Lecture 20: Builder Pattern
IN710: Programming 4
Semester One, 2020

Kaiako: Grayson Orr

Te Kura Matatini ki Otago, Ōtepoti, Aotearoa

Lecture 09: Adapter Pattern Recap

- Design pattern 05: adapter pattern
  - Definition
  - Problem/solution
  - Real world analogy
  - UML & implementation
  - Pros & cons

Lecture 10: Builder Pattern Topics

- Design pattern 06: builder pattern
  - Definition
  - Problem/solution
  - UML & implementation
  - Applicability
  - Pros & cons

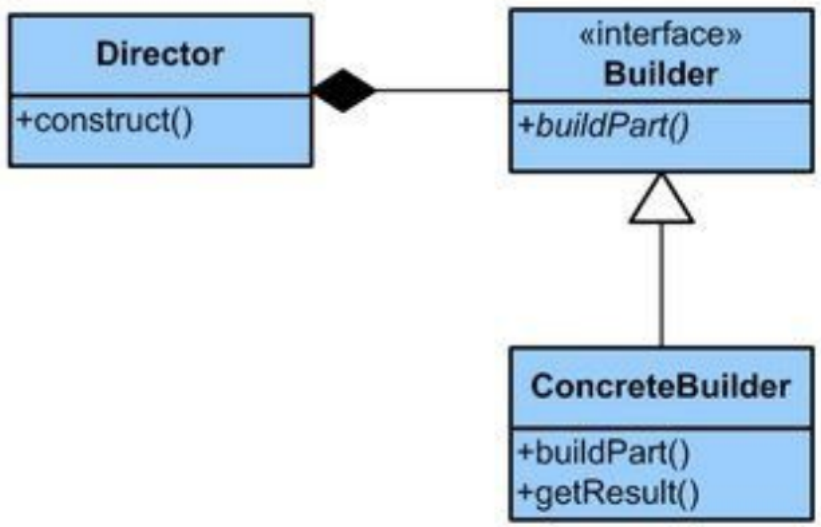
Builder Pattern: GoF

- GoF definition & UML

Builder

Type: Creational

What it is:
Separate the construction of a complex object from its representing so that the same construction process can create different representations.



Builder Pattern: Definition

- Creational pattern
- Separates the construction of a complex object from its representation
- The same construction process can create different representations

Builder Pattern: Problem

- Building a house

```
class House:
    def __init__(self, num_rooms, num_doors, num_windows,
                 has_garage, has_garden, has_swimming_pool):
        self.num_rooms = num_rooms
        self.num_doors = num_doors
        self.num_windows = num_windows
        self.has_garage = has_garage
        self.has_garden = has_garden
        self.has_swimming_pool = has_swimming_pool

def main():
    house = House(4, 8, 20, False, True, True)

if __name__ == '__main__':
    main()
```

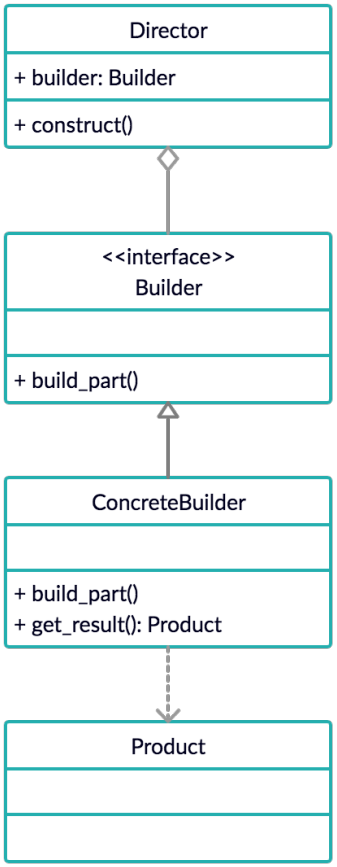
Builder Pattern: Solution

- HouseBuilder class

```
class HouseBuilder(Builder):
    def __init__(self):
        self.reset()

    def reset(self):
        self.__product = House()

    @property
    def product(self):
        product = self.__product
        self.reset()
        return product

    def set_num_rooms(self, num_rooms):
        pass

    def set_num_doors(self, num_doors):
        pass

    def set_num_windows(self, num_windows):
        pass

    def set_has_garage(self, has_garage):
        pass

    def set_has_garden(self, has_garden):
        pass

    def set_has_swimming_pool(self, has_swimming_pool):
        pass
```

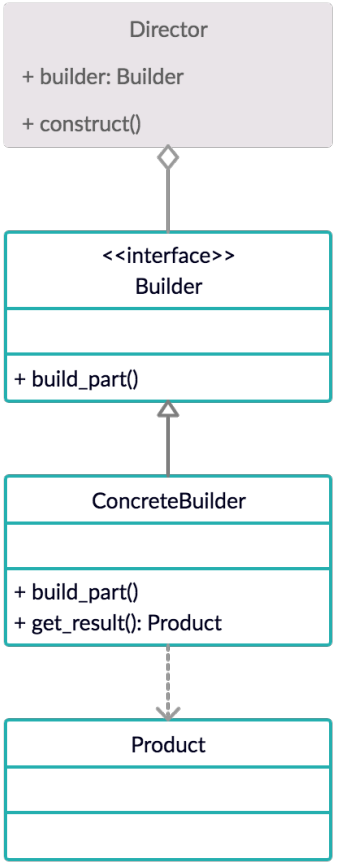
Builder Pattern: UML

- Consider the following UML diagram:



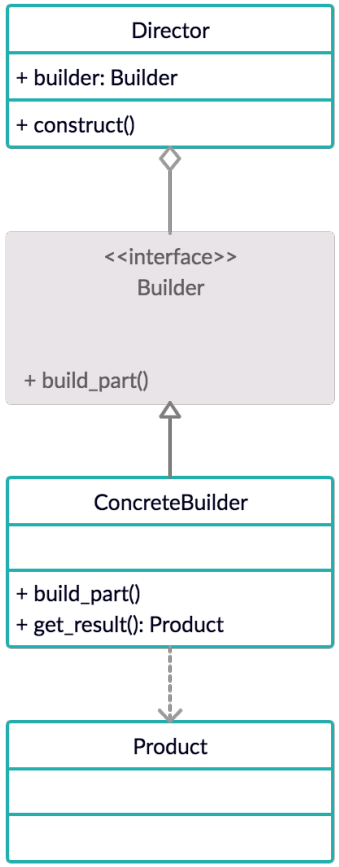
Builder Pattern: UML

- Director class
- Doesn't create the product class
- Refers to the builder interface class for creating the parts of a complex object



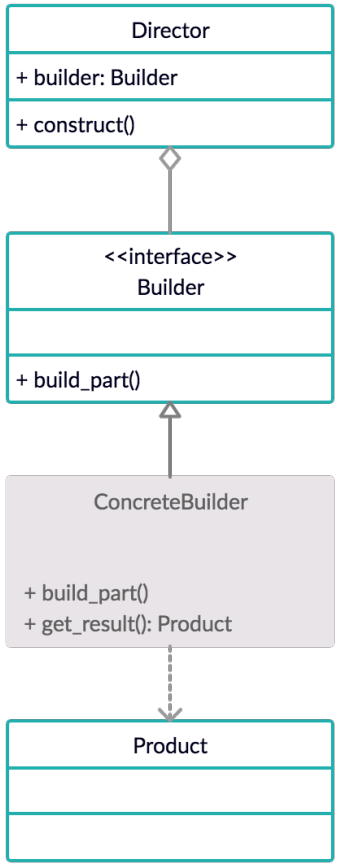
Builder Pattern: UML

- Builder interface class



Builder Pattern: UML

- ConcreteBuilder class
- Implements the builder interface class by creating the product object



Builder Pattern: Implementation

```
from abc import ABC, abstractmethod, abstractproperty

class Builder(ABC):
    @abstractproperty
    def product(self):
        pass

    @abstractmethod
    def set_first_name(self, first_name):
        pass

    @abstractmethod
    def set_last_name(self, last_name):
        pass

    @abstractmethod
    def set_email_address(self, email_address):
        pass

    @abstractmethod
    def set_phone_number(self, phone_number):
        pass
```

Builder Pattern: Implementation

```
class PersonBuilder(Builder):
    def __init__(self):
        self.reset()

    def reset(self):
        self.__product = Person()

    @property
    def product(self):
        product = self.__product
        self.reset()
        return product

    def set_first_name(self, first_name):
        self.__product.add(f'First name: {first_name}')

    def set_last_name(self, last_name):
        self.__product.add(f'Last name: {last_name}')

    def set_email_address(self, email_address):
        self.__product.add(f'Email address: {email_address}')

    def set_phone_number(self, phone_number):
        self.__product.add(f'Phone number: {phone_number}')
```

Builder Pattern: Implementation

```
class Person:
    def __init__(self):
        self.details = []

    def add(self, detail):
        self.details.append(detail)

    def display_details(self):
        for d in self.details:
            print(d)

class Director:
    def __init__(self):
        self.__builder = None

    @property
    def builder(self):
        return self.__builder

    @builder.setter
    def builder(self, builder):
        self.__builder = builder

    def build_person_details(self):
        self.builder.set_first_name('John')
        self.builder.set_last_name('Doe')
        self.builder.set_email_address('johndoe@gmail.com')
        self.builder.set_phone_number('0271234567')
```

Builder Pattern: Implementation

- Custom builder

```
def main():
    director = Director()
    person_builder = PersonBuilder()
    director.builder = person_builder

    print('Person details:')
    director.build_person_details()
    person_builder.product.display_details()

    print('\nCustom person details:')
    person_builder.set_first_name('Jane')
    person_builder.set_last_name('Doe')
    person_builder.product.display_details()

if __name__ == '__main__':
    main()
```

Builder Pattern: Applicability

- Telescopic constructors
- Create different representations of the same product
- Construct composite trees or other complex objects
- Android - AlertDialog.Builder

Builder Pattern: Pros

- Constructed objects can be step-by-step, defer construction steps or run steps recursively
- When building representations of a product, the same construction code can be reused
- The construction code is isolated from the business logic

Builder Pattern: Cons

- Overall complexity of the code increases - creating multiple new classes is required
- The builder classes are required to be mutable
- Dependency injection may be less supported

Practical

- Series of tasks covering today's lecture
- Worth 1% of your final mark for the Programming 4 course
- Deadline: Friday, 12 June at 5pm