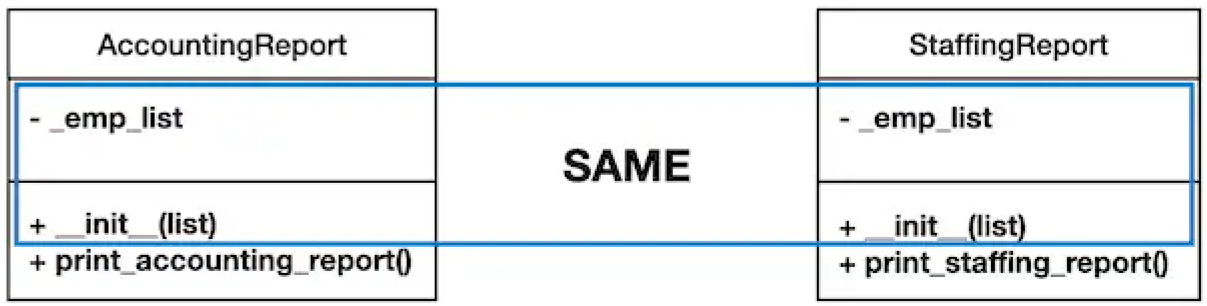
Polymorphism and Composition

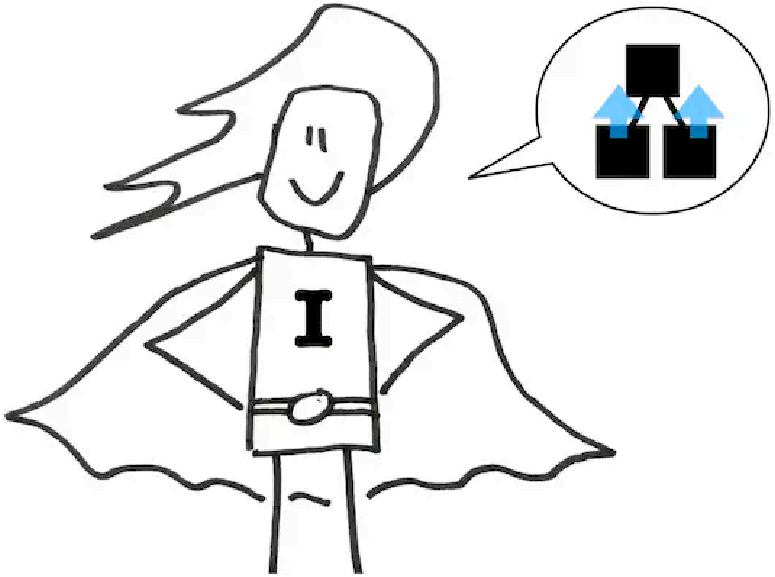
Detecting Duplication

- When we compare the UMLs of the two report classes, we see duplications.

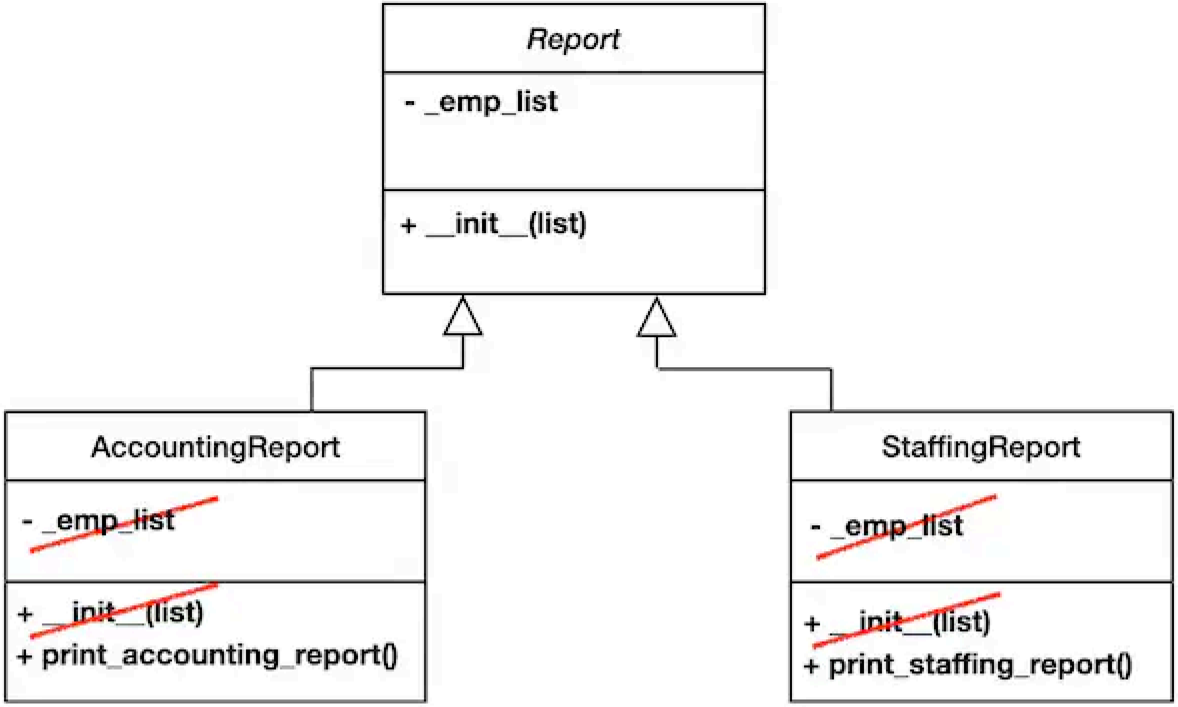


Inheritance as the Solution

- We already know the solution: inheritance.



Refactored Design and Code



```python
class Report:
    def __init__(self, emp_list):
        self._emp_list = emp_list

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

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

Refactor the main.py

- We refactor the main.py to have the reports list and a loop to invoke the print methods.
- Sub-classes have **different method names**, so we should check.

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

employees = ...
reports = [
    AccountingReport(employees),
    StaffingReport(employees),
]

for report in reports:
    if isinstance(report, AccountingReport): # if/else
        report.print_accounting_report()
    else:
        report.print_staffing_report()
```

Code smell

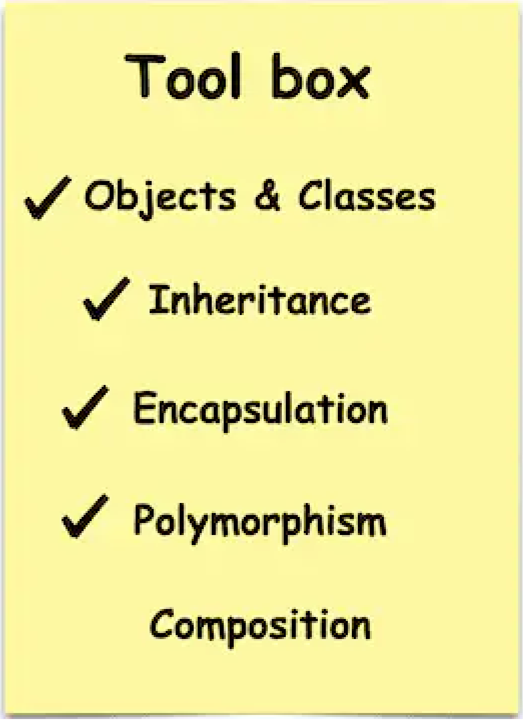
- However, we know it is a sense code smell when we use `isinstance()`.

  ```
  for report in reports:
      if isinstance(report, AccountingReport): # if/else
          report.print_accounting_report()
      else:
          report.print_staffing_report()
  ```

Polymorphism as the Solution

- We can use **Polymorphism** to solve this issue.
- Polymorphism means different subclasses cause different behavior.

- When different objects have the same method name, we can invoke the correct method using Polymorphism.



Tool box

✓ Objects & Classes

✓ Inheritance

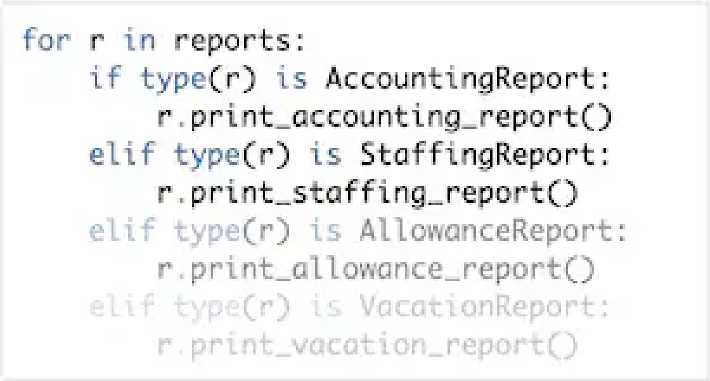
✓ Encapsulation

✓ Polymorphism

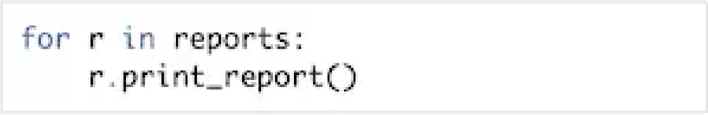
Composition

How Polymorphism Works

- We can use the same method name for the Report classes.



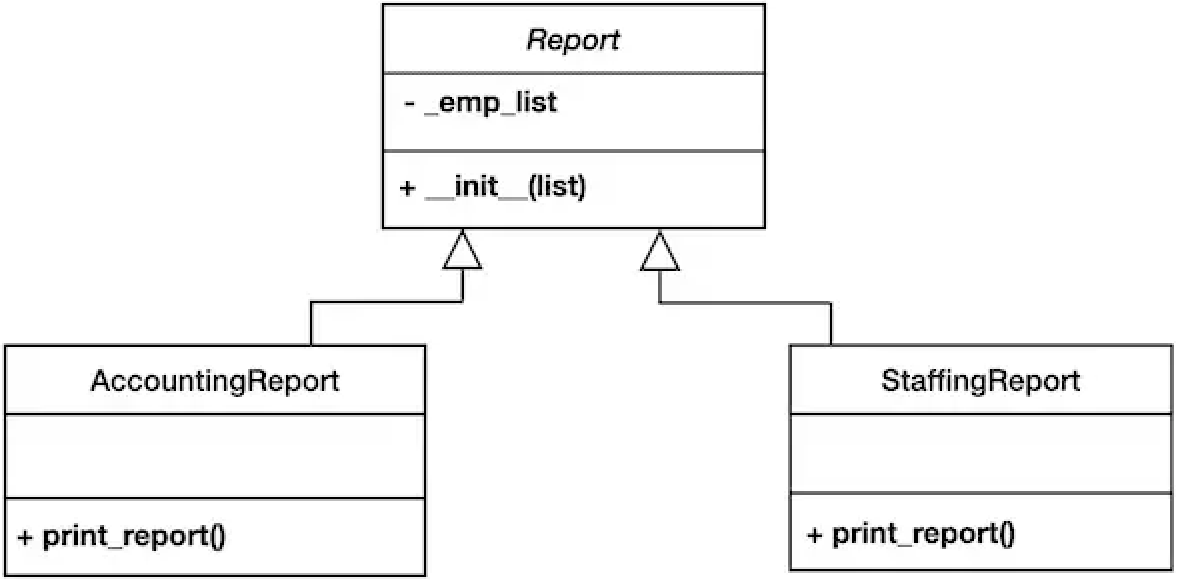
```
for r in reports:
    if type(r) is AccountingReport:
        r.print_accounting_report()
    elif type(r) is StaffingReport:
        r.print_staffing_report()
    elif type(r) is AllowanceReport:
        r.print_allowance_report()
    elif type(r) is VacationReport:
        r.print_vacation_report()
```



```
for r in reports:
    r.print_report()
```

Refactoring the UML

- The first step is to refactor the UML.



Clean Implementation

- Then, we can remove the if statement in the main.py.

```python
# reporting.py
class AccountingReport(Report):
    def print_report(self):
        print("Accounting"); print("==========")
        for e in self._emp_list:
            print(f"{e.get_full_name()}, ${e.salary}")

class StaffingReport(Report):
    def print_report(self):
        print("Staffing"); print("========")
        for e in self._emp_list:
            print(f"{e.get_full_name()}, {e.job_title}")
```

- The main function invokes the `print_report` method from each object.
- No if/else statement is needed.

```python
# main.py
reports = [AccountingReport(employees), StaffingReport(employees)]
for report in reports:
    report.print_report()  # Polymorphism in action!
```

Open-Closed Principle

- Polymorphism has a close relationship with the OCP (Open-Closed Principle) of SOLID.
- The OCP is possible with Polymorphism.

Lessons Learned

- When we use if/else statements with type checking, it is a **code smell**.
- Instead, we use Polymorphism: the **same method name** and invoke the method on the objects.

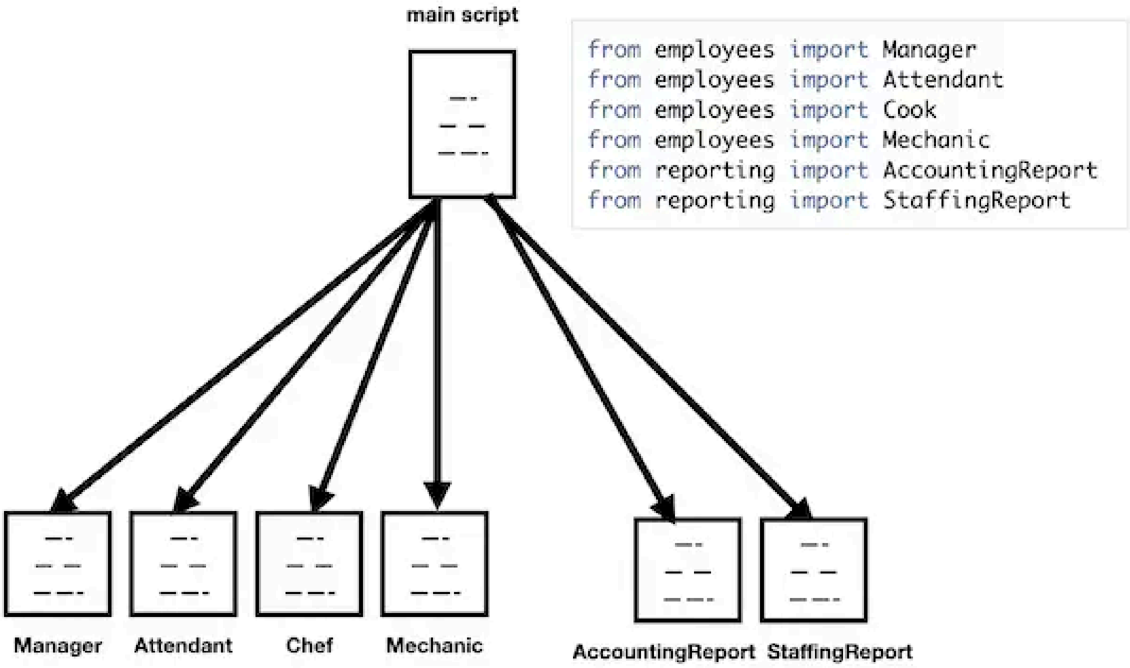
- **Key Benefits**:
- No type checking needed
- Easy to add new report types
- Clean, maintainable code
- Follows Open-Closed Principle

Aggregation for SRP (Single Responsibility Principle)

Software Design with APIE

- With the APIE principle guidelines, our software is well-modularized with clear interfaces.
- The dependency is well-designed and defined; we can use UML to clarify the dependencies with arrows.

- This is a simplified view of our software design.

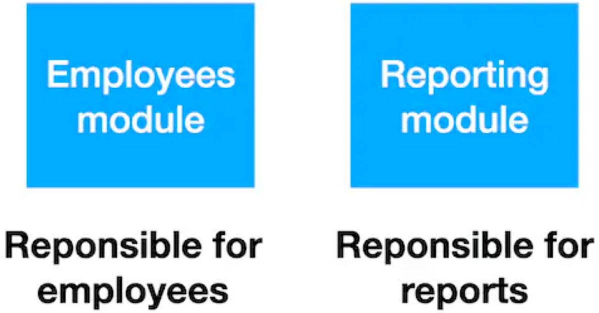


```python
from employees import Manager
from employees import Attendant
from employees import Cook
from employees import Mechanic
from reporting import AccountingReport
from reporting import StaffingReport
```

Single Responsibility Principle (SRP)

- The **Employees module** is responsible only for employees.
- Likewise, **reporting module** is only responsible for reporting.

- This idea is called the SRP (Single Responsibility Principle) of SOLID.



Employees module

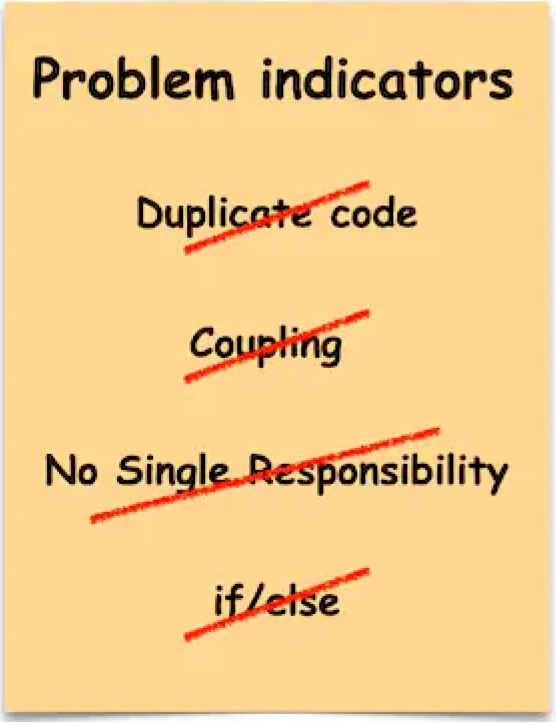
Reponsible for employees

Reporting module

Reponsible for reports

SRP Key Points

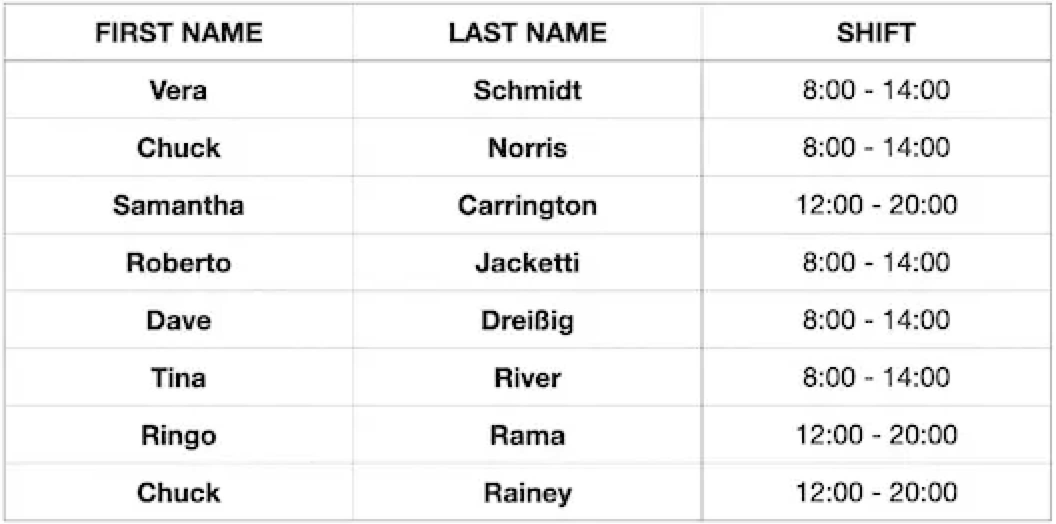
- When a module has responsibility for multiple entities, it is a code smell.
- We should remember that S in SRP stands for Single, not Simple.



New Feature: Working Hours

- Mr.Star wants to track working hours.
- There are two shifts: some employees start from 08:00 am to 2:00 pm (morning shift), while others start from noon to 8:00 pm (afternoon shift).

- Mr. Star also wants a new report: ScheduleReport.



FIRST NAME	LAST NAME	SHIFT
Vera	Schmidt	8:00 - 14:00
Chuck	Norris	8:00 - 14:00
Samantha	Carrington	12:00 - 20:00
Roberto	Jacketti	8:00 - 14:00
Dave	Dreißig	8:00 - 14:00
Tina	River	8:00 - 14:00
Ringo	Rama	12:00 - 20:00
Chuck	Rainey	12:00 - 20:00

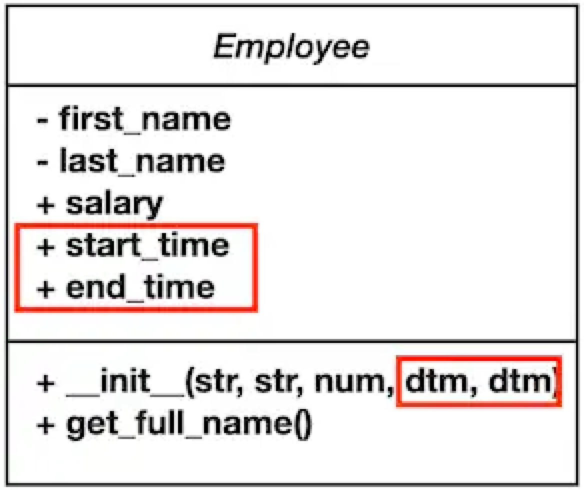
Requirements Version 6

We have new 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."

Sub requirement 3: As a "CEO,"
I want to "have a Schedule report including start time and end time,"
so that "I can track my staff shift."
```

Design Changes

- We need to refactor the UML design, Employee, to include start and end time.

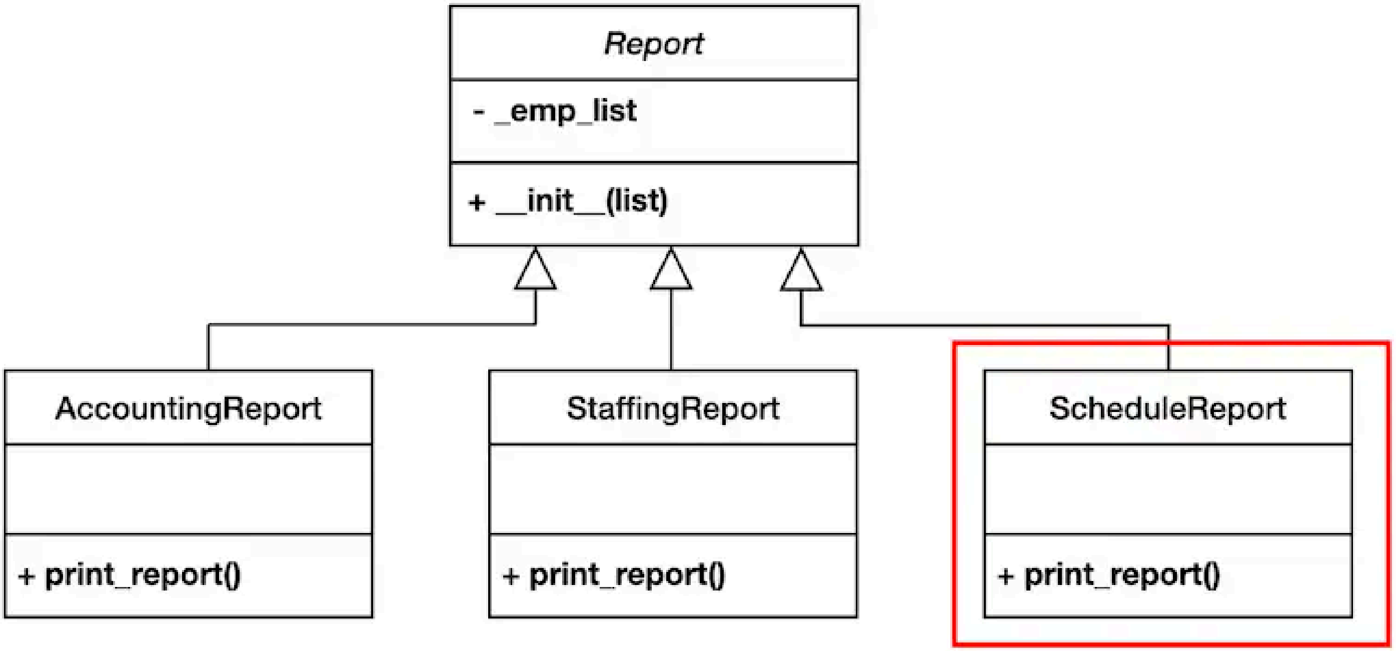


Employee

- first_name
- last_name
+ salary
+ start_time
+ end_time

+ __init__(str, str, num, dtm, dtm)
+ get_full_name()

- For the Reporting class, we can extend the Reporting class.



Implementation

We refactor the Employee class and create the ScheduleReport class.

```python
class Employee:
  def __init__(self, last_name, first_name,
               salary, start_time, end_time):
      self._first_name = first_name
      self._last_name = last_name
      self.salary = salary
      self.start_time = start_time
      self.end_time = end_time

class ScheduleReport(Report):
  def print_report(self):
      print("Schedule"); print("========")
      for e in self._emp_list:
          print(f"{e.get_full_name()}, {e.start_time:%H:%M} to {e.end_time:%H:%M}")
```

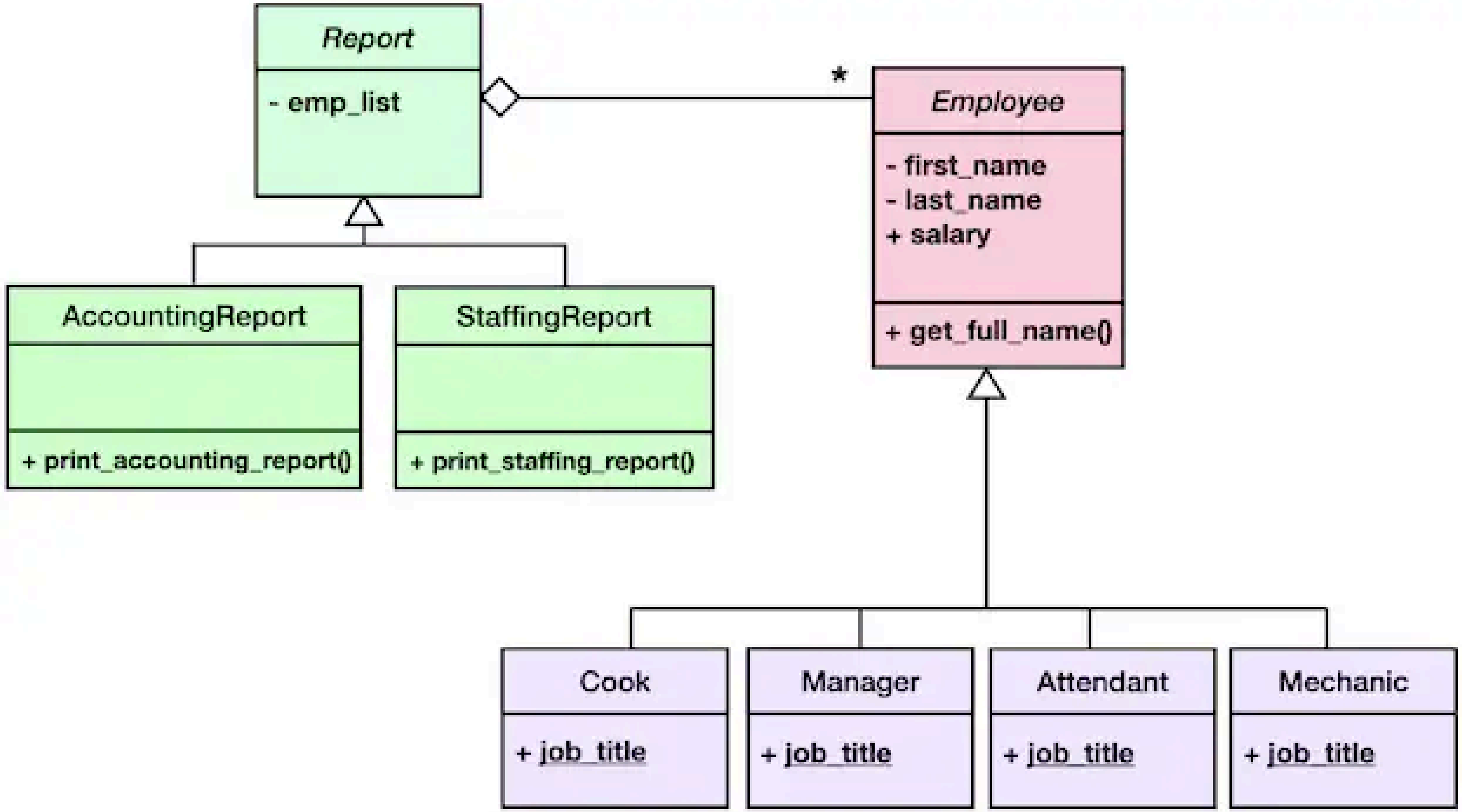
Aggregation Relationship

- We can see that there is an aggregation (ownership) relationship between a report and the employee.
- The report requires the employee to print out the report.

UML

- The UML can express the aggregation relationship using an empty diamond symbol.
- The owner has the **empty diamond** in the Aggregation relationship.
- In this case, the report has the diamond.

Software Design



- The * at the Employee means that a Report can have multiple objects.
- The report owns high-level abstraction (Employee), not a low-level class.
- The report itself is a high-level abstraction with multiple concrete classes.

The direction of a diamond

- In our design, the focus is on the Report.
- The Report needs the Employee class for its reporting, not the other way round.
- So, the Report owns the Employee, and the diamond is on the Report side.

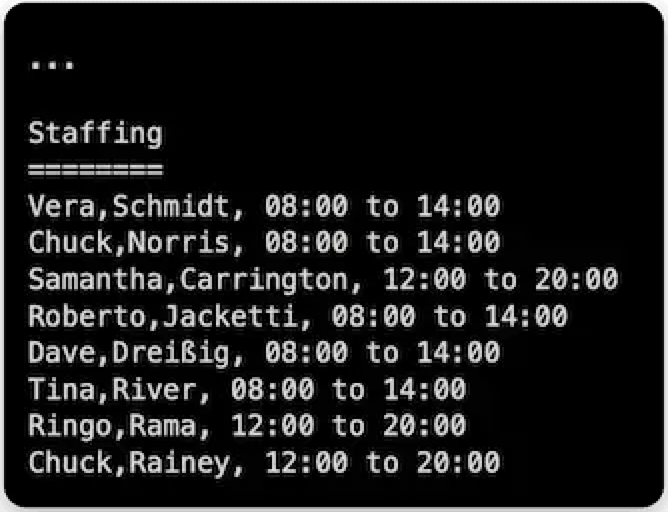
Aggregation and SRP

- With Aggregation, we can make each class responsible for only one task.
- For any changes, the impact of the changes is isolated with Aggregation.
- The empty diamond shows the isolation between the classes.

Easy Extension

- We have refactored into a good design.
- As a result, we need **one line of code** to add a new report in the main.py, without impacting any other code.
- We are managing complexity with software design.

```python
# main.py
import datetime

employees = [
    Manager("Schmidt", "Vera", 2000,
            datetime.time(8, 0), datetime.time(14, 0)),
    Attendant("Norris", "Chuck", 1800,
               datetime.time(12, 0), datetime.time(20, 0)),
    # ... more employees
]

reports = [
    AccountingReport(employees),
    StaffingReport(employees),
    ScheduleReport(employees)  # <- Added !!!
]

for report in reports:
    report.print_report()
    print()
```



```
...

Staffing
========
Vera,Schmidt, 08:00 to 14:00
Chuck,Norris, 08:00 to 14:00
Samantha,Carrington, 12:00 to 20:00
Roberto,Jacketti, 08:00 to 14:00
Dave,Dreißig, 08:00 to 14:00
Tina,River, 08:00 to 14:00
Ringo,Rama, 12:00 to 20:00
Chuck,Rainey, 12:00 to 20:00
```

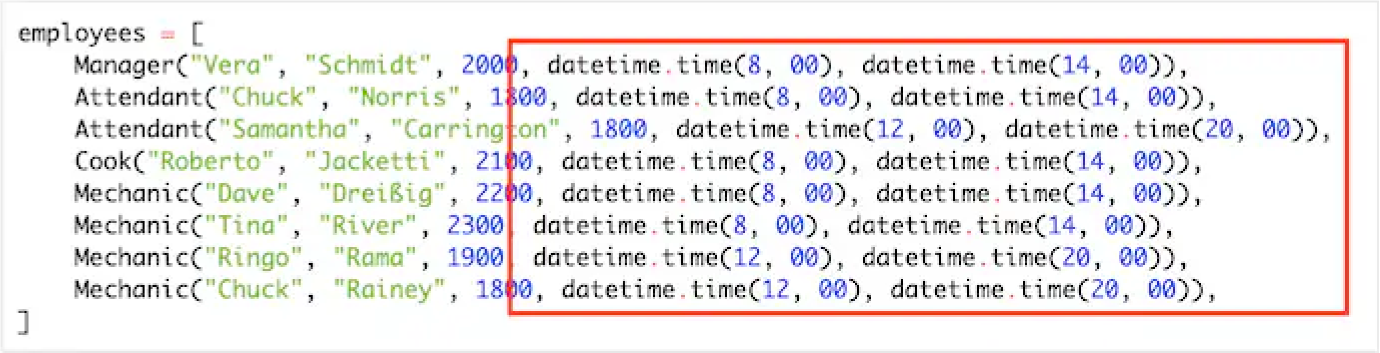
Lessons Learned

- Using software design, we can manage complexity.
- Software design helps to detect code smells and refactor them.
- With good design, we can add new functionality with minimal code changes.

Explosion of Classes and Composition as the solution

Code Smell: Duplication

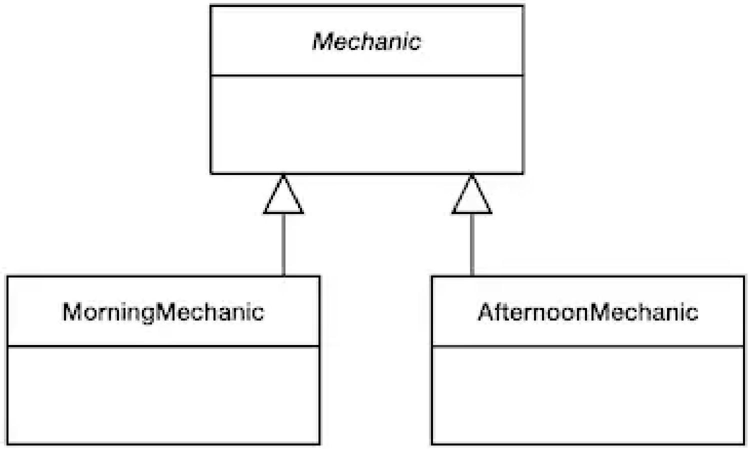
- We have only two time shifts (morning and afternoon), so we **duplicate** the start time and end time.



```
employees = [
    Manager("Vera", "Schmidt", 2000, datetime.time(8, 00), datetime.time(14, 00)),
    Attendant("Chuck", "Norris", 1800, datetime.time(8, 00), datetime.time(14, 00)),
    Attendant("Samantha", "Carrington", 1800, datetime.time(12, 00), datetime.time(20, 00)),
    Cook("Roberto", "Jacketti", 2100, datetime.time(8, 00), datetime.time(14, 00)),
    Mechanic("Dave", "Dreißig", 2200, datetime.time(8, 00), datetime.time(14, 00)),
    Mechanic("Tina", "River", 2300, datetime.time(8, 00), datetime.time(14, 00)),
    Mechanic("Ringo", "Rama", 1900, datetime.time(12, 00), datetime.time(20, 00)),
    Mechanic("Chuck", "Rainey", 1800, datetime.time(12, 00), datetime.time(20, 00)),
]
```

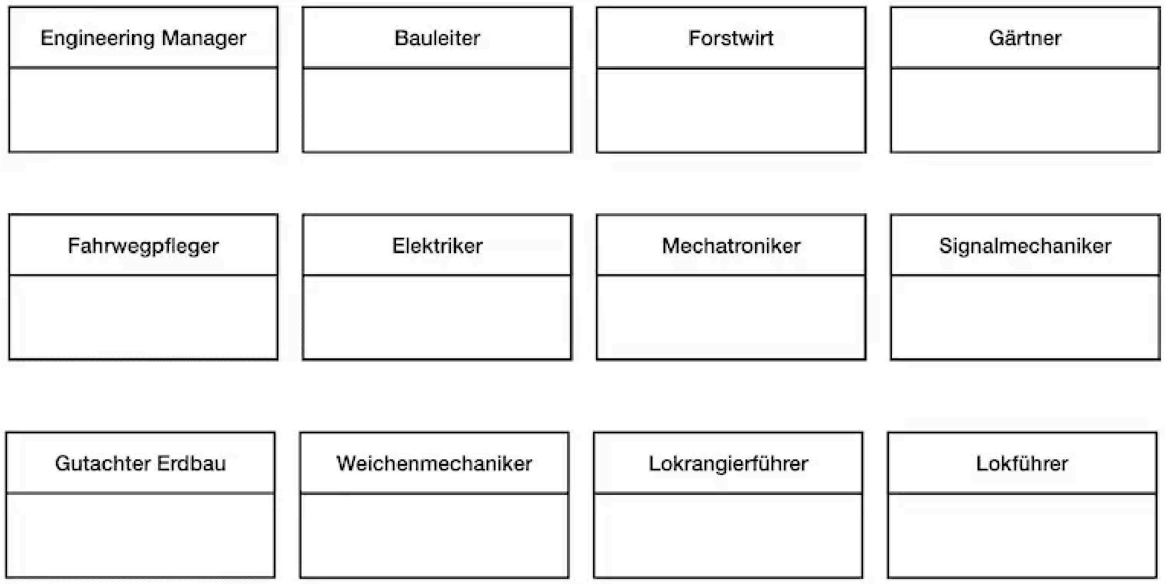
Inheritance?

- Maybe, we can extend the Mechanics class to have MorningMechanic and AfternoonMechanic.



The Scaling Problem

- However, what if we have tens of job titles?



Engineering Manager

Bauleiter

Forstwirt

Gärtner

Fahrwegpfleger

Elektriker

Mechatroniker

Signalmechaniker

Gutachter Erdbau

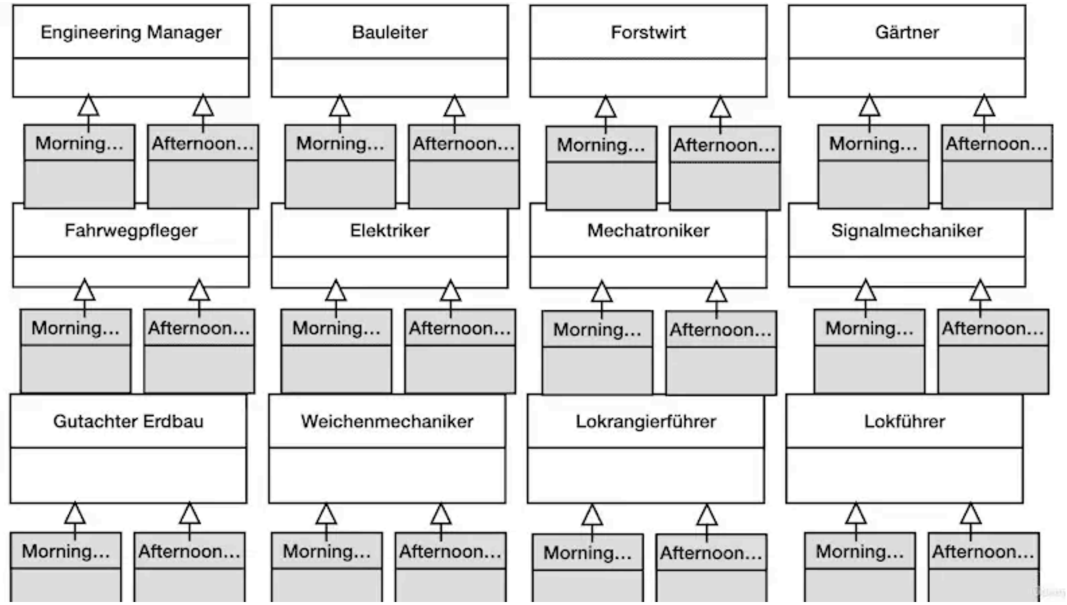
Weichenmechaniker

Lokrangierführer

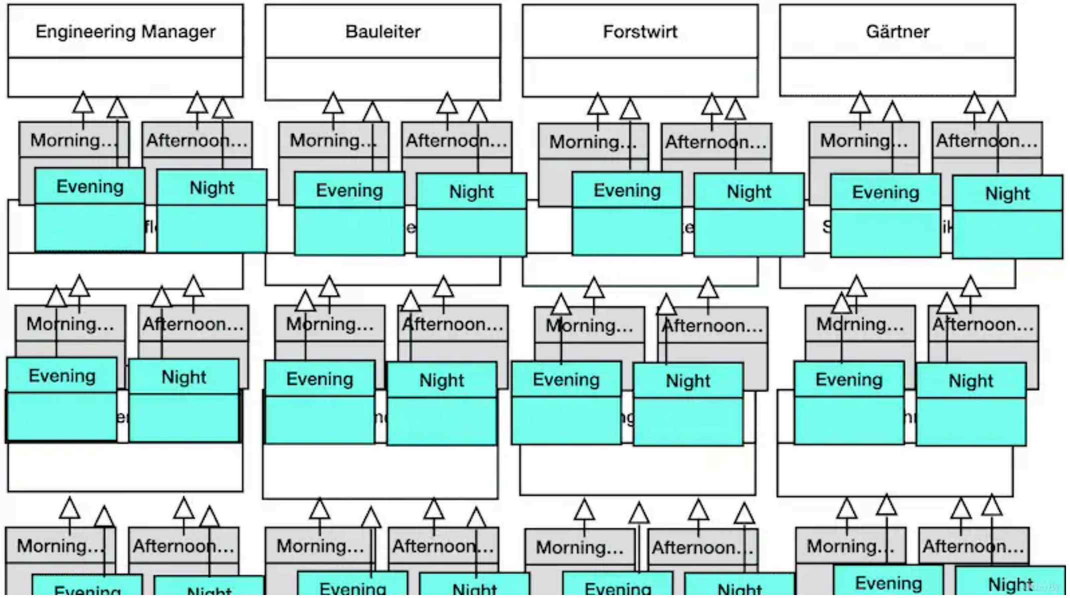
Lokführer

Explosion of classes

- We see the number of classes increases fast.

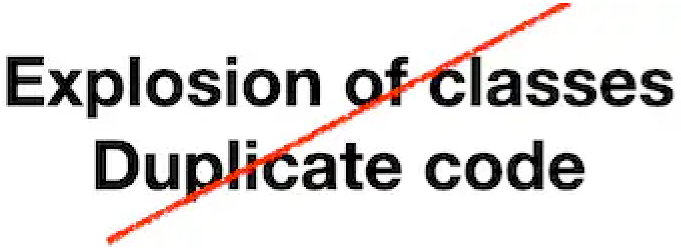


- What if we have more time-shift options?



- We call this `Explosion of classes`.
- Most of the classes duplicate similar features.
- These are code smells and an indication of bad software design.

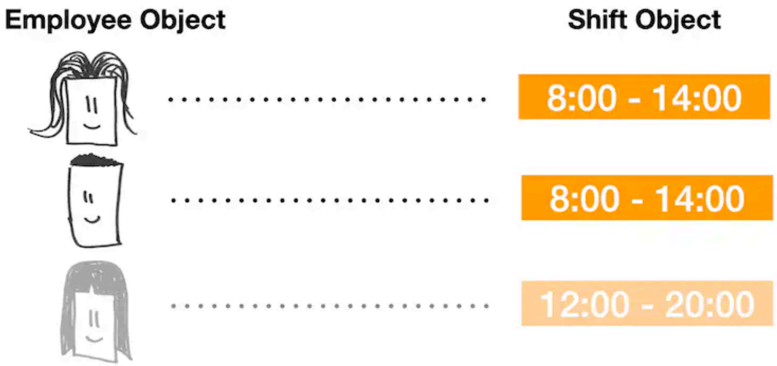
- **Warning**: Inheritance is not always the solution!



~~Explosion of classes~~
~~Duplicate code~~

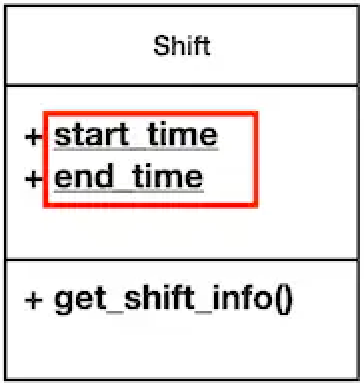
Understand Business Logic

- Each employee has a specific time shift.



Shift Class Design

- We can start making the Shift class.

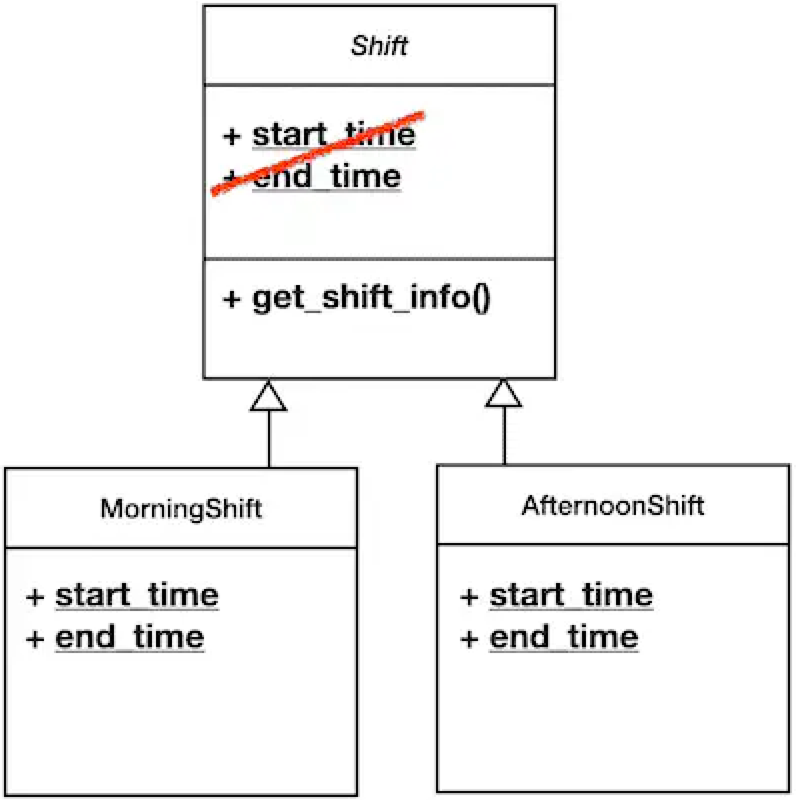


Shift

+ start_time
+ end_time

+ get_shift_info()

- We can also use inheritance to avoid duplication.
- However, it will cause the "explosion of classes."



Composition as the Solution

- So, in this case, we can use **composition**.
- Composition in OOP means a class is **composed of** other objects.
- Composition follows SRP by isolating concerns.

Composition Relationship

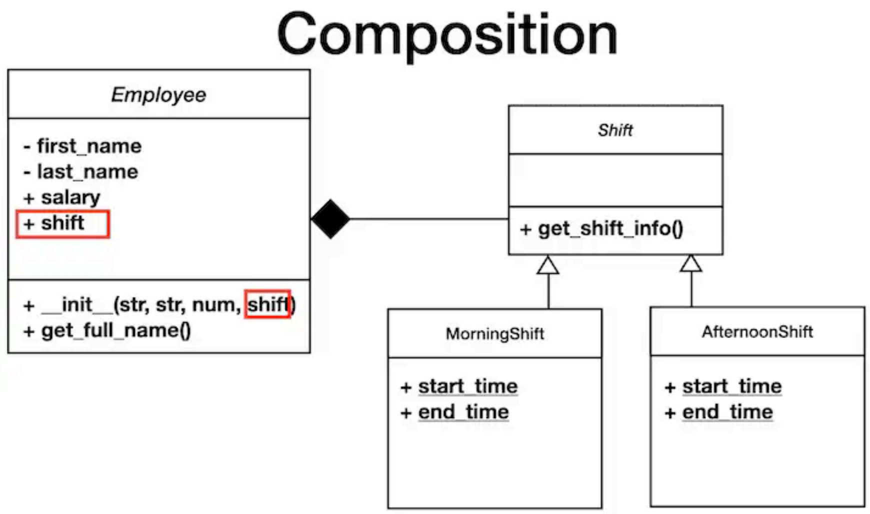
- We can make the Shift object a member of the Employee class using composition.
- This makes the relationship that Shift is a part of an Employee.
- Composition is a good solution when dealing with orthogonal concerns.

Aggregation vs Composition

- Aggregation is about ownership.
- We own a wallet or phone.
- Composition is about existence.
- A car is composed of an engine and a body.
- Composition is a much stronger relationship than Aggregation.

UML

- In UML, we use a **solid diamond** to indicate that the Shift is a part of the Employee.



UML Details

- The Employee class must have only **one** Shift object, so there is no number or * in the connection.
- Notice that start and end time are all class variables that are shared by all objects.

Implementation

```python
import datetime

class Shift:
    def get_shift_info(self):
        return f"{self.start_time:%H:%M} to {self.end_time:%H:%M}"

class MorningShift(Shift):
    start_time = datetime.time(8, 00)
    end_time = datetime.time(16, 00)

class AfternoonShift(Shift):
    start_time = datetime.time(12, 00)
    end_time = datetime.time(20, 00)
```

Python Pitfall

- Python uses `self` to express only the object member.
- However, start and end time are all **static** (class variables).
- So, there is no self in these fields.

Employee with Composition

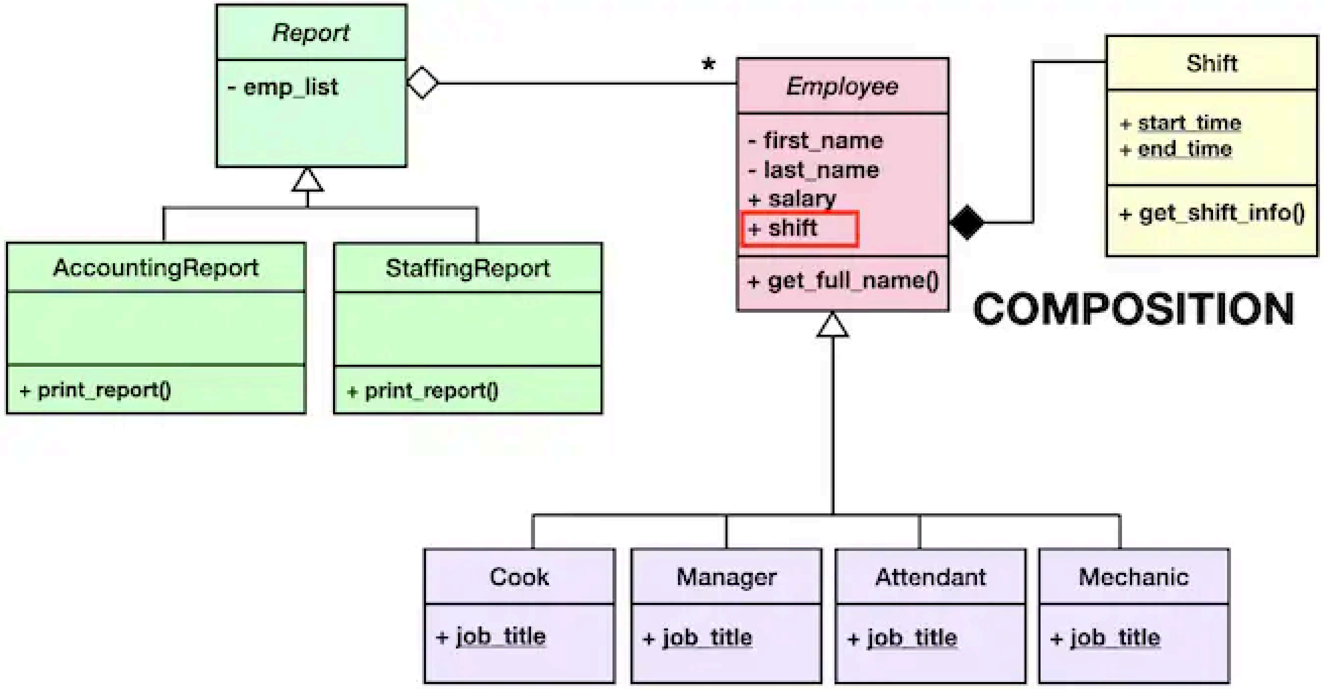
- The Employee class has an object shift as its fields to represent composition.

```python
class Employee:
    def __init__(self, first_name, last_name, salary, shift):
        self._first_name = first_name
        self._last_name = last_name
        self.salary = salary
        self.shift = shift  # shift object
```

Python Pitfall

- Python does not have the concept of aggregation or composition.
- It only has member fields.
- So, we use member fields to express aggregation or composition.

The Final Design



Composition, Aggregation, and Inheritance

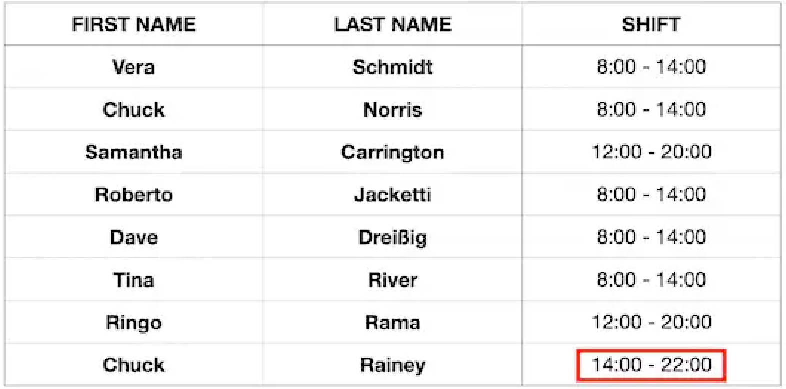
- The Report and Employee use the inheritance to remove duplication.
- The Employee and Shift use composition to avoid "explosion of classes."
- The Report uses aggregation for SRP (Single Responsibility Principle).

Usage Example

```python
from employee import Manager, Attendant, Mechanic
from shift import MorningShift, AfternoonShift

employees = [
    Manager("Schmidt", "Vera", 2000, MorningShift()),
    Attendant("Norris", "Chuck", 1800, MorningShift()),
    Mechanic("Rama", "Ringo", 1900, AfternoonShift()),
    Mechanic("Rainey", "Chuck", 1800, AfternoonShift()),
]
```

Easy Extension: Night Shift

- We have the software design to manage complexity from changes.
- What should we do to add the night shift?



FIRST NAME	LAST NAME	SHIFT
Vera	Schmidt	8:00 - 14:00
Chuck	Norris	8:00 - 14:00
Samantha	Carrington	12:00 - 20:00
Roberto	Jacketti	8:00 - 14:00
Dave	Dreißig	8:00 - 14:00
Tina	River	8:00 - 14:00
Ringo	Rama	12:00 - 20:00
Chuck	Rainey	14:00 - 22:00

Adding Night Shift

```python
# shift.py
class NightShift(Shift):
    start_time = datetime.time(14, 00)
    end_time = datetime.time(22, 00)

# main.py
from shift import MorningShift, AfternoonShift, NightShift

employees = [
    # ... existing employees
    Mechanic("Rainey", "Chuck", 1800, NightShift()),
]
```

- Adding one class by sub-classing from the Shift solves the problem.
- With software design, it is easy to extend without modifying existing code!

Lessons Learned

- We may have an *"explosion of classes"* when we use inheritance.
- In this case, we can use *Composition*.

- **Design Principle**: Favor composition over inheritance when dealing with orthogonal concerns.
- Benefits of Composition:
- Avoids class explosion
- More flexible than inheritance
- Easier to extend and maintain
- Better separation of concerns