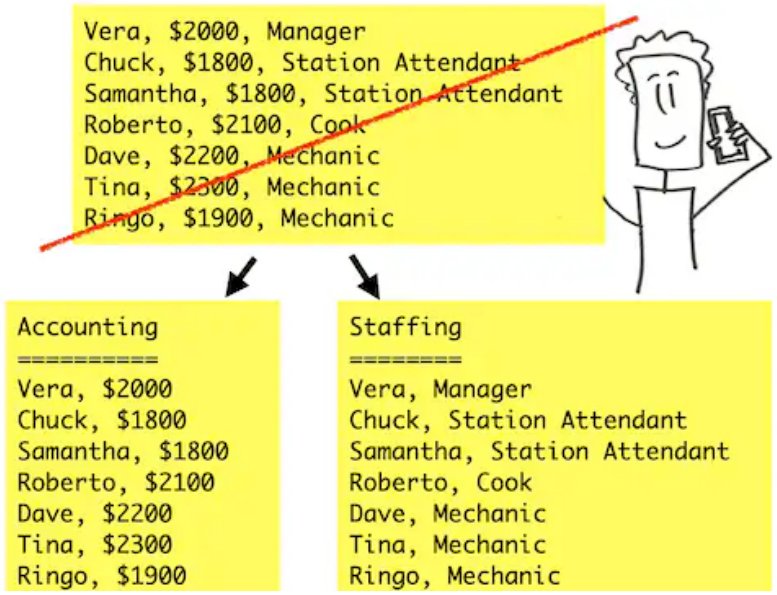
Encapsulation and Dependency Injection

New requirements

- Mr. Star likes the new program so much that he wants to create a new report.

- This time, he wants to create two different reports: one for Accounting and another for Staffing.



Requirements Version 4

```
Epic requirement:
As an "employer,"
I want to "generate reports"
so that "I can manage my employees."

Sub requirement 1: As an "account manager,"
I want to "have an Accounting report including name and salary,"
so that "I can track monthly salary payment."

Sub requirement 2: As a "staff manager,"
I want to "have a Staffing report including name and job title."
so that "I can track my staff."
```

Two Methods Solution

To solve this problem, we need to make two methods.

```
def print_accounting_report():
```

```
Accounting
==========
Vera, $2000
Chuck, $1800
Samantha, $1800
Roberto, $2100
Dave, $2200
Tina, $2300
Ringo, $1900
```

```
def print_staffing_report():
```

```
Staffing
========
Vera, Manager
Chuck, Station Attendant
Samantha, Station Attendant
Roberto, Cook
Dave, Mechanic
Tina, Mechanic
Ringo, Mechanic
```

Implementation

```
1  def print_accounting_report():
2      print("Accounting")
3      print("==========")
4      for e in employees:
5          print(f"{e.name}, ${e.salary}")
6
7  def print_staffing_report():
8      print("Staffing")
9      print("========")
10     for e in employees:
11         print(f"{e.name}, {e.job_title}")
12
13 print_accounting_report()
14 print()  # empty line
15 print_staffing_report()
```

Problem: Name Collision

- We hire Chuck Rainey, who is a new Station Attendant.
- We cannot distinguish this Chuck from Chuck the mechanic.

Chuck Rainey

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
Chuck	1800	Mechanic

Another Chuck???

Spreadsheet Update Needed

- We need to change the spreadsheet to include first and last names.

FIRST NAME	LAST NAME	SALARY	JOB TITLE
Vera	Schmidt	2000	Manager
Chuck	Norris	1800	Station Attendant
Samantha	Carrington	1800	Station Attendant
Roberto	Jacketti	2100	Cook
Dave	Dreißig	2200	Mechanic
Tina	River	2300	Mechanic
Ringo	Rama	1900	Mechanic
Chuck	Rainey	1800	Mechanic

Requirements Version 5

We also need to change the requirements.

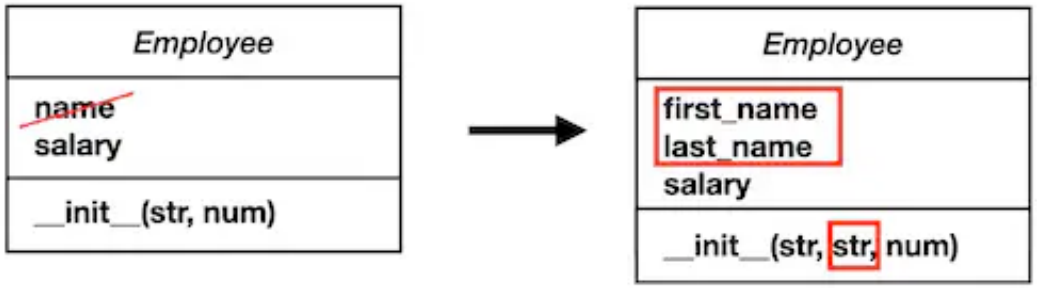
```
Epic requirement:
As an "employer,"
I want to "generate reports"
so that "I can manage my employees."

Sub requirement 1: As an "account manager,"
I want to "have an Accounting report including first name, last name, and salary."
So that "I can track monthly salary payment."

Sub requirement 2: As a "staff manager,"
I want to "have a Staffing report including first name, last name, and job title."
So that "I can track my staff."
```

Class refactoring

- The next step is to refactor our class design.



Code refactoring

- Finally, we refactor the code.
- We need to change the Employee class.

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

Implementation

```
employees = [
    Manager("Schmidt", "Vera", 2000),
    Attendant("Norris", "Chuck", 1800),
    Attendant("Carrington", "Samantha", 1800),
    Cook("Jacketti", "Roberto", 2100),
    Mechanic("Dreisig", "Dave", 2200),
    Mechanic("River", "Tina", 2300),
    Mechanic("Rama", "Ringo", 1900),
    Attendant("Rainey", "Chuck", 1800),
]

def print_accounting_report():
    print("Accounting")
    print("==========")
    for e in employees:
        print(f"{e.first_name} {e.last_name}, ${e.salary}")

def print_staffing_report():
    print("Staffing")
    print("========")
```

Lessons Learned

- Requirements evolve as business needs change
- Good design makes it easier to accommodate changes
- Separate reporting logic from data structure

- Always plan for future extensibility
- **Key Insight**: Design for change, not just current requirements.

We need to hide information: Encapsulation.

What if we need a different format?

- What if someone needs to print out the names in a different format?

- We can update the print code twice to meet this request.

```
def print_accounting_report():
    print("Accounting")
    print("==========")
    for e in employees:
        print(f"{e.first_name} {e.last_name}, ${e.salary}")

def print_staffing_report():
    print("Staffing")
    print("========")
    for e in employees:
        print(f"{e.first_name} {e.last_name}, ${e.job_title}")
```

Code Smell

- However, we need to change **two places**, not one, in this case.
- Also, we see that the print methods should know the details (fields) of the Employee class.
    - It should be known that the Employee class has first_name and last_name.
    - It means it can change these fields.

- So, if there is a change in the field name, the print code will be impacted and should be updated.
- So we detect **two code smells**.

1: Duplicate code

2: Implementation details

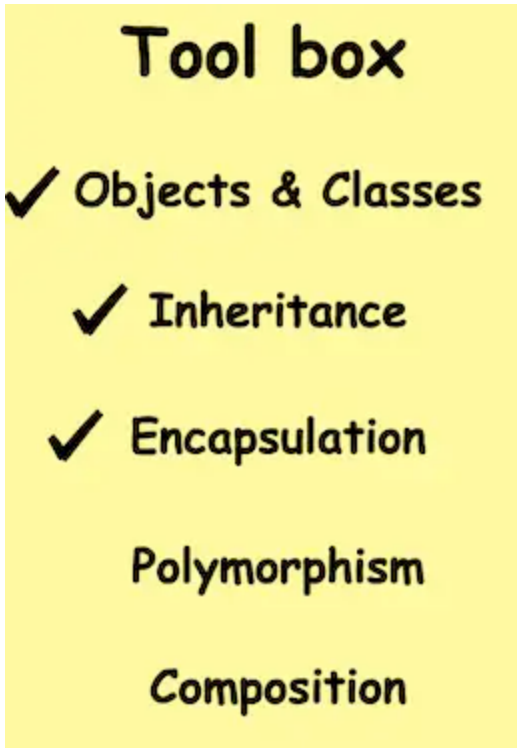
The Root Problem

- The problem is that the Employee class fields can be accessed outside.
- It's only the **Employee's job** to manage the fields such as first_name or last_name.

- So, the Employee class should **hide** them.
- At the same time, it should provide a way to access the fields through **methods**.

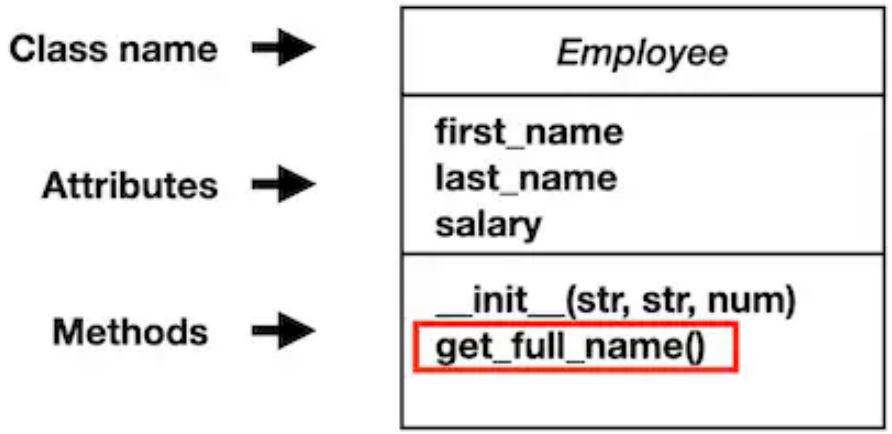
Encapsulation

- We use Encapsulation (Information hiding) to address this issue.
- Key Principle: Hide implementation details, expose only what's necessary.

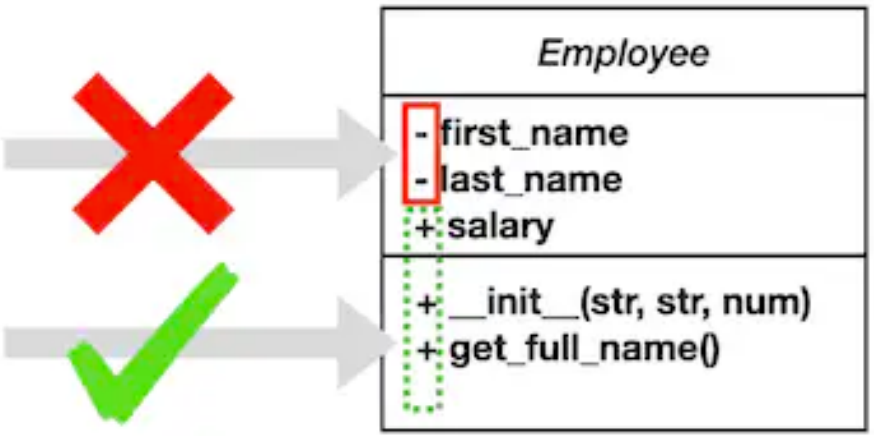


UML Design Changes

- We add a method `get_full_name()`.
- We hide the two fields by making them **private** (-).



- Instead, the `get_full_name()` method should be **public** (+) so that any entity that needs the name information can use it.



Refactor code

- We can refactor the Employee class.

```
class Employee:
    def __init__(self, last_name, first_name, salary):
        self.first_name = first_name
        self.last_name = last_name
        self.salary = salary
    def get_full_name(self):
        return f"{self.last_name},{self.first_name}"
```

Python pitfall

- Python does not have public or private modifiers, so we should manage the visibility independently.
- Python does not support private modifiers, so we prepend `_` to indicate private fields.

Python private fields convention

- This may make Python harder to debug than the languages that provide visibility features at a language level.

```
class Employee:
    def __init__(self, last_name, first_name, salary):
        self._first_name = first_name  # <-- make it private
        self._last_name = last_name    # <-- make it private
        self.salary = salary
    def get_full_name(self):
        return f"{self._last_name},{self._first_name}"
```

Implementation

- We can get the same results with a modification in the driver code.
- The two print methods can use the `get_full_name()` method to print the name **without knowing the implementation details** of the Employee class.

- We get the same results as before.
- However, the Employee fields are isolated from external change.

```
1  def print_accounting_report():
2      print("Accounting")
3      print("==========")
4      for e in employees:
5          print(f"{e.get_full_name()}, ${e.salary}")
6
7  def print_staffing_report():
8      print("Staffing")
9      print("========")
10     for e in employees:
11         print(f"{e.get_full_name()}, {e.job_title}")
```

Lessons Learned

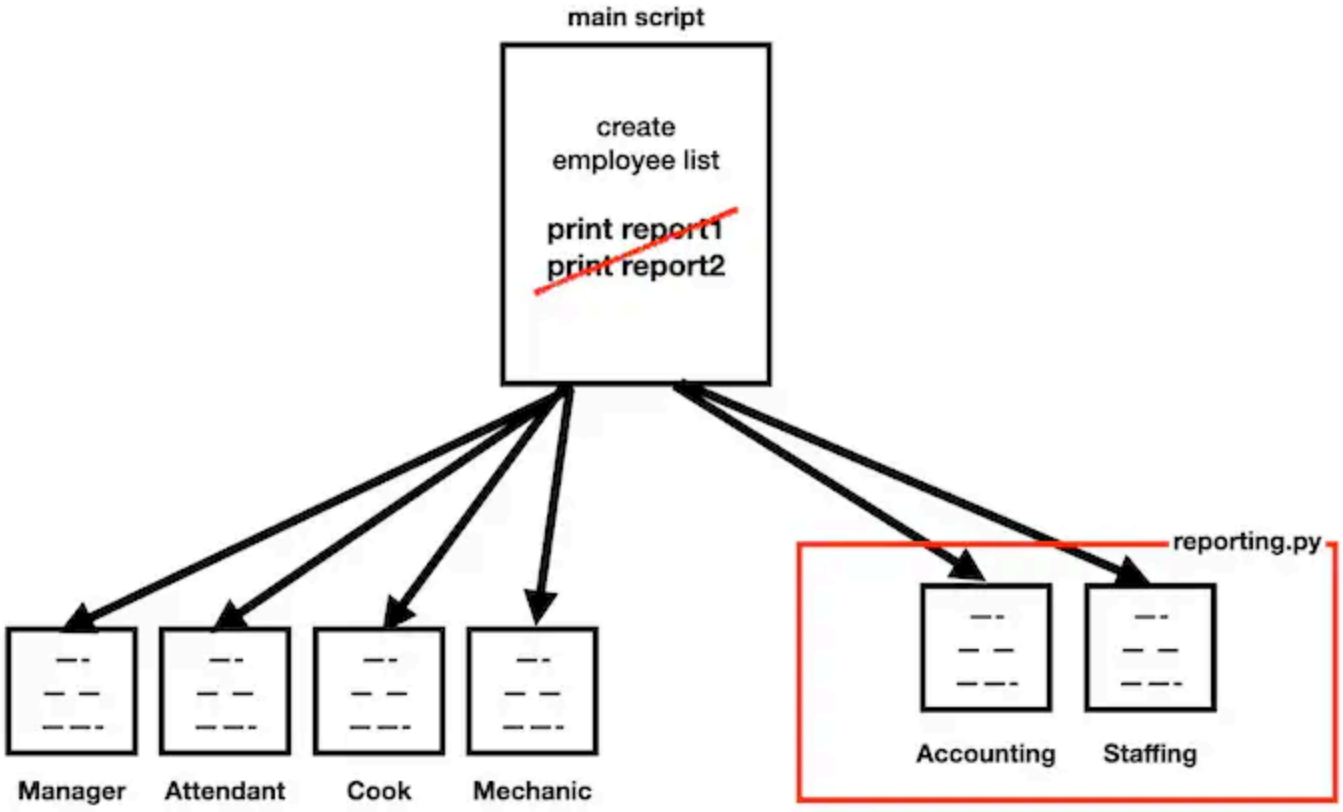
- A class should **hide private fields or methods**.
- The class should also **provide public methods** that other entities can use.
    - We call these public methods `interfaces`.

- We can manage complexity with encapsulation.
- **Benefits** of encapsulation include:
    - Single point of change for name formatting
    - Implementation details are hidden
    - The interface is stable and clean

Refactoring for Better Organization

- We find that the two reporting methods are beginning to get larger.

- So, we make two classes for reporting.



Creating Report Module

```
class AccountingReport:
    def print_accounting_report(self):
        print("Accounting")
        print("==========")
        # !!! Direct access to main.py's employees
        for e in employees:
            print(f"{e.get_full_name()}, ${e.salary}")

class StaffingReport:
    def print_staffing_report(self):
        print("Staffing")
        print("========")
        for e in employees:
            print(f"{e.get_full_name()}, {e.job_title}")
```

Refactor main.py

- We need to refactor the main.py.

```
from employee import Manager, Attendant, Cook, Mechanic
from reporting import AccountingReport, StaffingReport

employees = ... # report classes access it
AccountingReport().print_accounting_report()
print() # empty line
StaffingReport().print_staffing_report()
```

Coupling

Dependency Issue

- However, this code will **crash**.
- The module needs the object `employees` in the main.py, so it depends on the main.py.
- The main.py, in turn, depends on the `reporting` module.

- We call this situation **"circular dependency."**

reporting.py

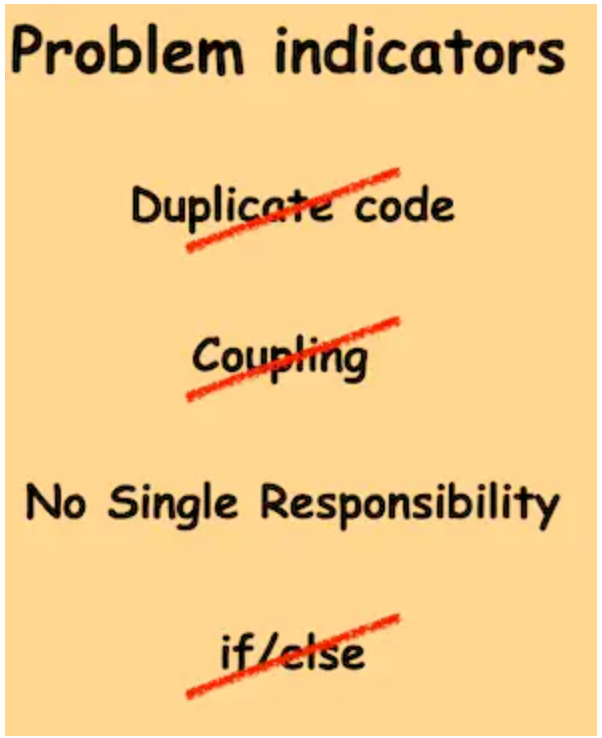
```
class AccountingReport:
    def print_accounting_report(self):
        print("Accounting")
        print("==========")
        for e in employees:
            print(f"{e.get_full_name()}, $
{e.salary}")
```

reporting.py

```
class StaffingReport:
    def print_staffing_report(self):
        print("Staffing")
        print("========")
        for e in employees:
            print(f"{e.get_full_name()},
{e.job_title}")
```

The Cause of the Problem - Coupling

- This issue is because the main.py and report.py are **coupled**.
- Coupled means one change impacts all the coupled entities.

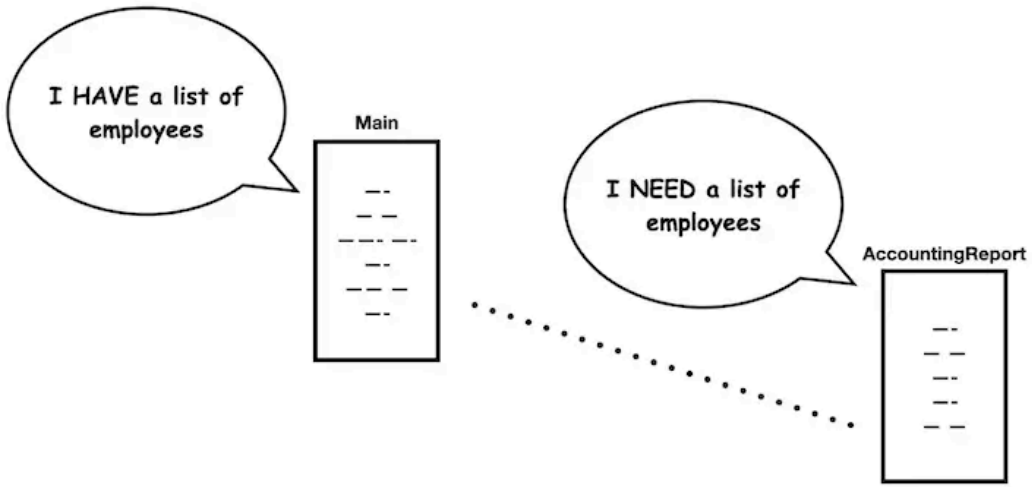


Warning: Coupling is a notorious code smell!

Solution - Dependency Injection

- This issue can be resolved by **dependency injection**.
- The main.py has the object in reporting.py, and the reporting.py tries to access the object **directly**.

- We can resolve the issue by **injecting the dependency** (employees object) into the reporting.py classes whenever necessary.



Refactored Report Classes

- Now the dependency is injected.

```python
class AccountingReport:
    def __init__(self, emp_list):
        # !!! Dependency is injected from outside
        # !!! This class doesn't access the main.py any more
        self._emp_list = emp_list

    def print_accounting_report(self):
        print("Accounting")
        print("==========")
        for e in self._emp_list:
            print(f"{e.get_full_name()}, ${e.salary}")

class StaffingReport:
    def __init__(self, emp_list):
        self._emp_list = emp_list

    def print_staffing_report(self):
        print("Staffing")
        print("========")
        for e in self._emp_list:
            print(f"{e.get_full_name()}, {e.job_title}")
```

Refactored main

- The main injects the employees into the report classes.

```
from reporting import AccountingReport, StaffingReport

employees = ...

accounting_report = AccountingReport(employees)  # injected dependency
accounting_report.print_accounting_report()
print()
staffing_report = StaffingReport(employees)
staffing_report.print_staffing_report()
```

Lessons Learned

- **Coupling** is one of the worst code smells, so we should remove couplings as much as possible.
- We can remove some couplings using **dependency injection**.

- When we inject a dependency into any object, the object does not need to access the dependency, so coupling is removed.
- Key Principle: Inject dependencies instead of accessing them directly.

Coupling and Dependency Injection in UML

- In UML, we use a **solid line** to express the coupling (they know each other).
- When we remove the coupling using dependency injection, we use a **dotted line**.