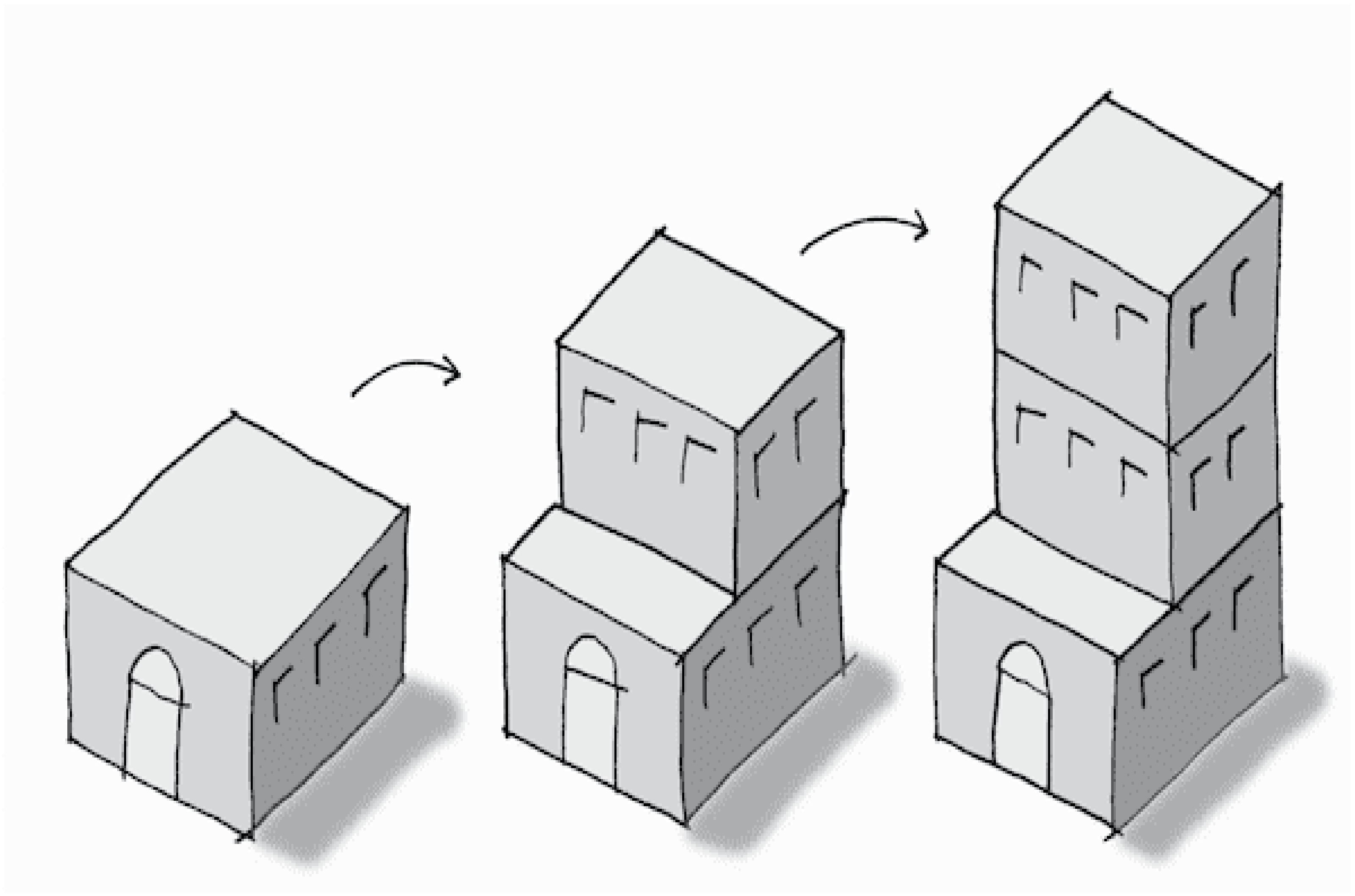
Builder Pattern

Build Complicated Instances with the Director



Builder Pattern

Same steps, different results.

You need to separate **how to build** from **what you’re building** so you can reuse the **process** while swapping **parts/variants**.

Core Idea (House Analogy)

- **Director** = Site manager following the *construction plan* (order of steps)
- **Builder** = Crew that knows *how* to do each step for a specific house type
- **Product** = The finished house (ranch, two-story, etc.)

The Problem

- Creating complex objects requires many steps.
- The construction process should be the same, but the representation should vary.
- We want to separate the construction logic (how) from the representation (what).

The challenge: how to **separate construction algorithm (how)** from **representation (what)** so the same process can create **different complex objects**?

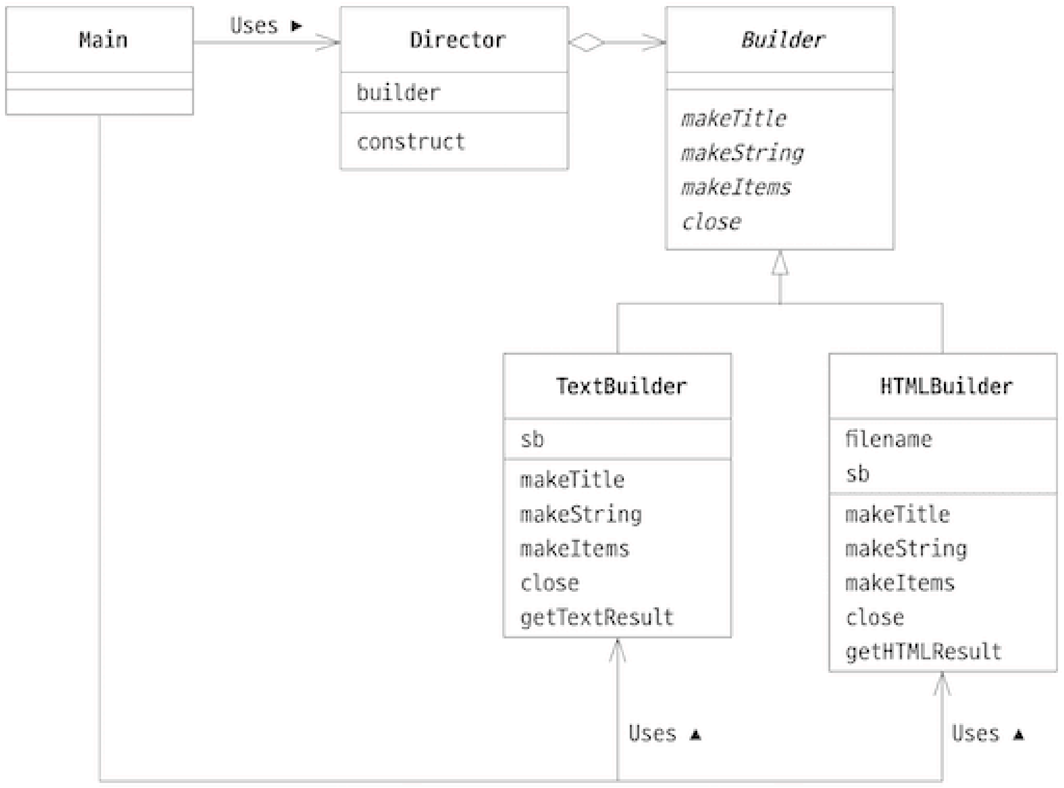
The *Builder* as the Solution

- Separate the construction (how) of a complex object from its representation (what).
- The same construction process can create different representations (outcomes).
- The *Director* `controls` the process while *Builders* `implement` the details.

The Design

The easiest way for the Director to control Builder is to use aggregation (ownership).

- We can have different builders.



Step 1: Understand the Players

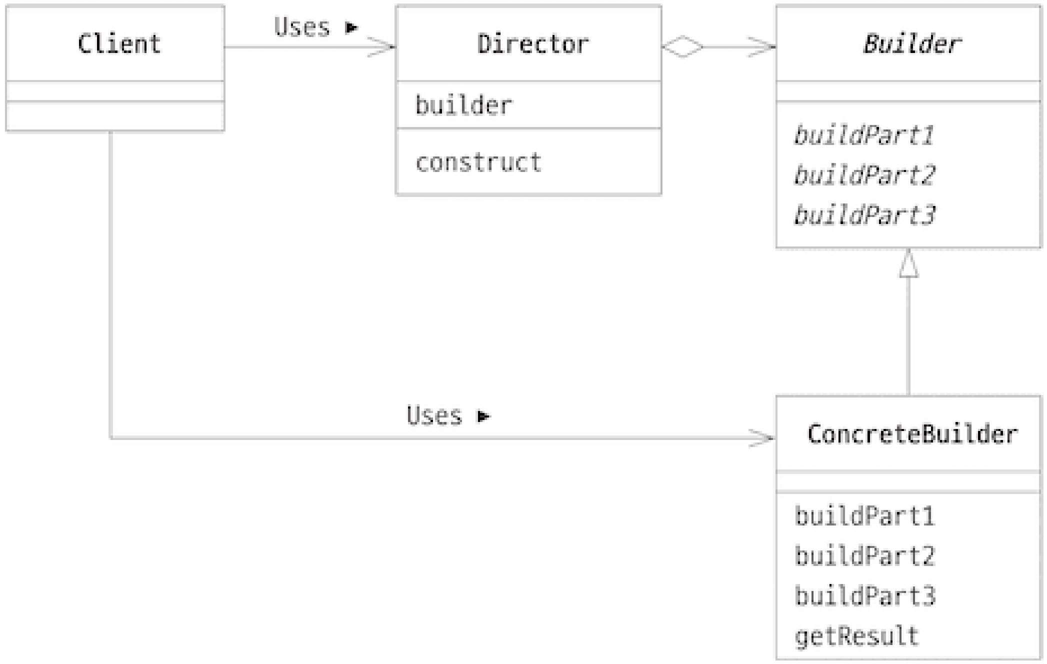
In this design, we have players:

- **Director** (controls construction sequence)
- *Builder* (abstract interface for construction)
    - **ConcreteBuilder** (implements construction steps)

We need a client to initiate construction:

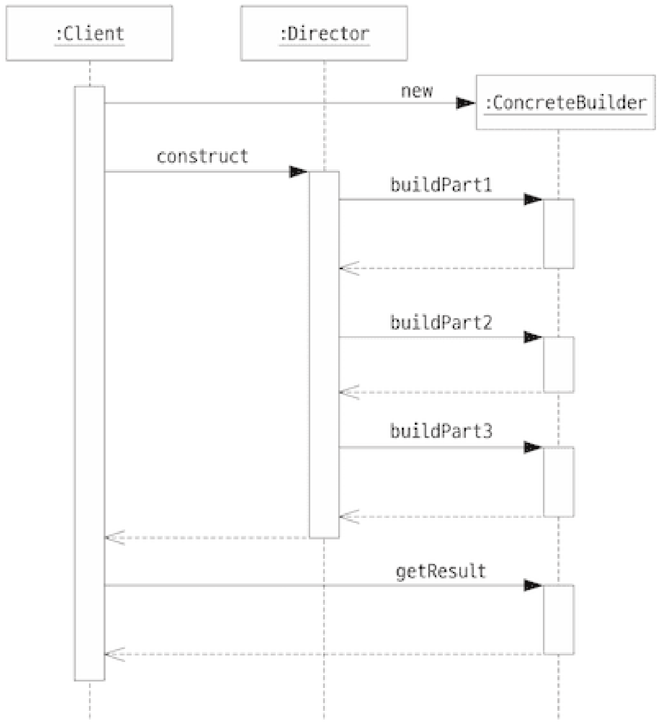
- **Client**

Step 2: Separation of *abstraction* and concretion



- **Director** works with *Builder* interface, not concrete builders.
- **ConcreteBuilders** implement specific construction logic.

Sequence Diagram



- The **Client** creates (chooses) the Builder to use.
- The **Director** orchestrates the construction by calling builder methods in sequence.

Code

- Main Method
- Director (Controls Construction)
- Builder Interface
- Concrete Builders (Text & HTML)

Main Method

```python
from director import Director
from text_builder import TextBuilder
from html_builder import HTMLBuilder

def main():
    if format_type == "text":
        text_builder = TextBuilder()
        director = Director(text_builder)
        director.construct()
        result = text_builder.get_text_result()
        print(result)
    elif format_type == "html":
        html_builder = HTMLBuilder()
        director = Director(html_builder)
        director.construct()
        filename = html_builder.get_html_result()
```

Step 1: Choose builder type

Decide **what** to build:

```
text_builder = TextBuilder()
# or
html_builder = HTMLBuilder()
```

- Select the appropriate builder based on the desired output format.
- Both implement the same *Builder* interface.

Step 2: Create a director with a builder

Then tell the Director (who knows **how** to build it) to use the Builder.

```
director = Director(text_builder)
director.construct() # Template/Algorithm
```

- **Director** receives a builder and controls the construction process.
- The same director can work with any builder implementation.

Director Implementation

The construct() method has the algorithm, but instead of delegating the implementation later, it uses aggregation for implementation.

```python
class Director:
    def __init__(self, builder):
        self.builder = builder

    def construct(self):
        self.builder.make_title("Greeting")
        self.builder.make_string("General greetings")
        self.builder.make_string("Time-based greetings")
        ...
        self.builder.close()
```

Builder Interface

```python
from abc import ABC, abstractmethod

class Builder(ABC):
    @abstractmethod
    def make_title(self, title): pass

    @abstractmethod
    def make_string(self, string): pass

    @abstractmethod
    def make_items(self, items): pass

    @abstractmethod
    def close(self): pass
```

Text Builder Implementation

```python
class TextBuilder(Builder):
    def __init__(self):
        self.buffer = []

    def make_title(self, title):
        self.buffer.append("=" * 30)
        self.buffer.append(f"[{title}]")
        self.buffer.append("")

    def make_string(self, string):
        self.buffer.append(f"■ {string}")
        self.buffer.append("")

    def make_items(self, items):
        for item in items:
            self.buffer.append(f"  - {item}")
        self.buffer.append("")

    def get_text_result(self):
        return "\n".join(self.buffer)
```

HTML Builder Implementation

```python
class HTMLBuilder(Builder):
    def __init__(self):
        self.buffer = []
        self.filename = "untitled.html"

    def make_title(self, title):
        self.filename = f"{title}.html"
        self.buffer.extend([
            "<!DOCTYPE html>", "<html>",
            f"<head><title>{title}</title></head>",
            "<body>", f"<h1>{title}</h1>"
        ])

    def make_items(self, items):
        self.buffer.append("<ul>")
        for item in items:
            self.buffer.append(f"<li>{item}</li>")
        self.buffer.append("</ul>")

    def close(self):
        self.buffer.extend(["</body>", "</html>"])
        with open(self.filename, 'w') as f:
            f.write("\n".join(self.buffer))
```

Output Examples

Text Output:

```
==============================
[Greeting]

■ General greetings

  - How are you?
  - Hello.
  - Hi.
==============================
```

HTML Output:

```html
<!DOCTYPE html>
<html>
<head><title>Greeting</title></head>
<body>
<h1>Greeting</h1>
<p>General greetings</p>
<ul>
<li>How are you?</li>
<li>Hello.</li>
</ul>
</body>
</html>
```

Discussion

Why does the Main (Client) not interact with the Builder directly?

- Because the Main (Client) can use different Directors to use different algorithms.
- One builder builds a two-floor house, and the other builder builds a ranch house.

Why does the Builder work with the Builder interface, not concrete Builders?

- We add abstraction to remove coupling: It is just like we tend to ask for "drinks", not a specific "Coke" or "Pepsi".

The Builder pattern is only useful for objects with many constructor parameters.

- No, it's also useful when construction requires a specific sequence of steps or when you want to create different representations of the same object

Builder Method Chaining

When we have this code:

```python
class HouseBuilder:
    def __init__(self):
        self.house = House()

    def build_foundation(self):
        # Build foundation
        return self  # For method chaining

    def build_walls(self):
        # Build walls
        return self

    def get_result(self):
        return self.house
```

We can use the Method chaining to allow multiple method calls to be linked together by returning `self` from each method.

```
house = HouseBuilder() \
    .build_foundation() \
    .build_walls() \
    .build_roof() \
    .get_result()
```

Builder Benefits

- **Flexibility**: Same process, different products
- **Control**: Director controls construction sequence
- **Separation**: Construction logic separated from representation
- **Extensibility**: Easy to add new builder types

When to Use Builder

- Complex object construction with many steps
- Same construction process needs **different** representations
- Want to isolate construction code from representation
- Need to control construction sequence

Related Patterns

- **Template Method** (Skeleton algorithm): Director uses template method approach

Comparison to Template Method

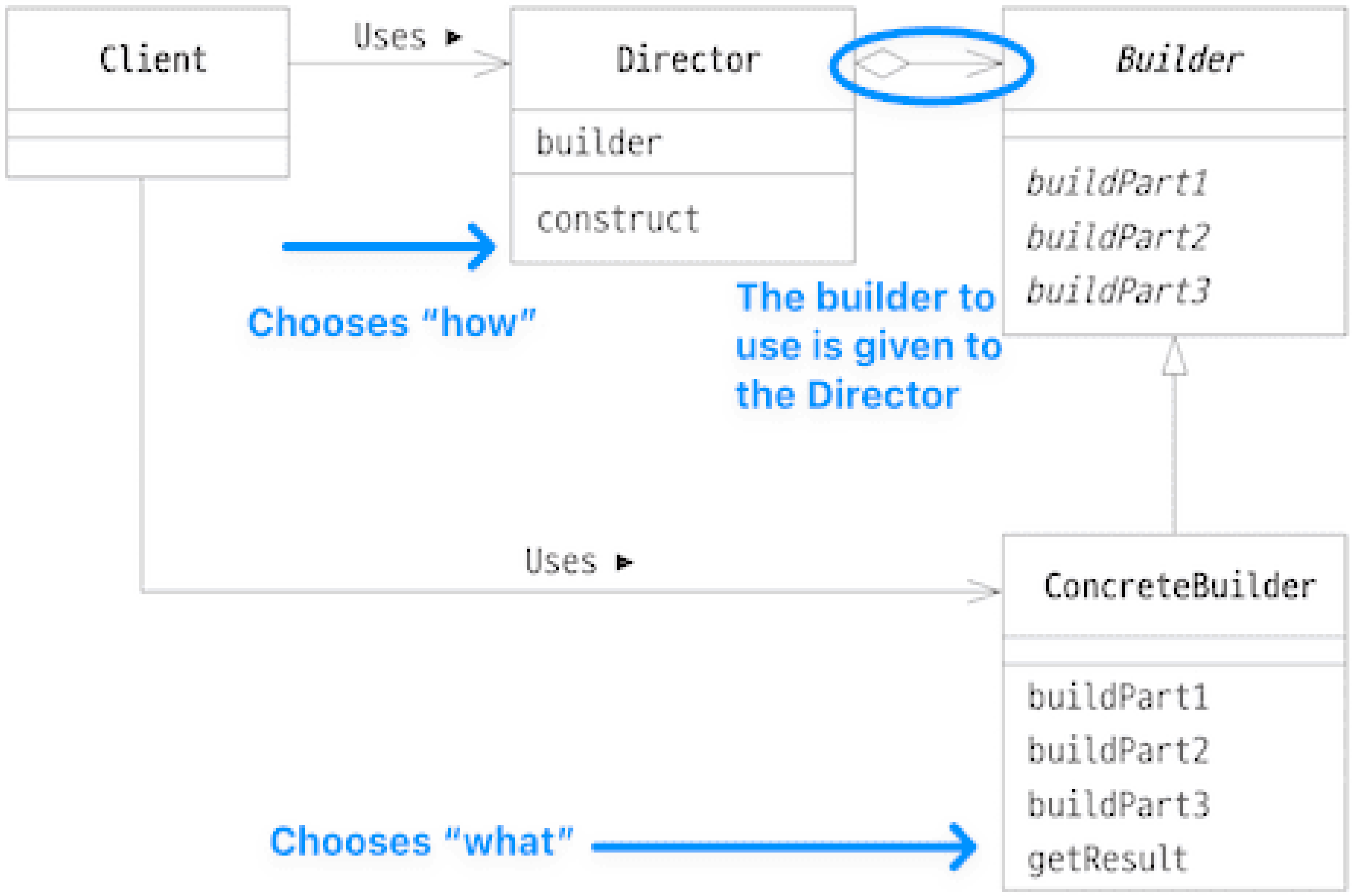
Template Method:

- Super class manages sub classes (the algorithm is fixed)
- We don't need the sequential UML diagram as it is static.

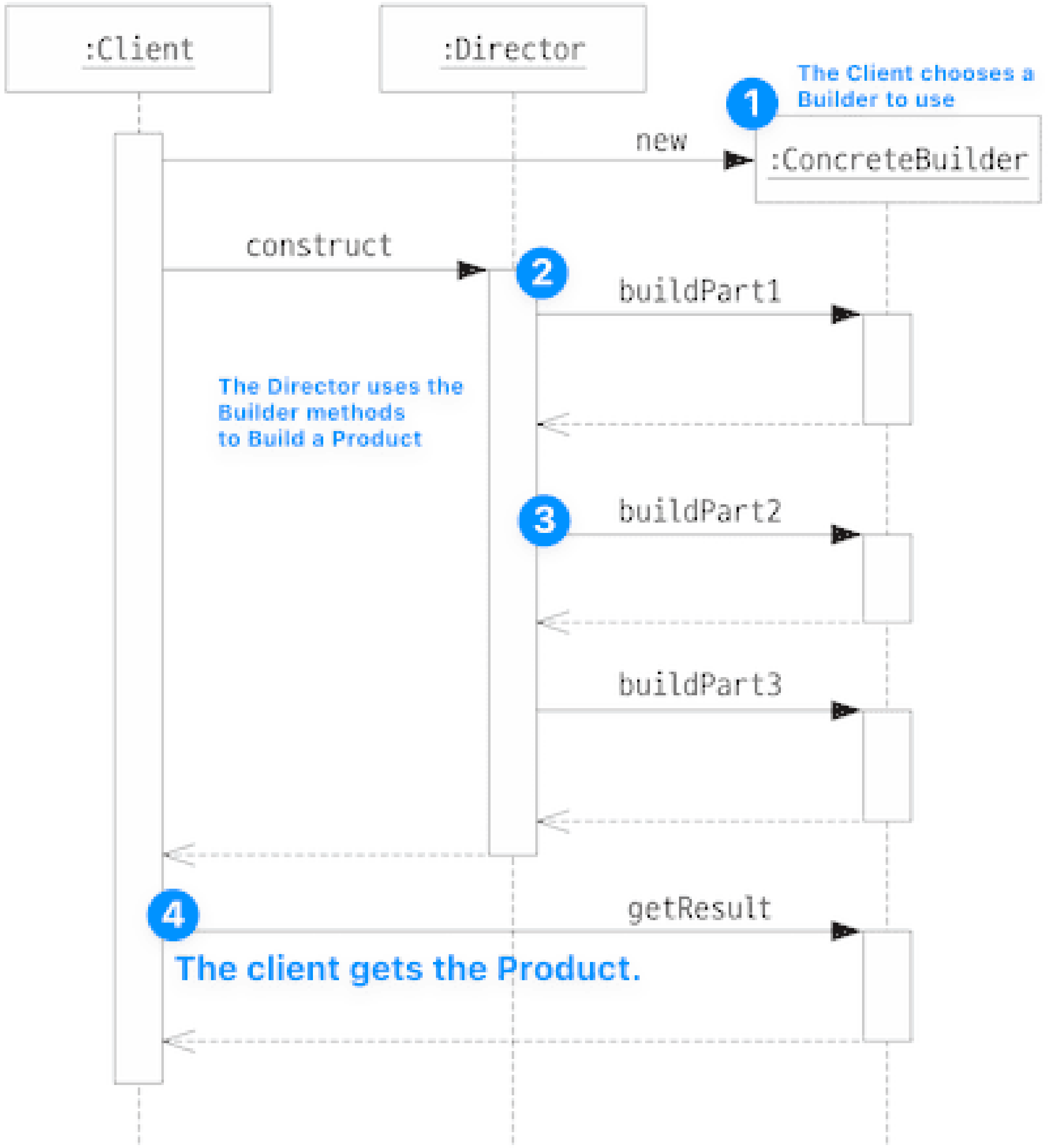
Builder Pattern:

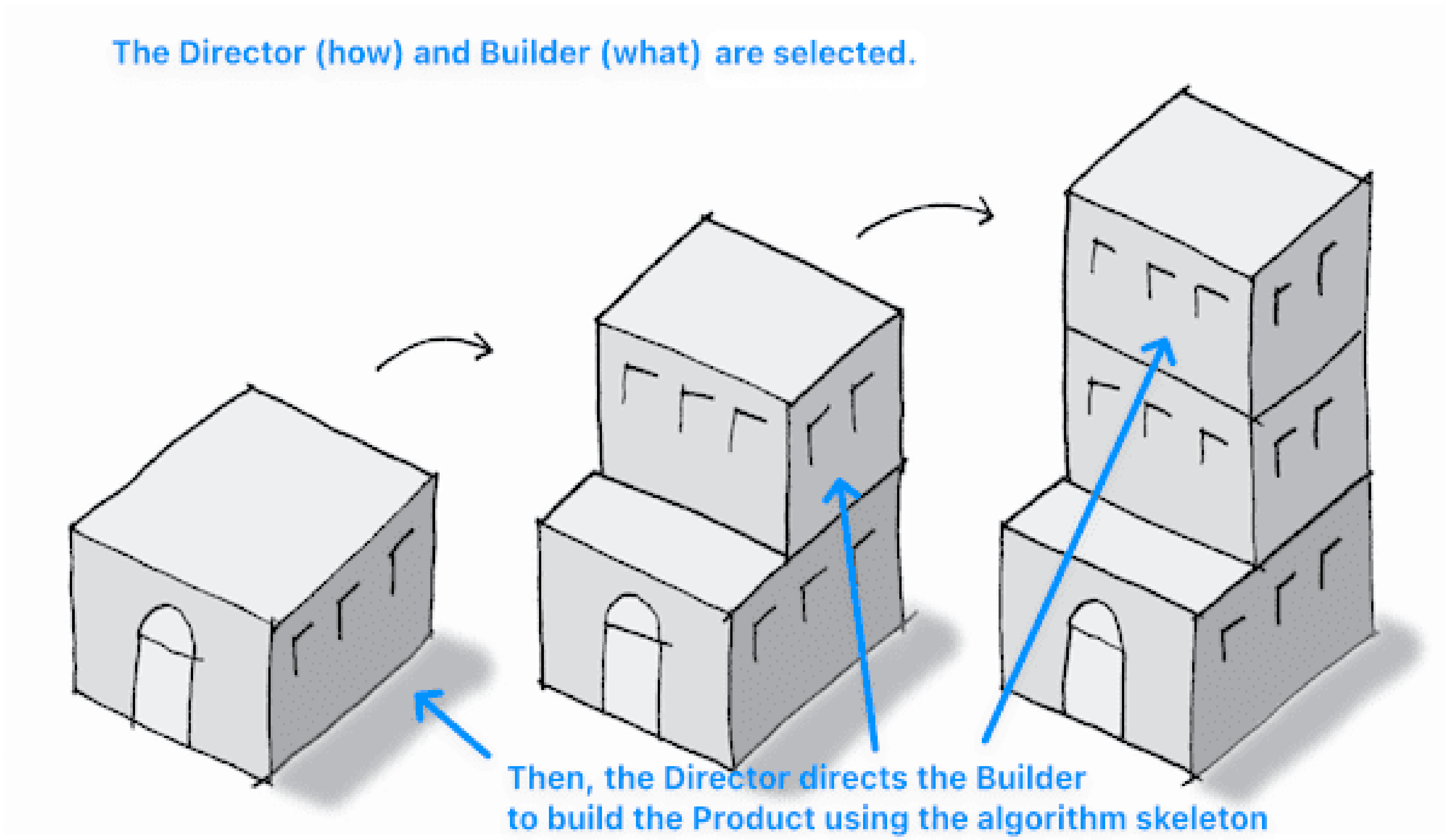
- The **Director** class manages **Builder** classes (we can choose algorithms/concrete builders)
- We need the sequence UML diagram to show the interaction between the Director and the Builder

UML



Client
Uses ▶
Director
builder
construct
Builder
buildPart1
buildPart2
buildPart3
Chooses "how"
The builder to use is given to the Director
Uses ▶
ConcreteBuilder
buildPart1
buildPart2
buildPart3
getResult
Chooses "what"


:Client
:Director
1
The Client chooses a Builder to use
new
:ConcreteBuilder
construct
2
buildPart1
The Director uses the Builder methods to Build a Product
3
buildPart2
buildPart3
getResult
4
The client gets the Product.


The Director (how) and Builder (what) are selected.
Then, the Director directs the Builder
to build the Product using the algorithm skeleton