SOFTWARE DESIGN TECHNIQUES (CSCN72040)

Week-6

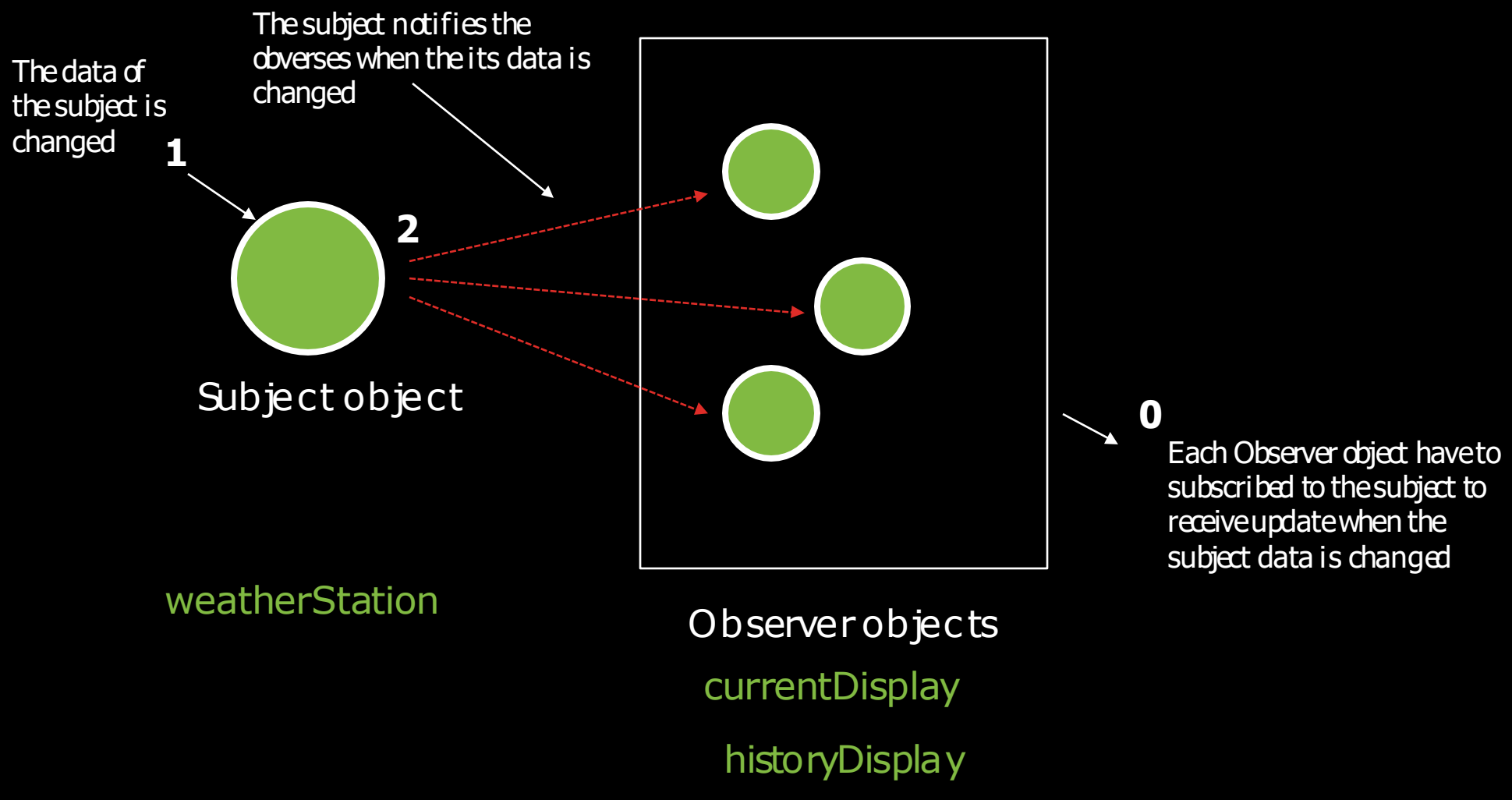
Behavioural Design Patterns:Part2

Petros Spachos

Review

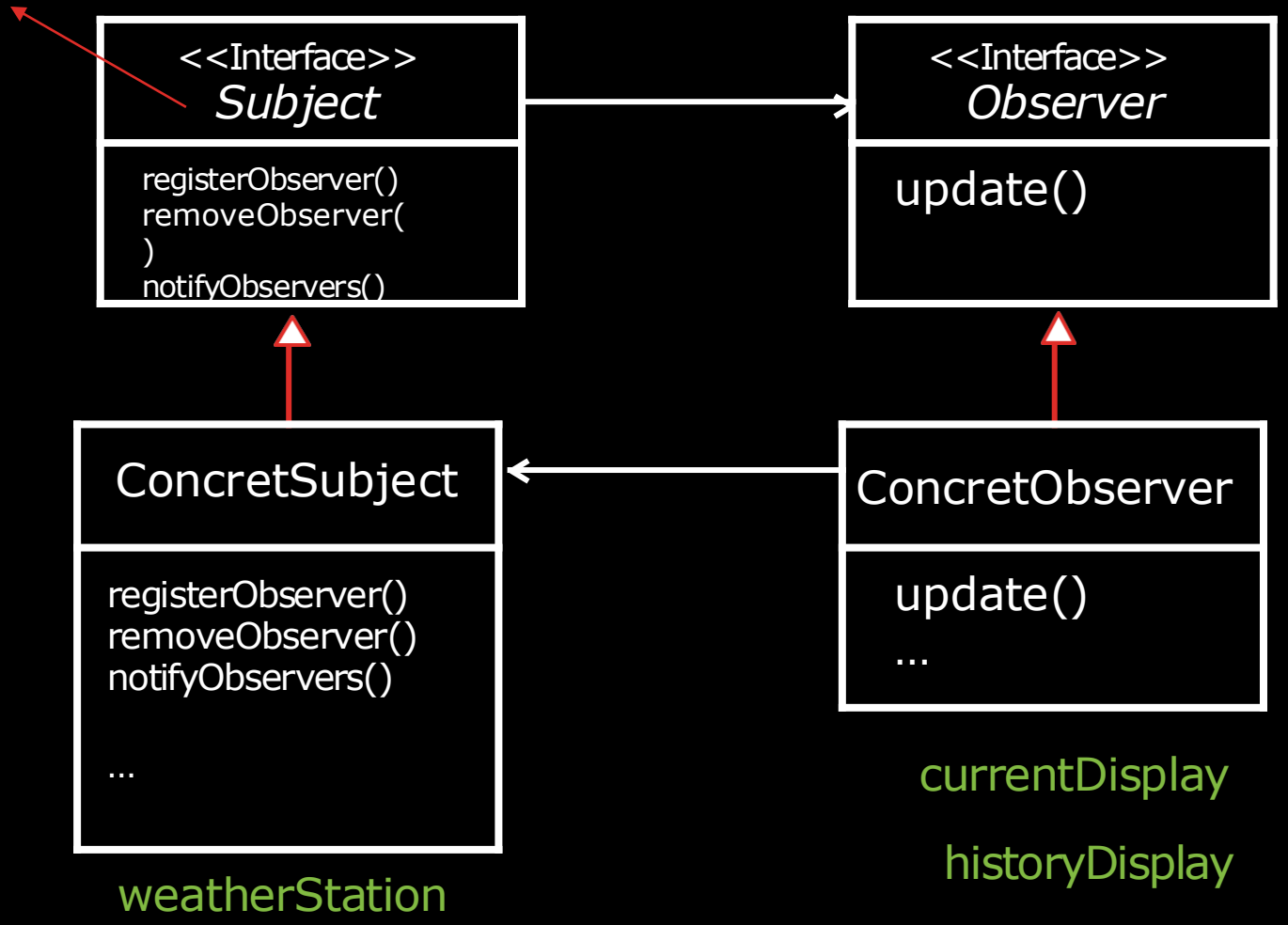
OBSERVER DESIGN PATTERN

OBSERVER DESIGN PATTERN

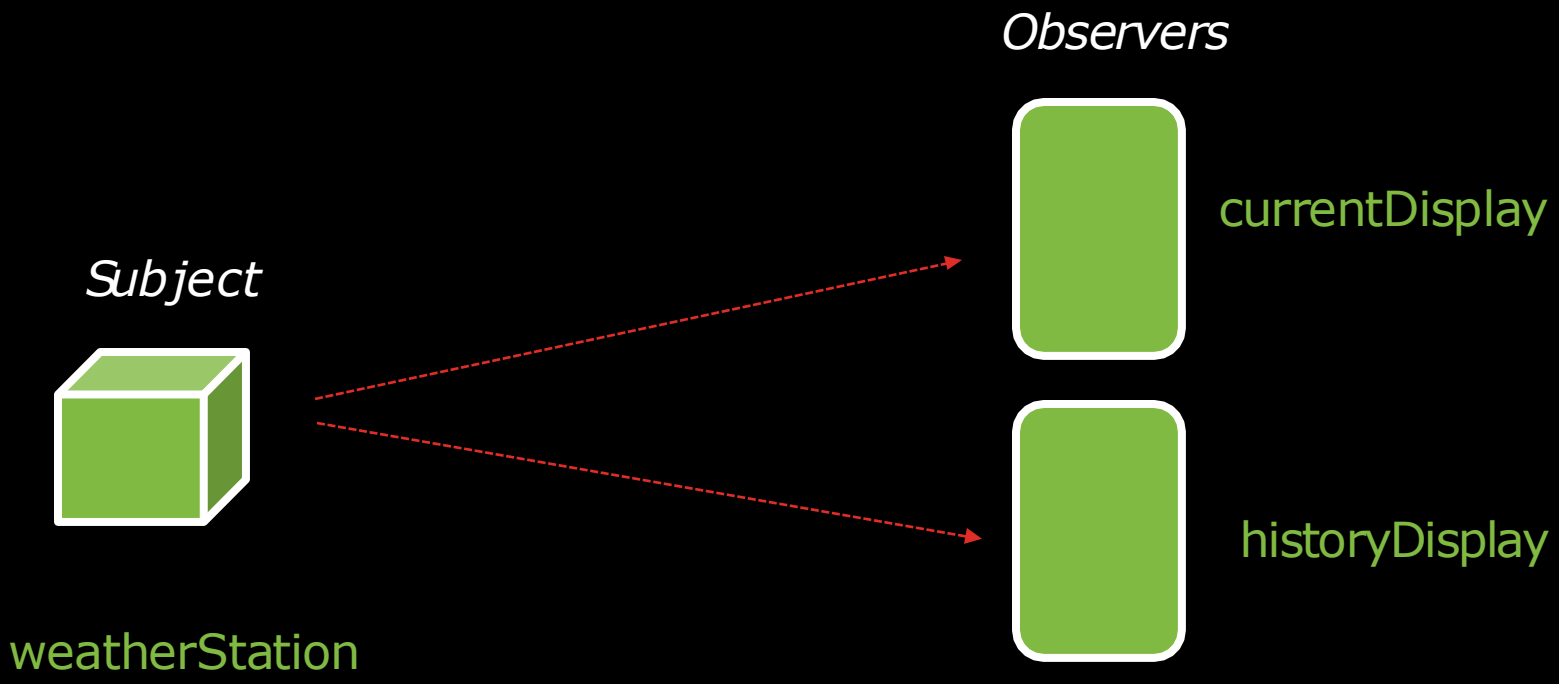


OBSERVER DESIGN PATTERN: CLASS DIAGRAM

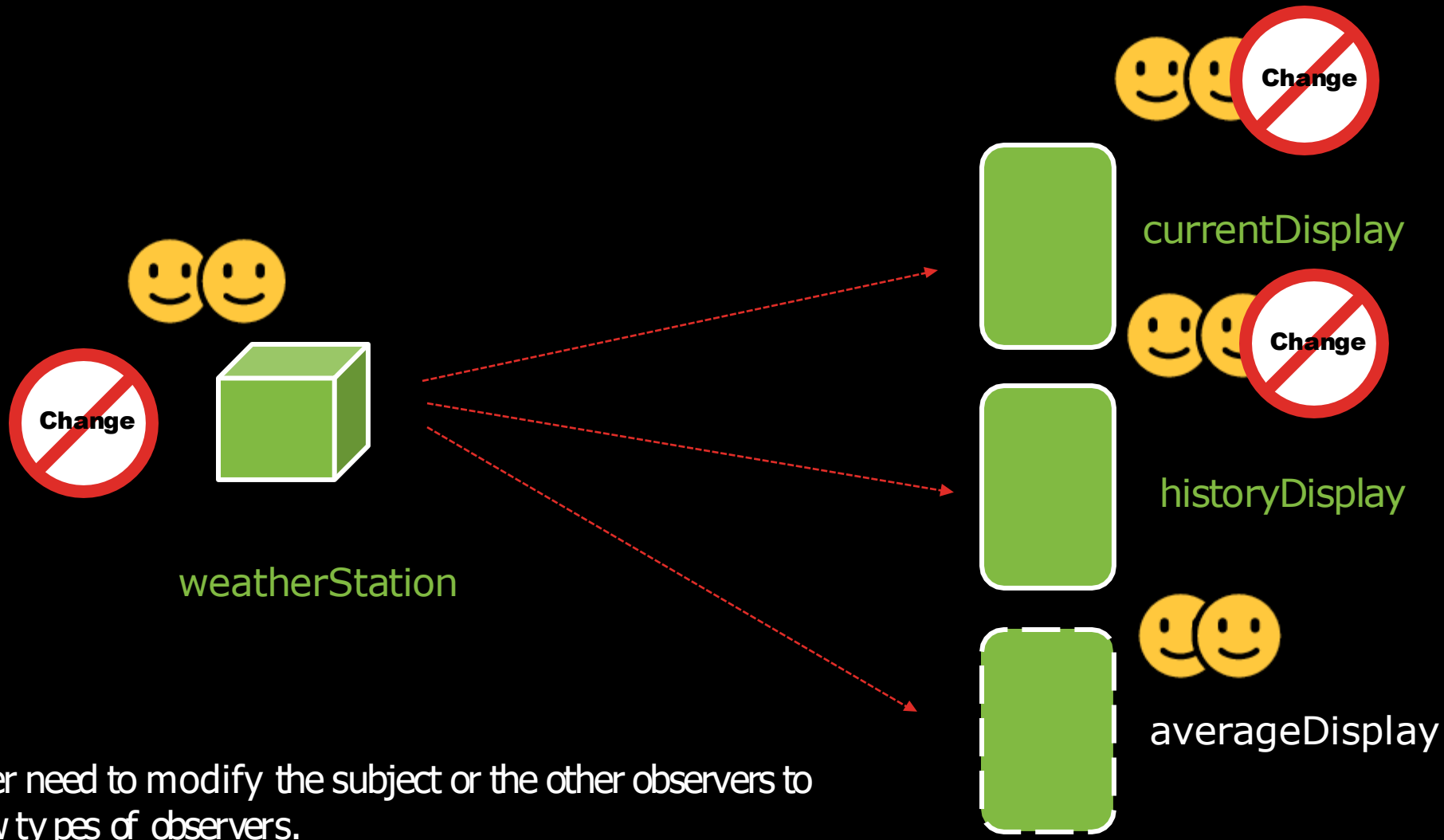
Also known as Observable



DEMO


Observers
currentDisplay
Subject
historyDisplay
weatherStation

DEMO



We never need to modify the subject or the other observers to add new types of observers.

OBSERVER DESIGN PATTERN

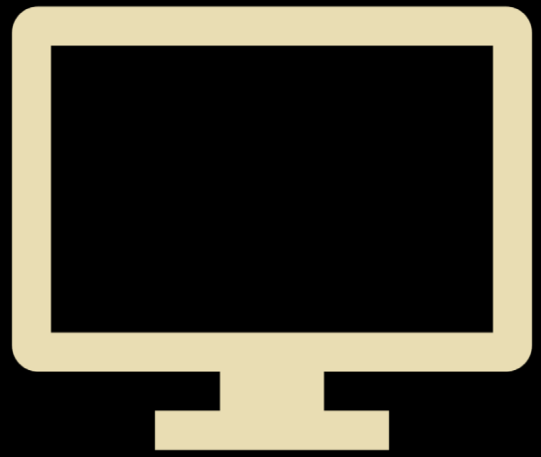
Let's walk through all the ways the pattern achieves loose coupling [3]:

- We can add new observers at any time
- We never need to modify the subject to add new types of observers.
- We can reuse subjects or observers independently of each other

Design principle

Aim for loosely coupled designs between objects

LIVE PROGRAMMING DEMO



OUTLINE

Introduction

Iterator design pattern

Describing iterator design pattern

Class diagram of the iterator design pattern

Live demo: Implementation of the iterator design pattern

Memento design pattern

Describing memento design pattern

Class diagram of the memento design pattern

Live demo: Implementation of the memento design pattern

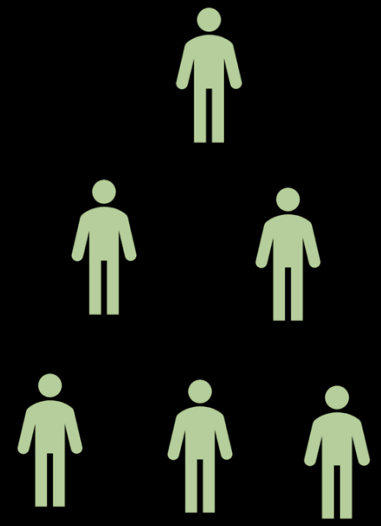
Summary and conclusion

INTRODUCTION

INTRODUCTION

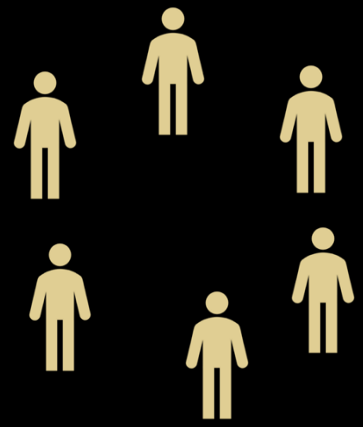
Service for groups

INTRODUCTION



Service for groups

INTRODUCTION

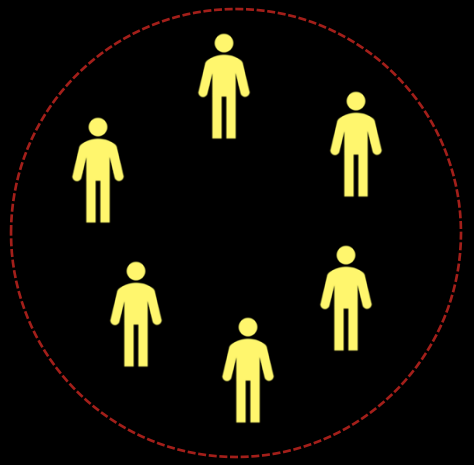


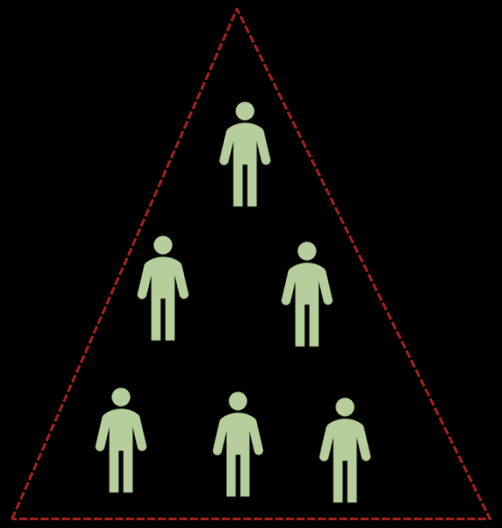
Service for groups

INTRODUCTION

Service for groups

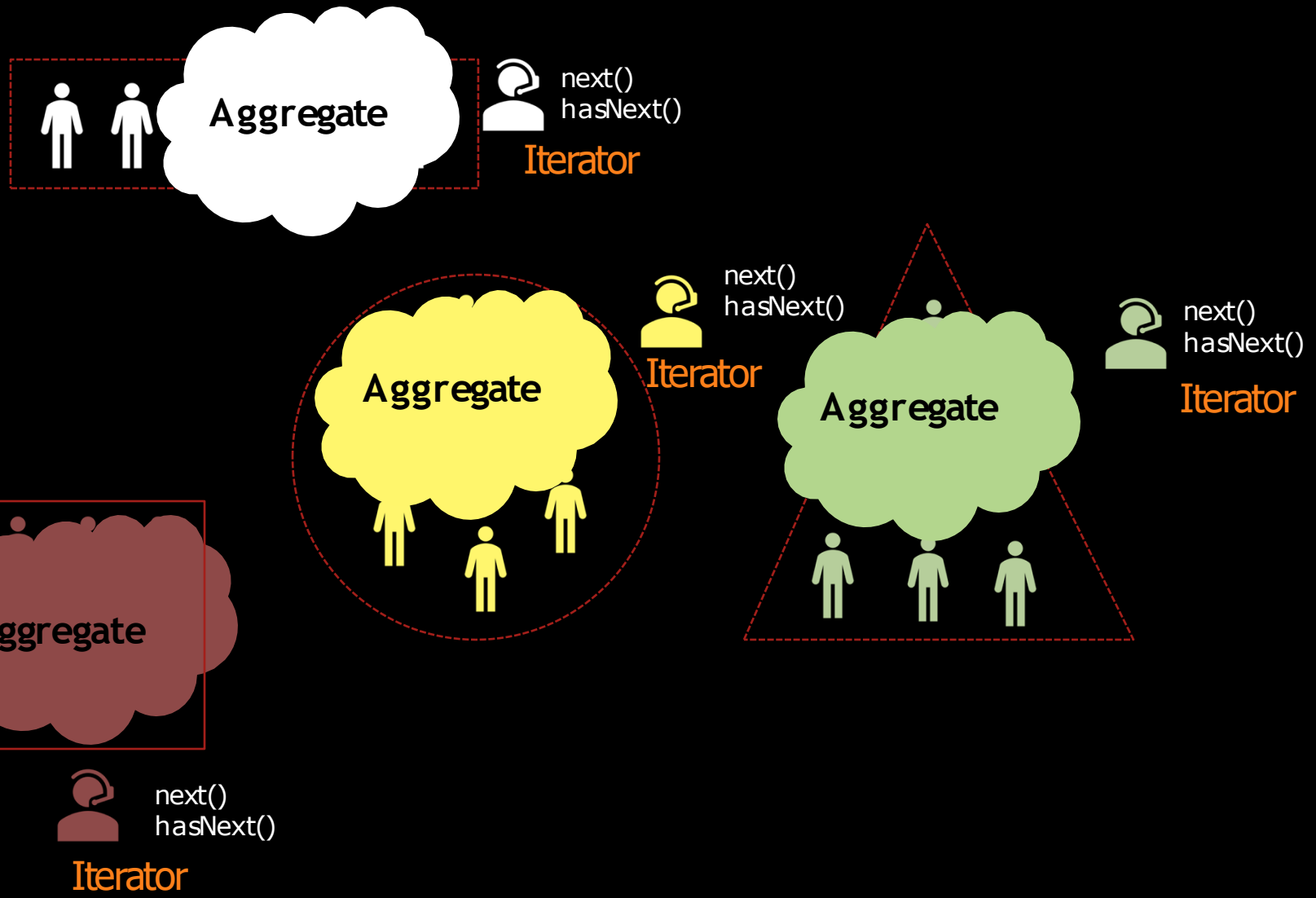
INTRODUCTION





Service for groups

INTRODUCTION



ITERATOR DESIGN PATTERN

ITERATOR DESIGN PATTERN

Intent

Provide a way to access the elements of an **aggregate** object sequentially without exposing its underlying representation [2].

Other Known names

Cursor

Motivation An aggregate object such as a list should give you a way to access its elements without exposing its internal structure. Moreover, you might want to traverse the list in different ways, depending on what you want to accomplish. [2].

Applicability

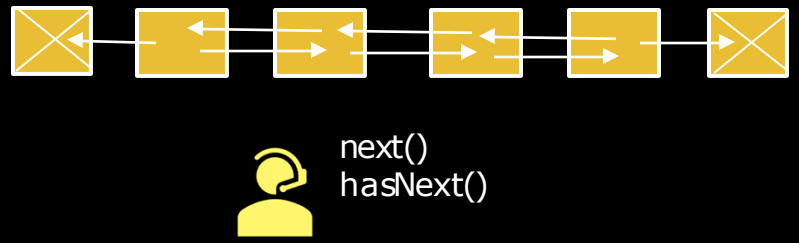
- to access an aggregate object's contents without exposing its internal representation.
- to support multiple traversals of aggregate objects.
- to provide a uniform interface for traversing different aggregate structures (that is, to support polymorphic iteration).

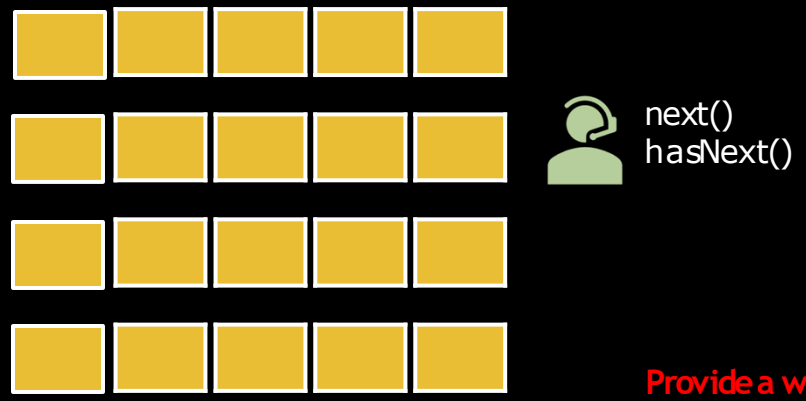
Participants

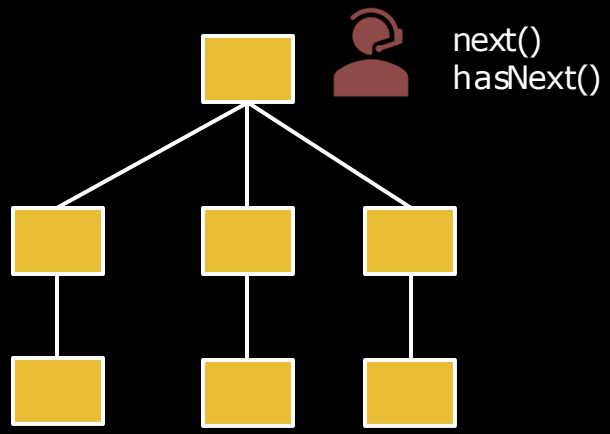
Iterator, ConcreteIterator , *Aggregate*(also known as *Iterable*) and ConcreteAggregate.

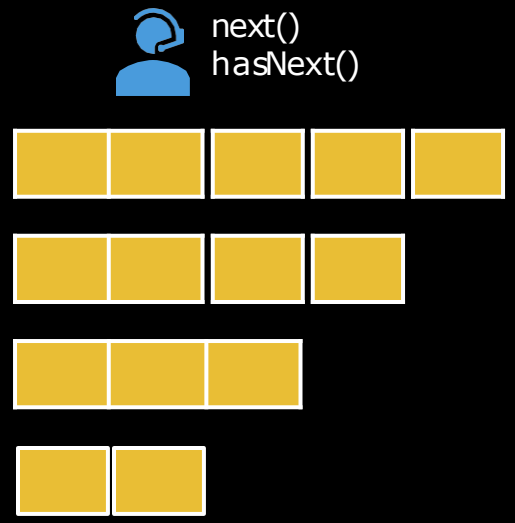
ITERATOR DESIGN PATTERN

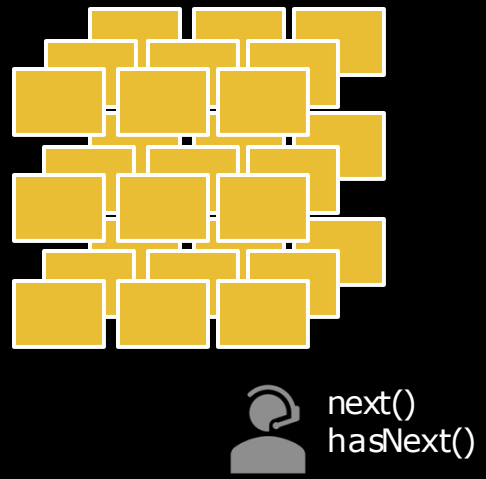
Different underlying representations





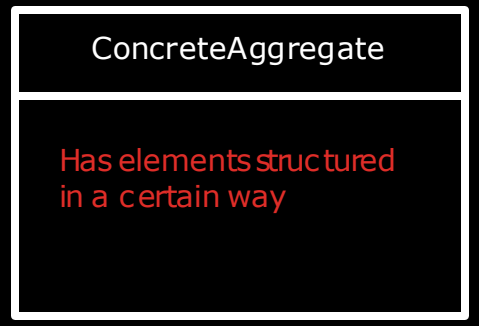


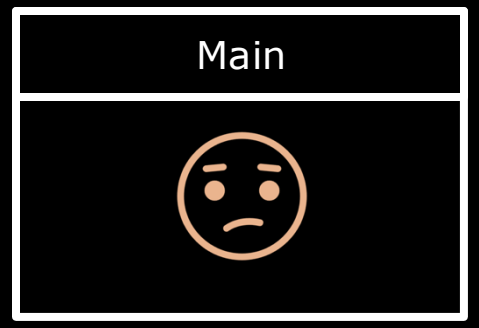




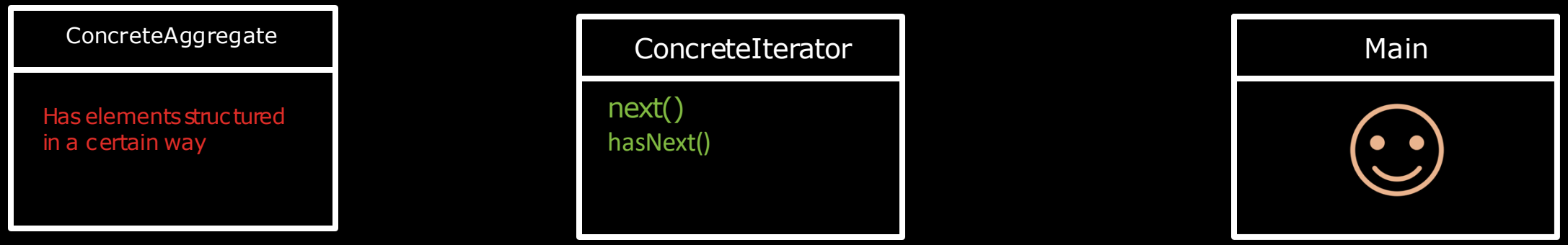
Provide a way to access the elements of an aggregateobject sequentially without exposing its underlying representation

ITERATOR DESIGN PATTERN



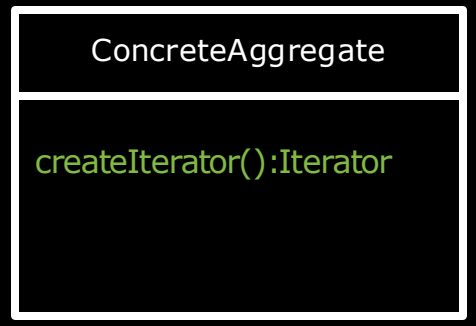


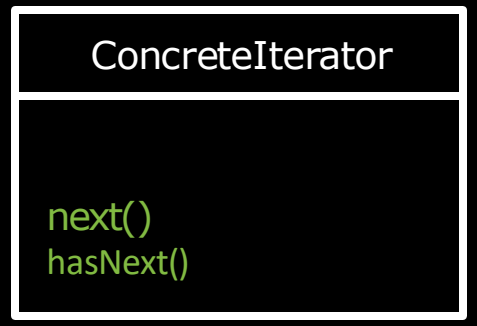
ITERATOR DESIGN PATTERN



ITERATOR DESIGN PATTERN

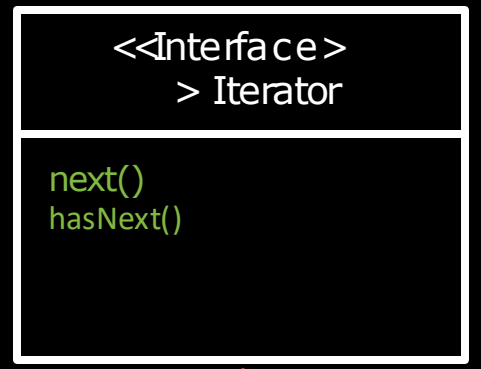
Creating the iterator is the responsibility of the ConcreteAggregate

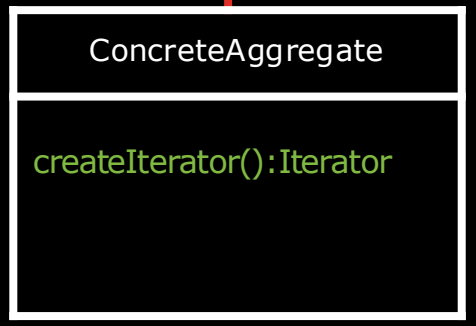


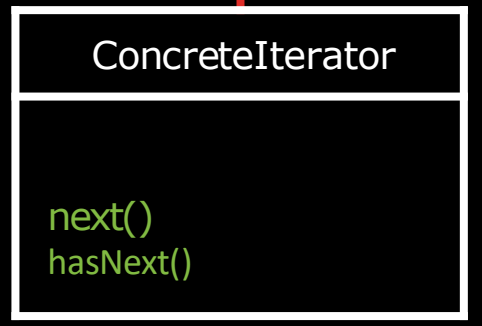


hasNext() to make sure we still have elements.
next() reruns the next element

ITERATOR DESIGN PATTERN



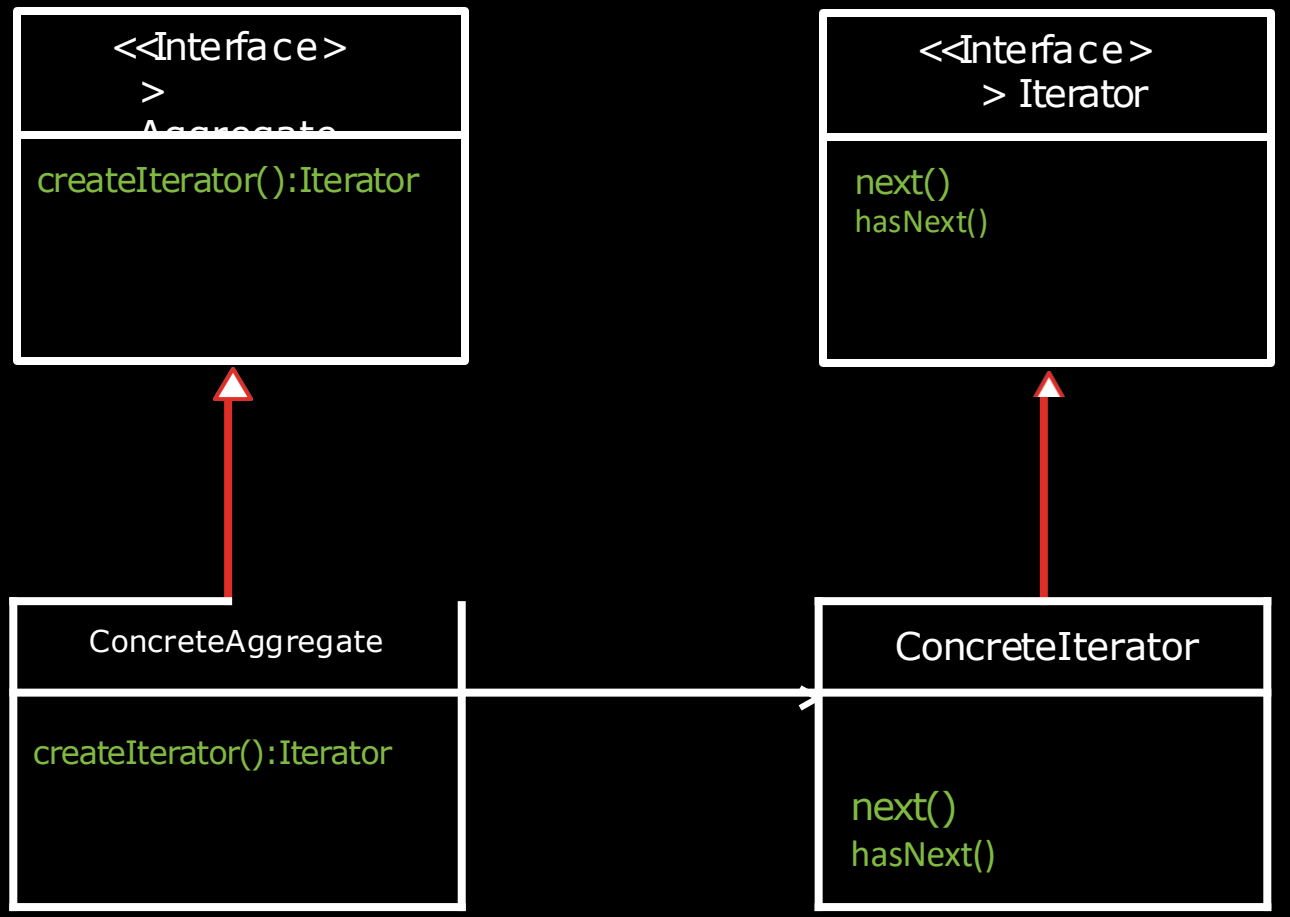




Creating the iterator is the responsibility of the ConcreteAggregate

hasNext() to make sure we still have elements.
next() reruns the next element

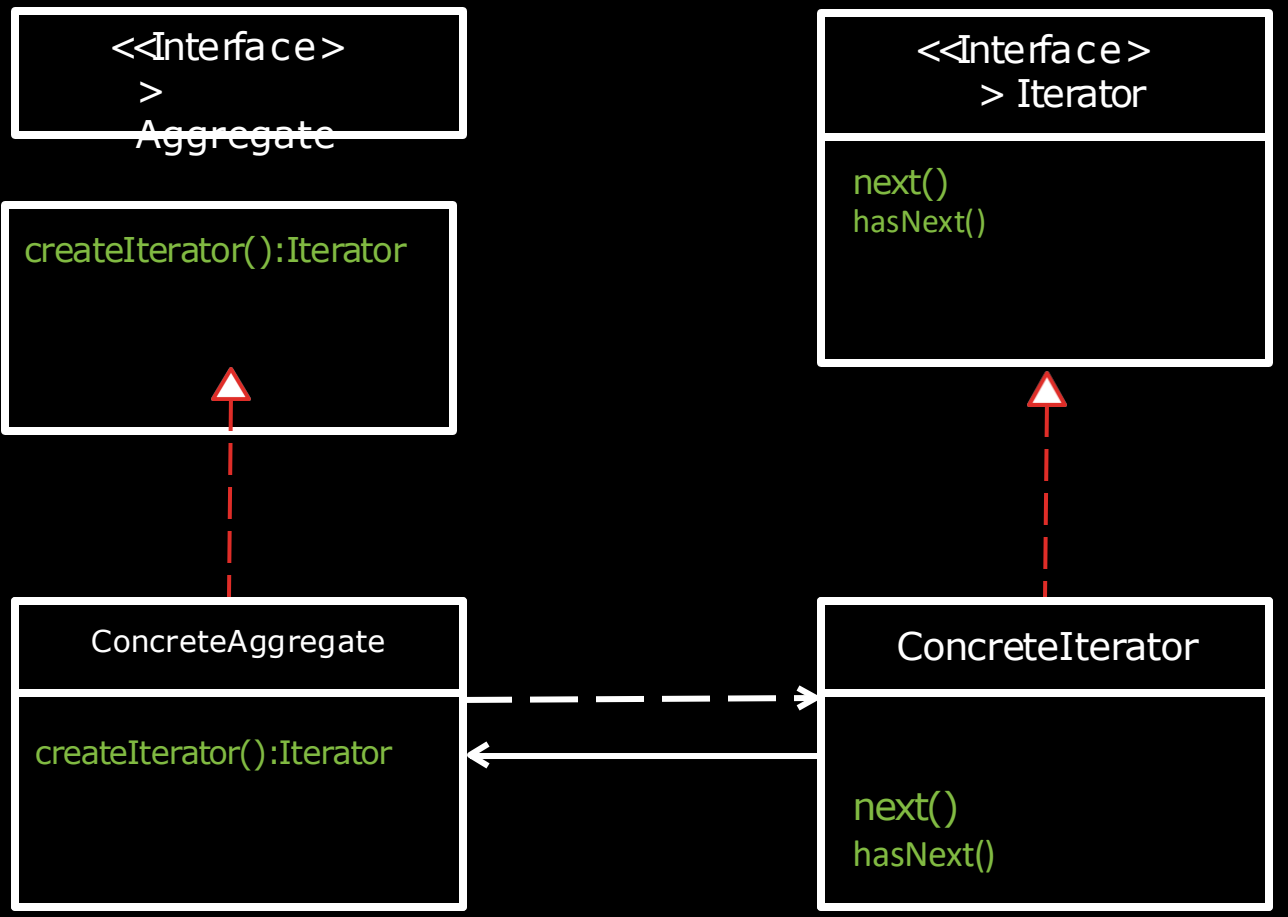
ITERATOR DESIGN PATTERN



Creating the iterator is the responsibility of the ConcreteAggregate

hasNext() to make sure we still have elements. next() returns the next element

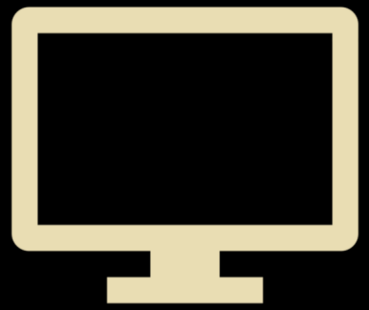
ITERATOR DESIGN PATTERN : CLASS DIAGRAM



Creating the iterator is the responsibility of the ConcreteAggregate

hasNext() to make sure we still have elements. next() returns the next element

LIVE DEMO



ITERATOR DESIGN PATTERN

Let’s walk through all the Consequences [2]

- It supports variations in the traversal of an aggregate.
- Iterators simplify the Aggregate interface. Iterator’s traversal interface obviates the need for a similar interface in Aggregate, thereby simplifying the aggregate’s interface.
- More than one traversal can be pending on an aggregate. An iterator keeps track of its own traversal state. Therefore, you can have more than one traversal in progress at once.

MEMENTO DESIGN PATTERN

MEMENTO DESIGN PATTERN

Intent

Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later [2].

Other Known names

Token

Motivation Sometimes it's necessary to record the internal state of an object. This is required when implementing checkpoints and undo mechanisms that let users back out of tentative operations or recover from errors. You must save state information somewhere so that you can restore objects to their previous states. But objects normally encapsulate some or all of their state, making it inaccessible to other objects and impossible to save externally. Exposing this state would violate encapsulation, which can compromise the application's reliability and extensibility [2].

Applicability

- a snapshot of (some portion of) an object's state must be saved so that it can be restored to that state later.

Participants

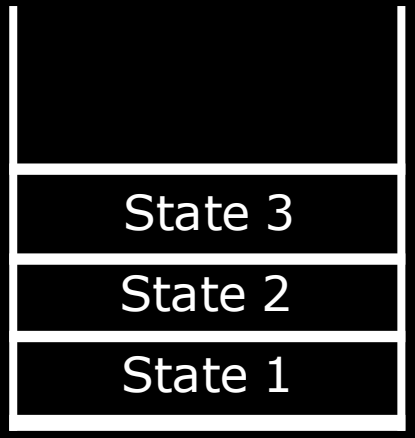
Originator:

- The object that we need to maintain the state.
- Creates a memento containing a snapshot of its current internal state.
- uses the memento to restore its internal state.

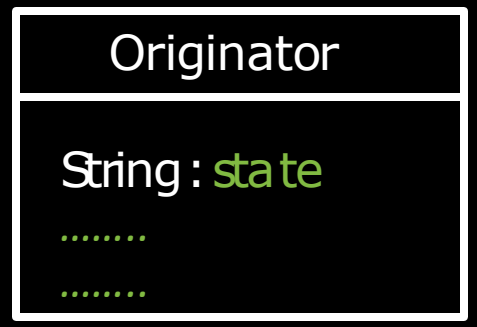
Memento: stores internal state of the Originator object.

Caretaker: is responsible for the memento's safekeeping.

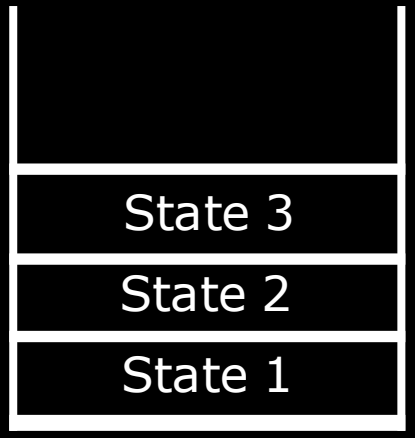
MEMENTO DESIGN PATTERN



MEMENTO DESIGN PATTERN



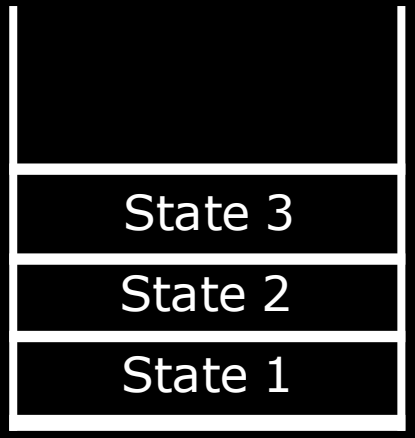
Handling the history of the state is the responsibility of the Caretaker



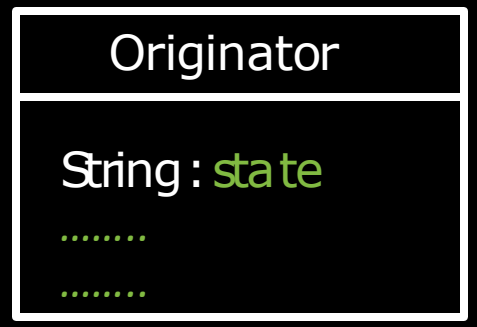
MEMENTO DESIGN PATTERN

stores the internal state of the Originator object

Handling the history of the state is the responsibility of the Caretaker

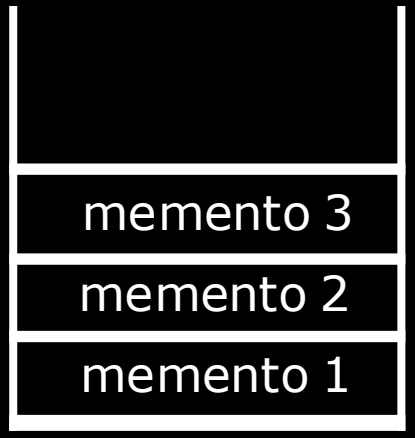


MEMENTO DESIGN PATTERN



stores the internal state of the Originator object

Handling the history of the state is the responsibility of the Caretaker

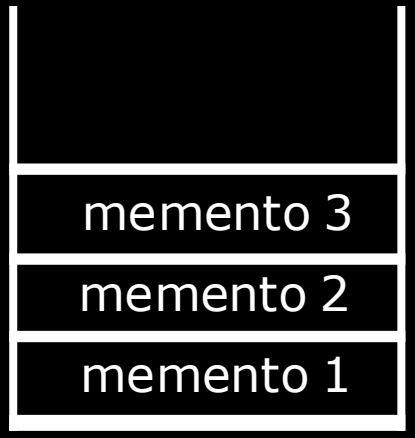


MEMENTO DESIGN PATTERN

Creating the memento is the responsibility of the Originator because it has access to all fields

stores the internal state of the Originator object

Handling the history of the state is the responsibility of the Caretaker

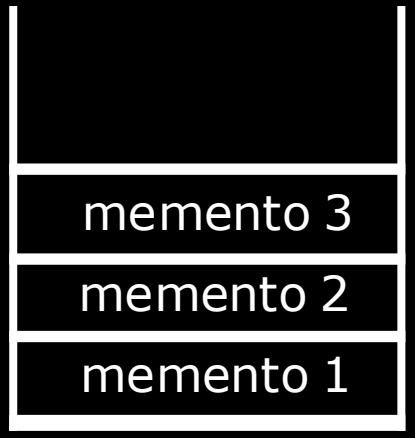


MEMENTO DESIGN PATTERN

Creating the memento is the responsibility of the Originator because it has access to all fields

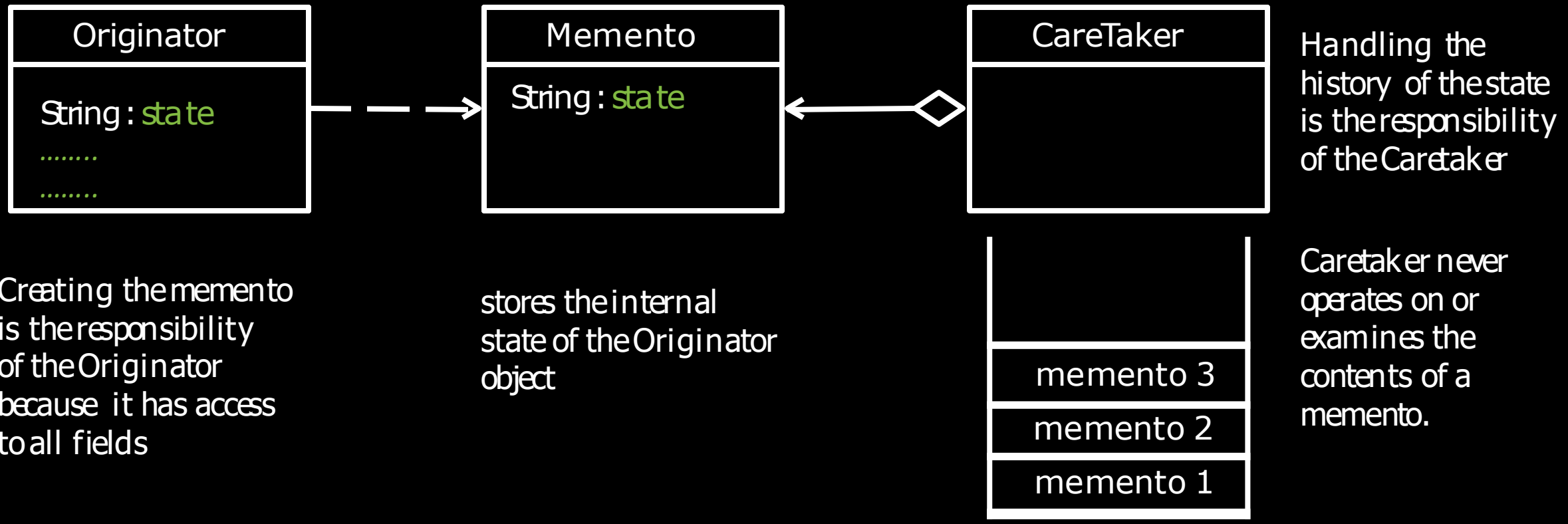
stores the internal state of the Originator object

Handling the history of the state is the responsibility of the Caretaker

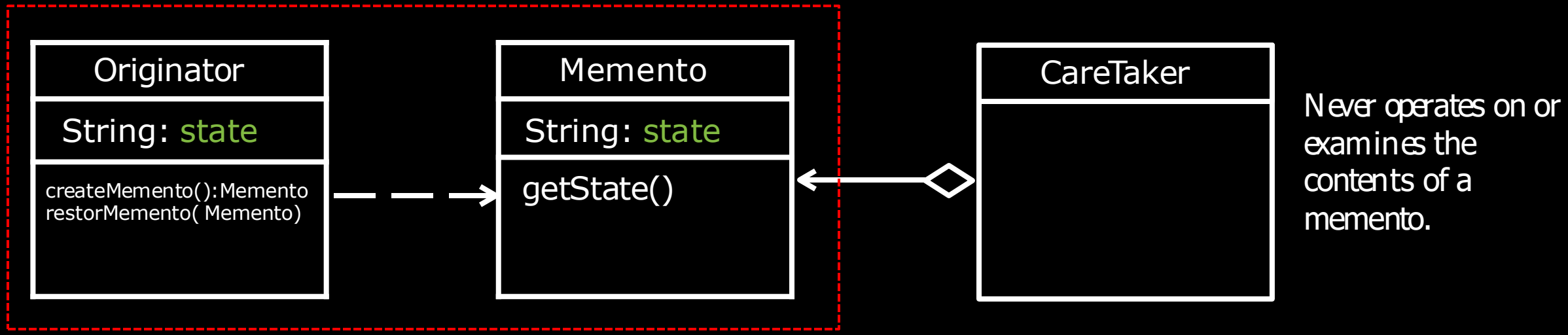


Caretaker never operates on or examines the contents of a memento.

MEMENTO DESIGN PATTERN



MEMENTO DESIGN PATTERN : CLASS DIAGRAM

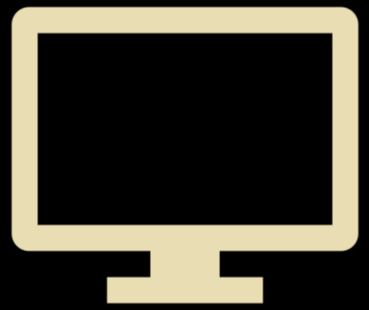


Creating the memento is the responsibility of the Originator because it has access to all fields

stores the internal state of the Originator object

Handling the history of the state is the responsibility of the Caretaker

DEMO



MEMENTO DESIGN PATTERN

Let’s walk through all the Consequences [2]

- Preserving encapsulation boundaries
- It simplifies Originator. We can reuse subjects or observers independently of each other
- Using mementos might be expensive. Mementos might incur considerable overhead if Originator must copy large amounts of information to store in the memento or if clients create and return mementos to the originator often enough

NEXT STEPS

To do

Further Reading

FURTHER READING

Will be posted in the course shell

REFERENCES

1. Christopher Alexander, Sara Ishikawa, Murray Silverstein, Max Jacobson, Ingrid Fiksdahl-King, and Shlomo Angel. A Pattern Language. Oxford University Press, New York, 1977.
2. Gamma, Erich, et al. "Design patterns: Elements of reusable object-oriented software (1995)
3. Freeman, Eric, et al. Head first design patterns. “O'Reilly Media, Inc.", 2020