SE 350 - Object-Oriented Software Development

Software Design Patterns: Everyday Patterns

Instructor: Stefan Mitsch

Learning Objectives

- Learn about design patterns that became everyday programming practice

Observer Example

```
// a button notifies subscribers when it is pressed
JButton b = new JButton("Hello, world!");

// a subscriber, implemented with an anonymous class
b.addActionListener(new ActionListener() {
  public void actionPerformed(ActionEvent e) {
    System.out.println("Hello, anonymous world!");
  }
});

// a subscriber, implemented with a lambda expression
b.addActionListener(e -> System.out.println("Hello, lambda world!"));
```

Observer Overview

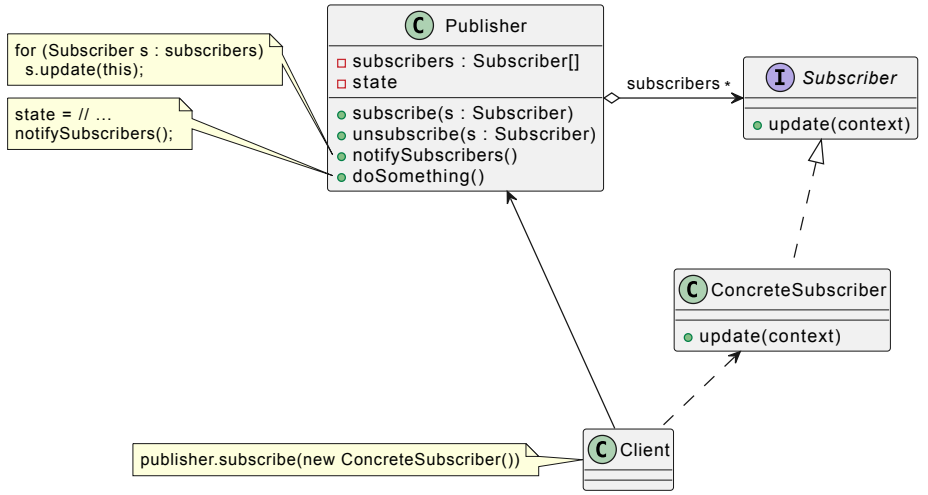
Problem

Avoid frequent status requests or unsolicitated calls

Intent

- Define a subscription mechanism
- Notify multiple objects at once about events

Structure

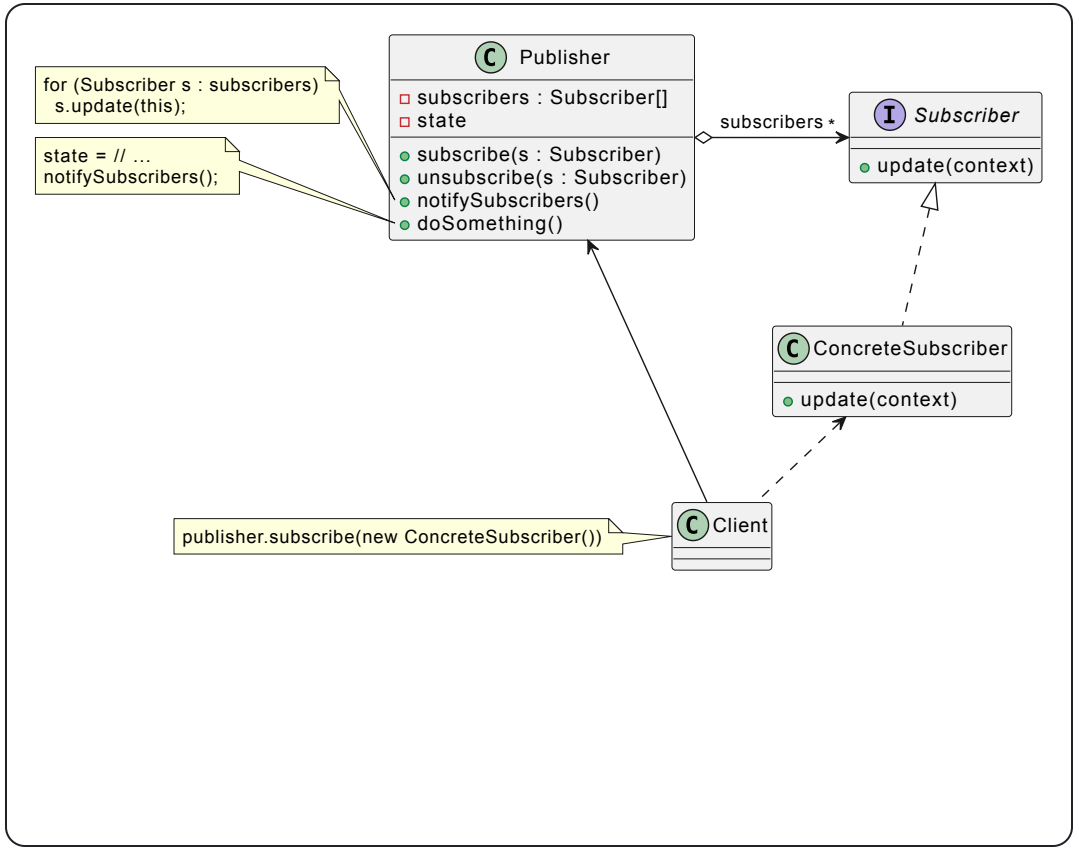


Observer Participants

1. The `Publisher` issues events to other objects when the state of the publisher changes or certain behavior executes. Publishers provide a subscription infrastructure for new subscribers to join and current subscribers to leave.

2. When a new event happens, the publisher notifies all subscribed subscribers.

3. The `Subscriber` interface declares methods how subscribers can be notified (often a single `update` method); the method may have parameters for event details.



4. `ConcreteSubscriber` classes perform some action in response to being notified.

5. Often, subscribers need context information to handle the notification correctly. The publisher can either pass context data as argument to the `update` method, or pass itself so that subscribers can fetch the necessary context.

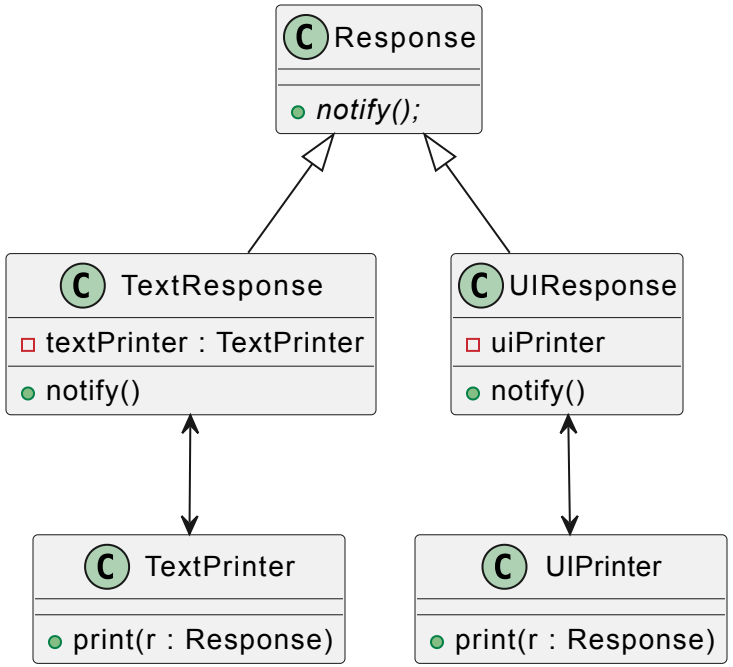
6. The `Client` creates publishers and subscribers separately and registers subscribers for events.

Replace Hard-Coded Notifications

Code Smell: Subclasses are hard-coded to notify a specific object

Before: A subclass per object to notify, tightly coupled publisher and subscriber

▶ After Refactoring



Observer Discussion

Applicability

- Use when changes to the state of one object (`Publisher`) require changing other objects (`Subscriber`)
- Use when interests change at runtime (`subscribe` and `unsubscribe`)

Benefits

- Publisher and subscriber are decoupled and can evolve separately
- Subscriptions can be changed at runtime

Drawbacks

- Subscriber notification order is random

Iterator Example

```
List<Integer> values = List.of(1, 2, 3, 4, 5);

// iterate with explicit iterator
Iterator<Integer> it = values.iterator();
while (it.hasNext()) {
  System.out.println(it.next());
}

// iterate with for-expression
for (int v : values) System.out.println(v);
```

Iterator Overview

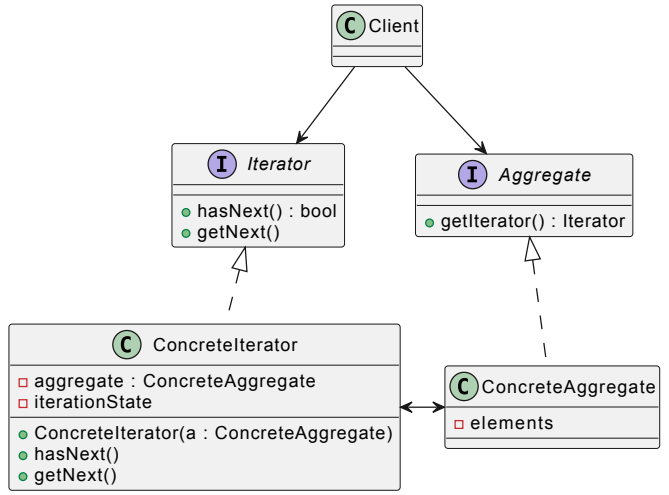
Problem

Traverse elements of an aggregate object sequentially without exposing its underlying representation

Intent

- Traverse elements sequentially
- Adapt way of traversing same data structure (e.g., traverse a tree depth first vs. breadth first)

Structure

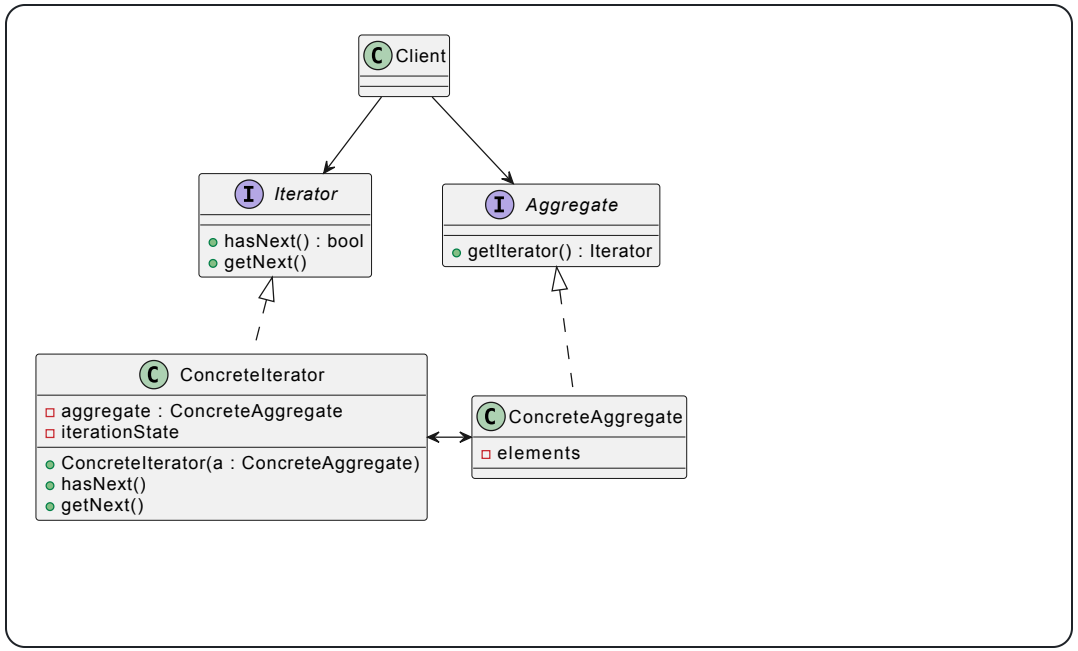


Iterator Participants

1. The `Iterator` interface declares the traversal operations.

2. `ConcreteIterator` objects implement specific traversal operations (e.g., depth first); they track the iteration state. Multiple iterators can traverse the same aggregate object.

3. The `Aggregate` interface declares methods for getting iterators.



4. `ConcreteAggregate` objects return new concrete iterators each time a client requests one.

5. The `Client` works with both collections and iterators through their interfaces, so that it avoids being coupled to concrete implementations. A client does not create an iterator itself, it gets it from a collection.

Iterator Discussion

Applicability

- The `Aggregate` has a complex structure
- There are multiple ways of traversing an `Aggregate`
- The `Client` wants to address multiple different aggregates in a uniform way

Benefits

- Encapsulated bulky traversal algorithms
- Decoupled clients and aggregates
- Collections can be traversed in parallel (multiple iterators)
- Iteration can be delayed, suspended, resumed

Drawbacks

- Not for simple collections
- Iterator can introduce runtime complexity over traversing elements at a low level directly

Builder Example

```
// client creates a builder
StringBuilder s = new StringBuilder();

// director knows the steps of building the object
s.append("Hello");
s.append(",");
s.append(" ");
s.append("world");

// client fetches the final product
String str = s.toString();
```

```
// client creates a builder
Stream.Builder<Int> builder = Stream.builder();

// director knows the steps of building the object
builder.
  add(1).
  add(2).
  add(3);

// client fetches the final product
Stream<Int> values = builder.build();
```

Builder Overview

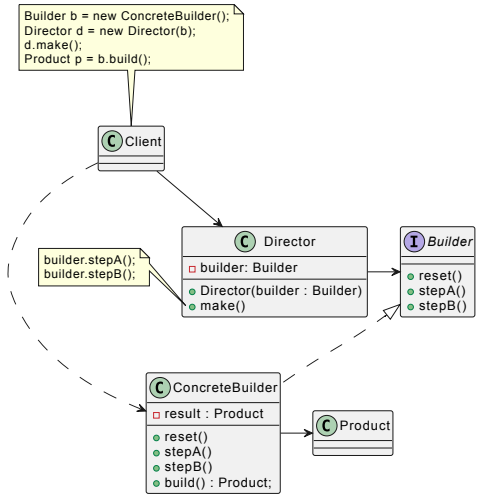
Problem

Avoid tight coupling between object creation and object representation

Intent

- Create complex objects step by step
- Produce different types and representations using the same construction code

Structure

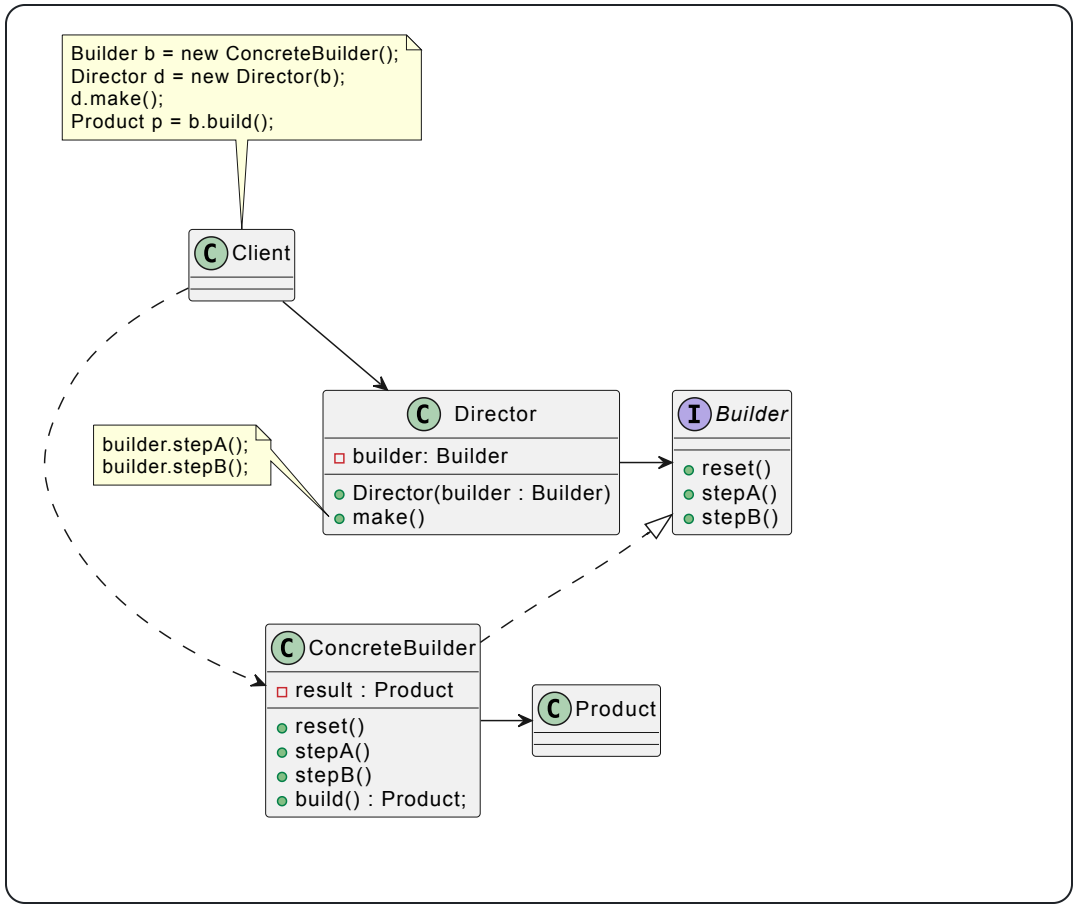


Builder Participants

1. The `Builder` interface declares construction steps.

2. `ConcreteBuilder` objects provide different implementations of construction steps and create concrete `Product` objects.

3. `Product` objects are the results of production. Different builders can create different concrete products.



4. The `Director` class decides which construction steps to use in which order.

5. The `Client` associates a builder object with a director; the director uses this builder for all construction steps.

Builder Discussion

Applicability

- When constructors have many optional parameters
- When different `Product` involve similar steps of construction
- To create complex objects (like Composite)

Benefits

- Construct objects step by step, recursive construction
- Reuse construction code
- Isolate construction from other functionality

Drawbacks

- Several additional classes needed

Design Pattern Resources

- Refactoring Guru Design Patterns
- Java Design Patterns Catalog
- Code examples
  - Java Design Pattern Implementations
  - GoF Java Design Pattern Implementations
  - GoF Java Design Pattern Implementations
  - GoF Python Design Pattern Implementations