# Behavioral Pattern: Iterator

**Kevin Dockx**
Architect

@KevinDockx https://www.kevindockx.com

## Coming Up

**Describing the iterator pattern**

- Iterating over a list in alphabetical order

**Structure of the iterator pattern**

# Coming Up

**Use cases for this pattern**

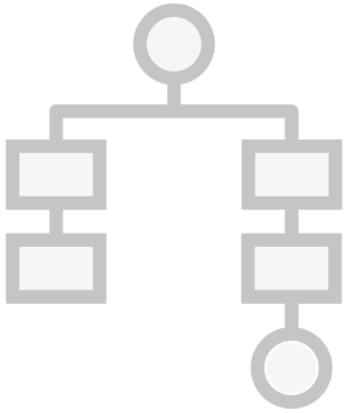
**Pattern consequences**

**Related patterns**

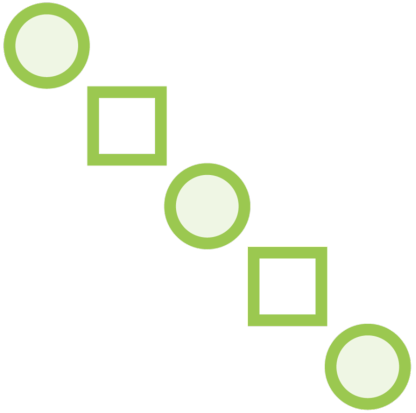
# Describing the Iterator Pattern



Creational



Structural



**Behavioral**

# Iterator

**The intent of this pattern is to provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation**

# Describing the Iterator Pattern

**Aggregate objects (List, Dictionary, Stack, Queue, ...) keep their items in an internal structure**

- You don't want to expose that and break encapsulation

# Describing the Iterator Pattern

**You might not always want to traverse aggregate objects in the same way**

- Alphabetically, reverse order, custom order, ...

**Avoid bloating the aggregate object interface**

# Describing the Iterator Pattern

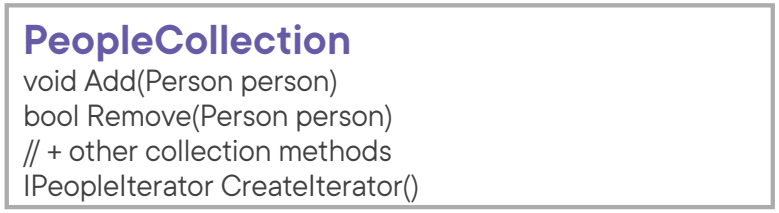
**Custom collection of people**

- Iterator will iterator over them alphabetically, by the person's name

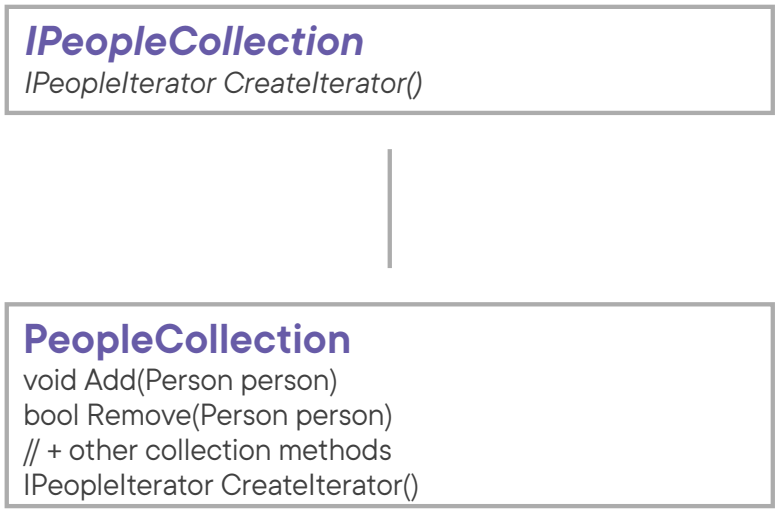
# Describing the Iterator Pattern

**PeopleCollection**
void Add(Person person)
bool Remove(Person person)
// + other collection methods

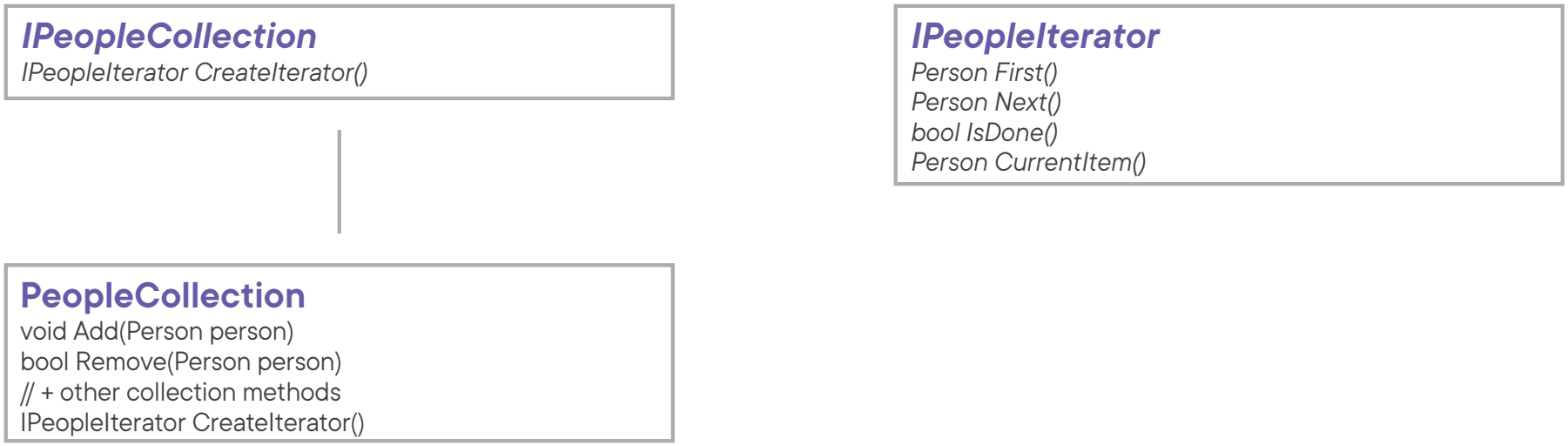
# Describing the Iterator Pattern



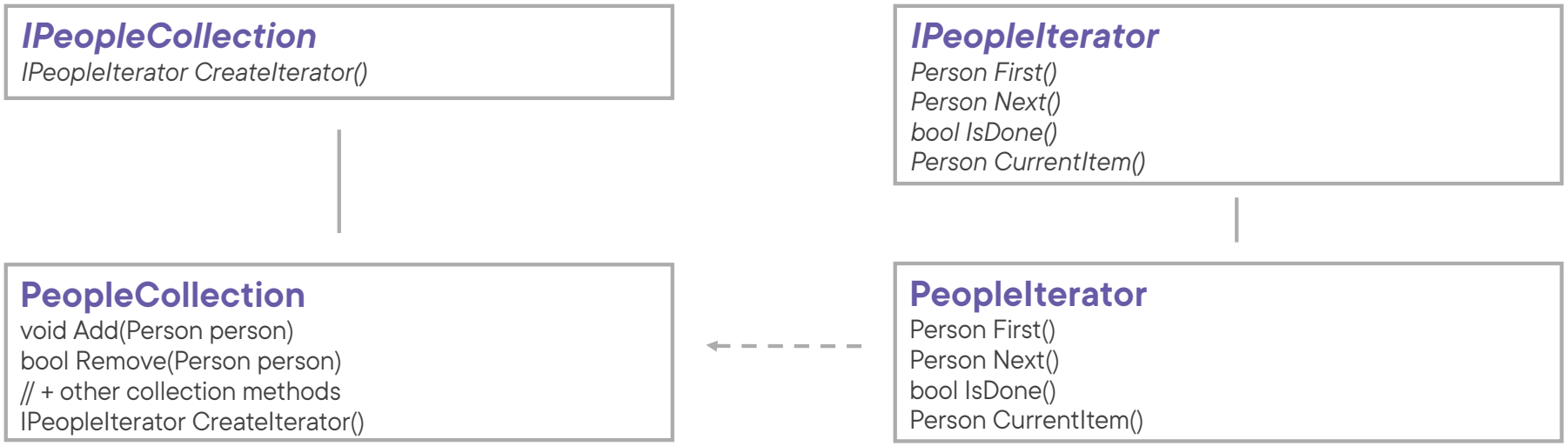
# Describing the Iterator Pattern



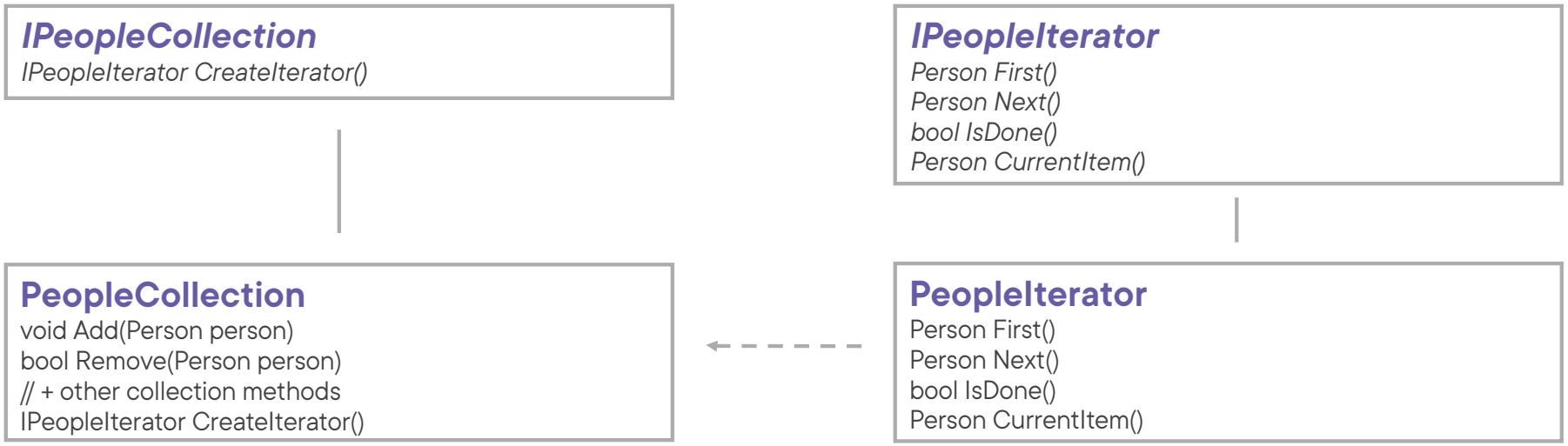
# Describing the Iterator Pattern



# Describing the Iterator Pattern

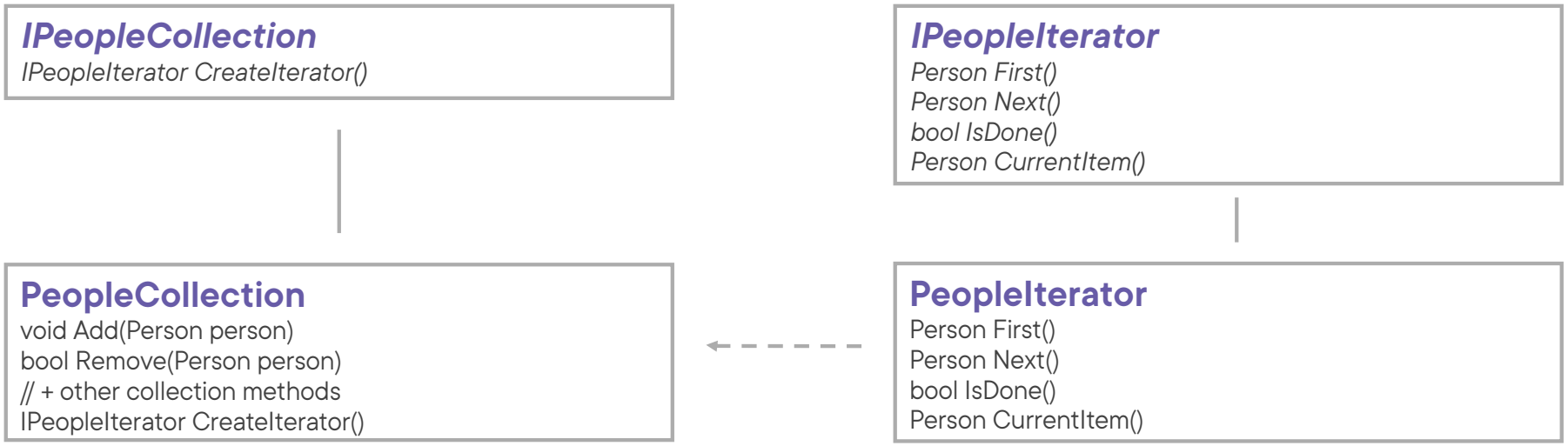


# Structure of the Iterator Pattern

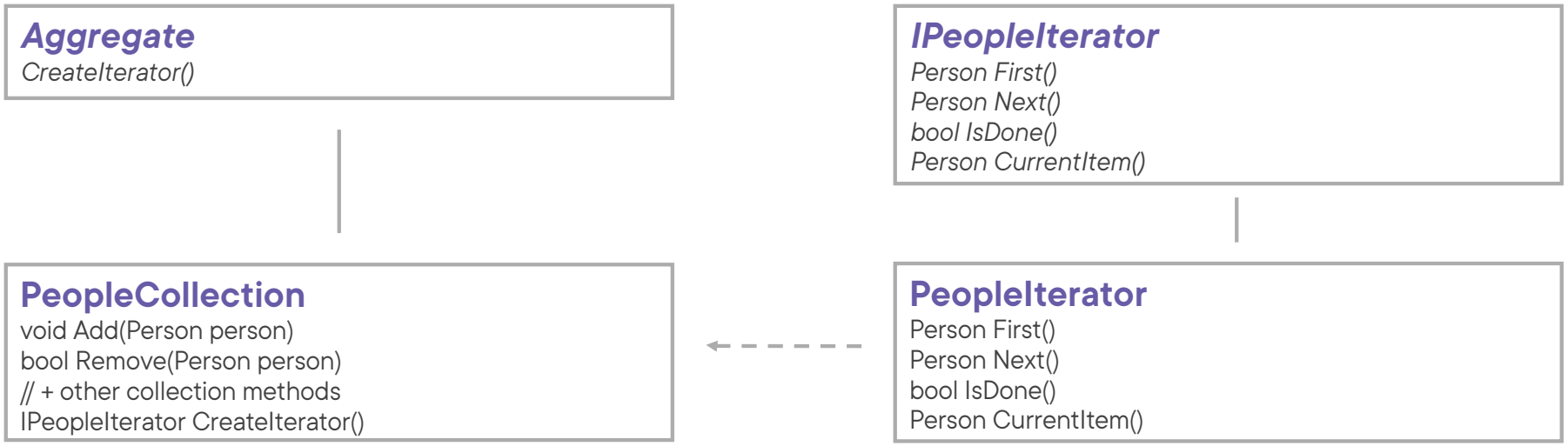


**Aggregate** defines an interface
for creating an **Iterator** object

# Structure of the Iterator Pattern

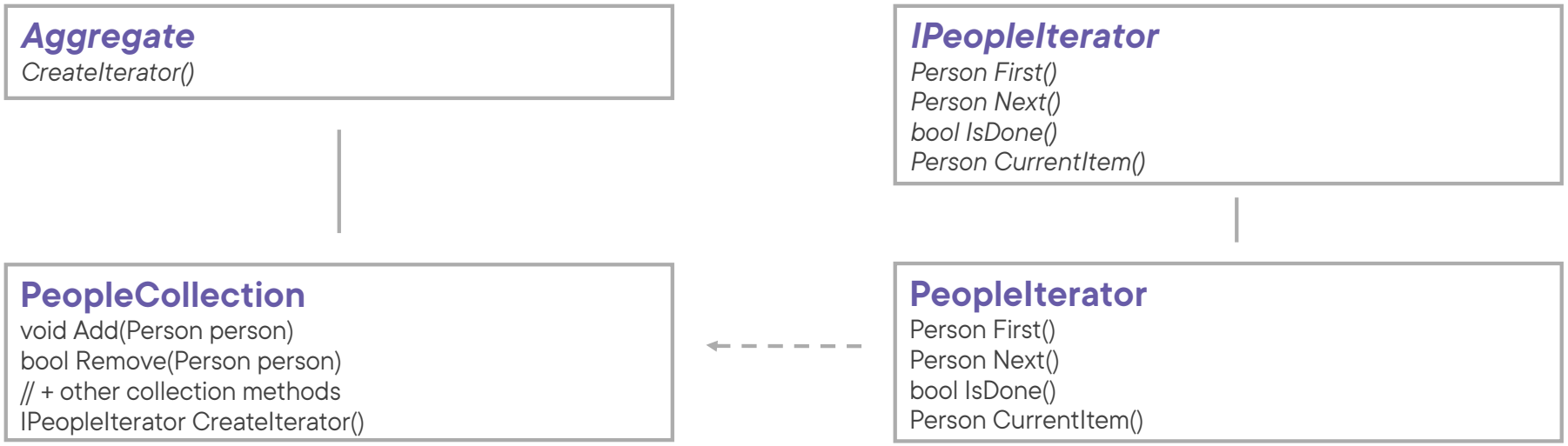


# Structure of the Iterator Pattern

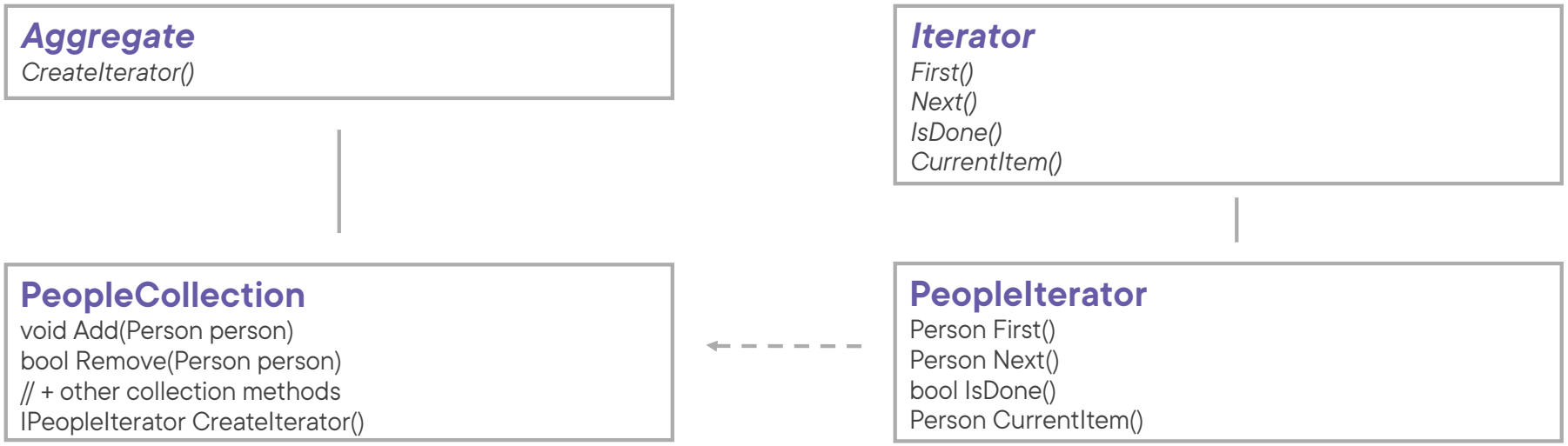


**Iterator** defines an interface for accessing and traversing elements

# Structure of the Iterator Pattern

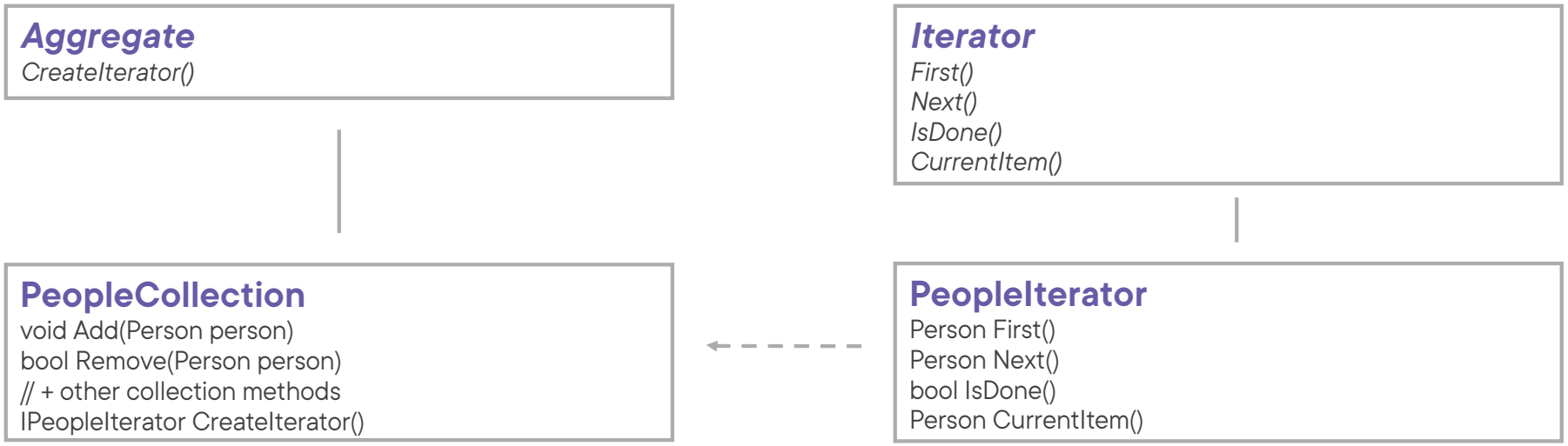


# Structure of the Iterator Pattern

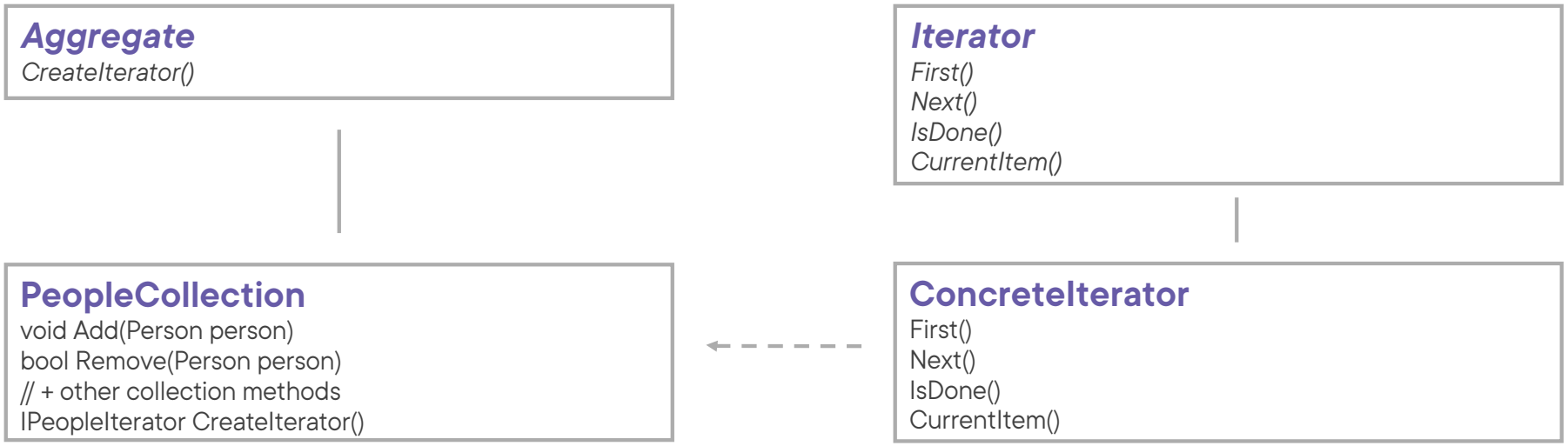


**ConcreteIterator** implements the **Iterator** interface and keeps track of the current position in the traversal of the **Aggregate**

# Structure of the Iterator Pattern

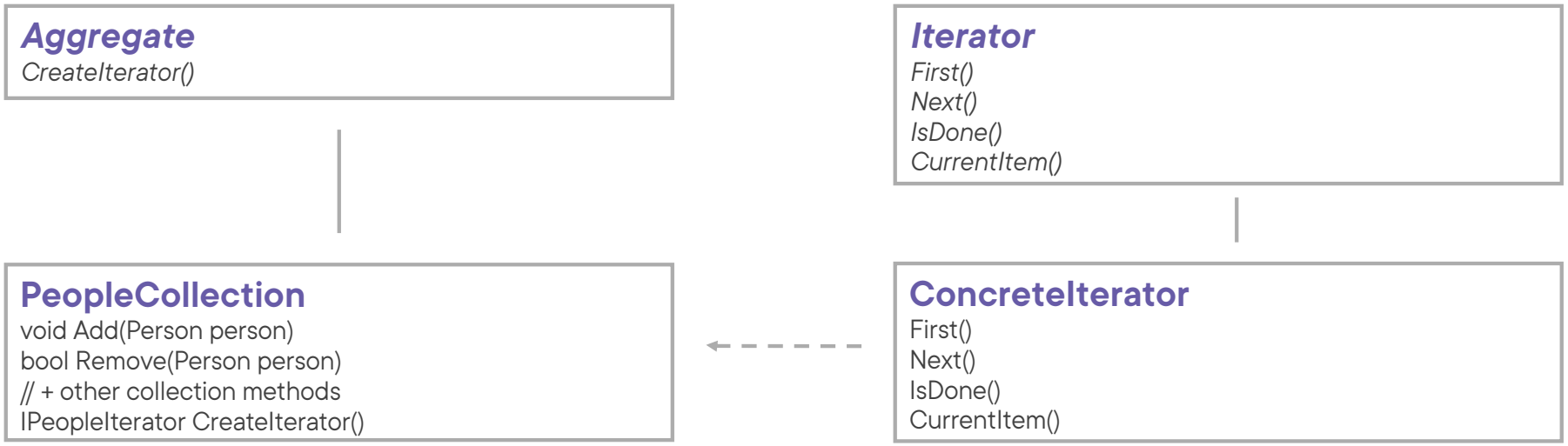


# Structure of the Iterator Pattern

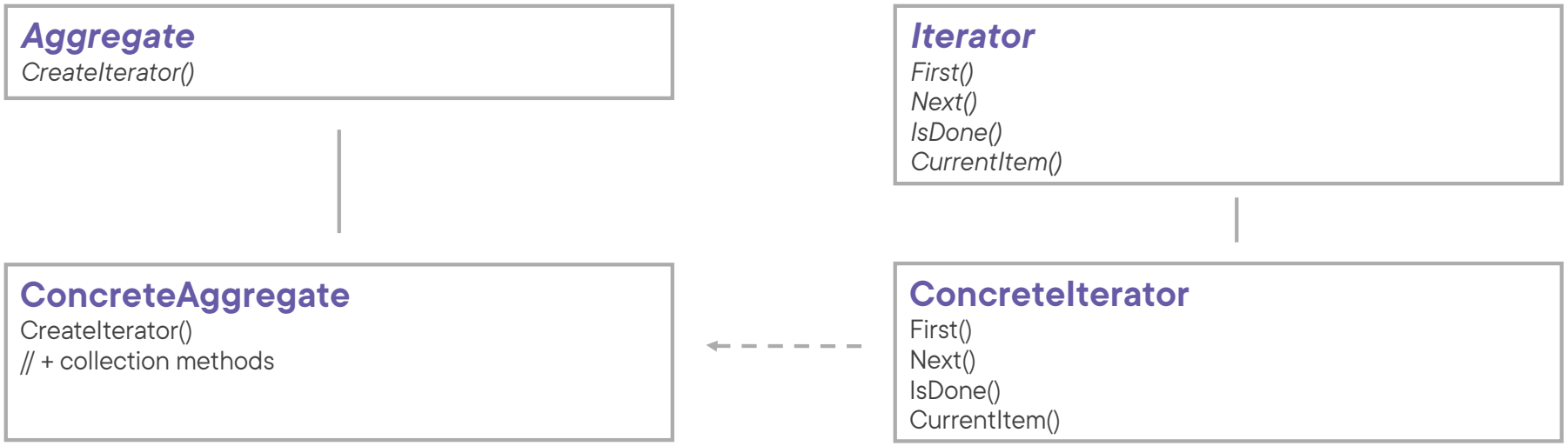


**ConcreteAggregate** implements the **Iterator** creation interface to return an instance of the proper **ConcreteIterator**

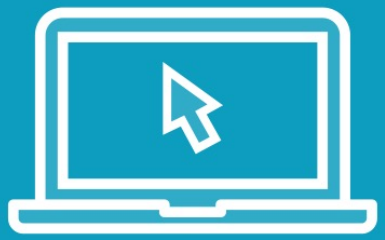
# Structure of the Iterator Pattern



# Structure of the Iterator Pattern



# Demo



## Implementing the iterator pattern

# Use Cases for the Iterator Pattern

**When you want to access an aggregate object's content without exposing its internal representation**

**When you want to support multiple ways of traversal for the same aggregate object**

**When you want to avoid code duplication in regards to traversing the aggregate object**

# Pattern Consequences

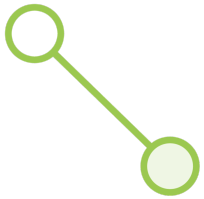
**Iterators simplify the interface of your aggregate structure as traversal code is separated out: single responsibility principle**

**You can implement new types of aggregate objects an iterators without them interfering with each other: open/closed principle**

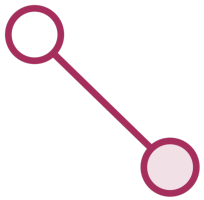
**Iterators can exist next to each other at the same time on the same collection**

**Can be a bit overkill when you only use simple traversals and collections**

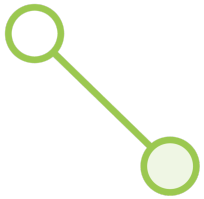
# Related Patterns



**Composite**
Iterators are often used to traverse its recursive structure



**Memento**
The memento can be used to store the state of the iterator and, potentially, roll it back



**Visitor**
You can use an iterator to traverse a potentially complex data structure, and apply logic to the items in that structure with a visitor

# Summary

**Intent of the iterator pattern:**

- To provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation

# Summary

**Implementation:**

- Leverage existing framework when possible

# Up Next:
# Behavioral Pattern: Visitor