
Vera, $2000, Manager
Chuck, $1800, Station Attendant
Samantha, $1800, Station Attendant
Roberto, $2100, Cook
Dave, $2200, Mechanic
Tina, $2300, Mechanic
Ringo, $1900, Mechanic
```

Separation of Concerns

- In the code, we can separate our concerns into three sections: classes, creating data (population), and reporting.

- Then, we can separate into different modules and interfaces to manage complexity from changes.



```
class Employee:
    def __init__(self, name, salary):
        self.name = name
        self.salary = salary

class Mechanic(Employee):
    job_title = "Mechanic"

class Attendant(Employee):
    job_title = "Station Attendant"

class Cook(Employee):
    job_title = "Cook"

class Manager(Employee):
    job_title = "Manager"
```

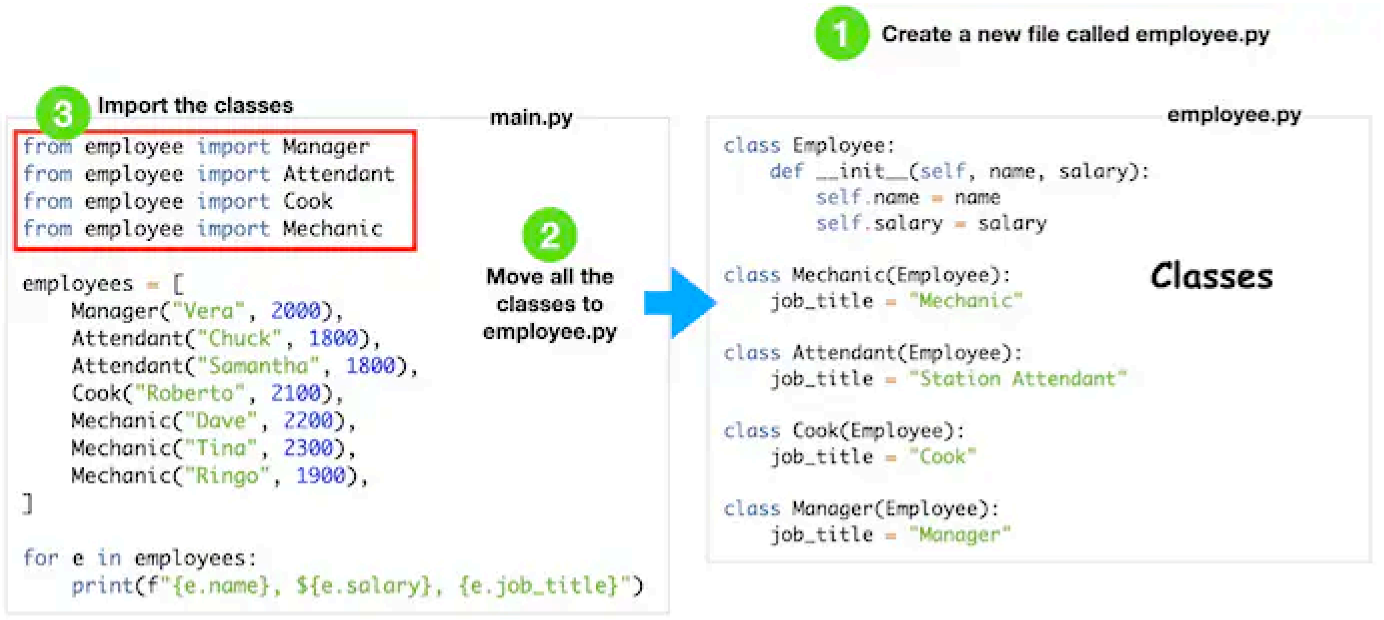
```
employees = [
    Manager("Vera", 2000),
    Attendant("Chuck", 1800),
    Attendant("Samantha", 1800),
    Cook("Roberto", 2100),
    Mechanic("Dave", 2200),
    Mechanic("Tina", 2300),
    Mechanic("Ringo", 1900)
]
```

```
for e in employees:
    print(f"{e.name}, ${e.salary}, {e.job_title}")
```

Module Organization

- We create `employee.py` for the 'Classes' section.
- We create `main.py` for the 'Create data' and 'Report' sections.

- The main.py uses the `from employee import ...` to import classes in the employee.py module.

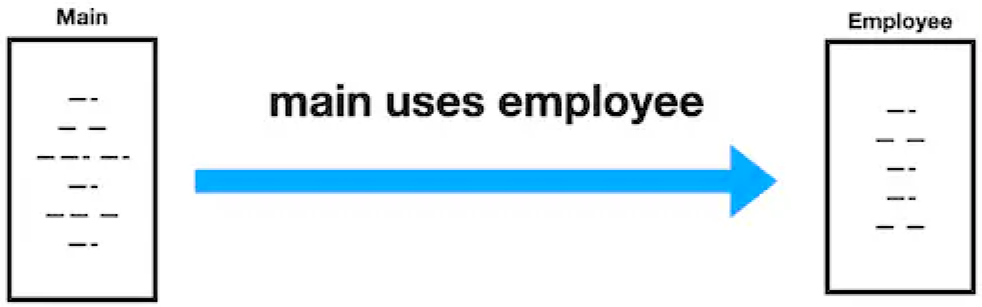

1 Create a new file called employee.py
3 Import the classes
main.py
employee.py
2 Move all the classes to employee.py
Classes

```
from employee import Manager
from employee import Attendant
from employee import Cook
from employee import Mechanic

employees = [
    Manager("Vera", 2000),
    Attendant("Chuck", 1800),
    Attendant("Samantha", 1800),
    Cook("Roberto", 2100),
    Mechanic("Dave", 2200),
    Mechanic("Tina", 2300),
    Mechanic("Ringo", 1900),
]

for e in employees:
    print(f"{e.name}, ${e.salary}, {e.job_title}")
```

```
class Employee:
    def __init__(self, name, salary):
        self.name = name
        self.salary = salary

class Mechanic(Employee):
    job_title = "Mechanic"

class Attendant(Employee):
    job_title = "Station Attendant"

class Cook(Employee):
    job_title = "Cook"

class Manager(Employee):
    job_title = "Manager"
```

Module Relationship

- The `main.py` module **uses** the `employee.py` module, and an arrow is used to describe this relationship in UML.



Lessons Learned

- We should isolate the duplicated fields or methods when we see duplications.
- We can isolate them using **inheritance**.

- As we refactor, we should organize code to **separate concerns** so that we can make modules and interfaces.
- **Key Principle**: Inheritance eliminates duplication and promotes code reuse.