Software Design Patterns

Lecture 10

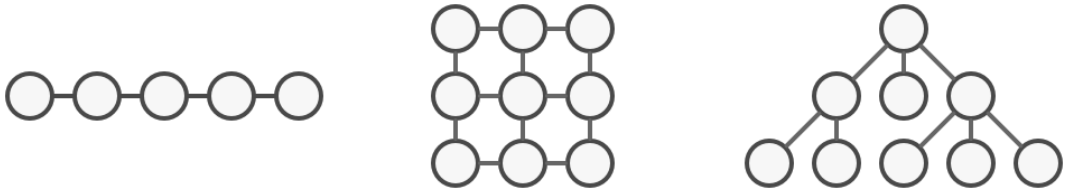
Iterator
Mediator

Dr. Fan Hongfei

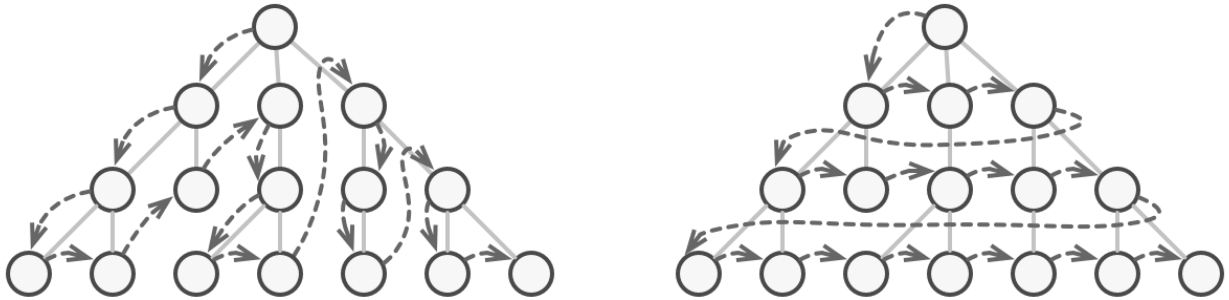
7 November 2024

Iterator: Problem

- Collections are one of the most used data types in programming



- There should be a way to go through each element of the collection

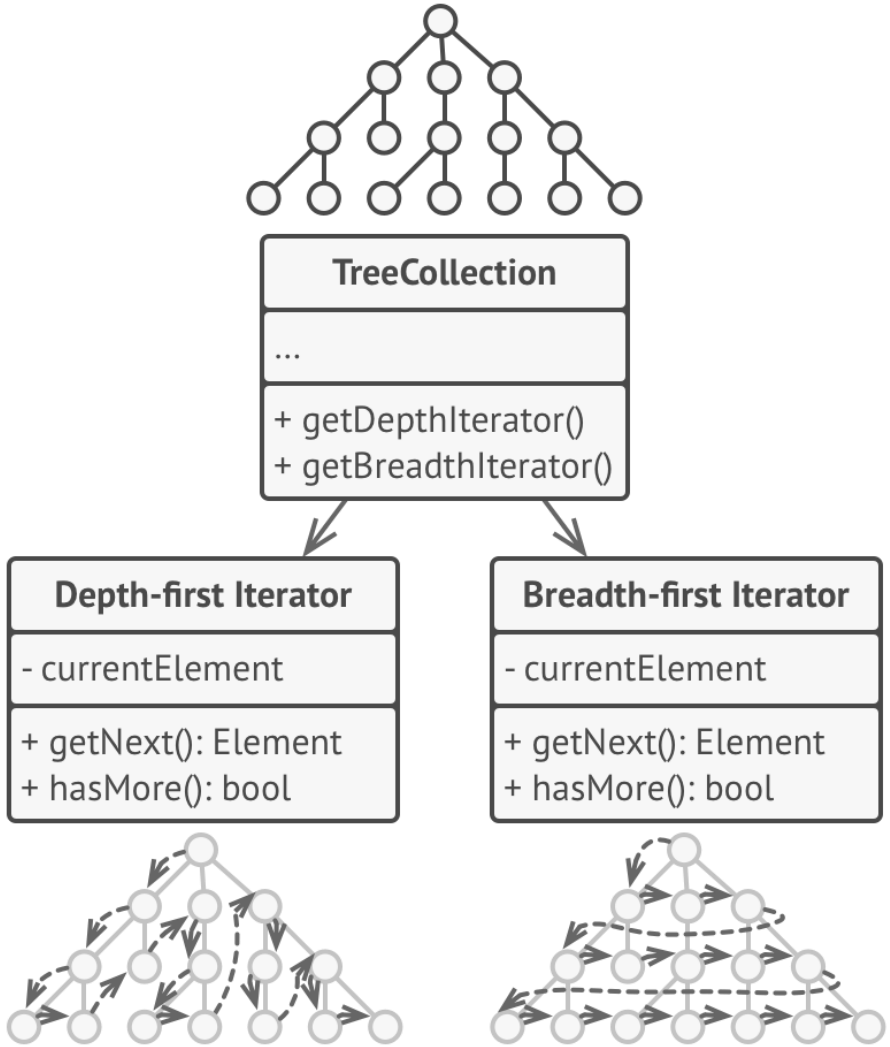


- More traversal algorithms to the collection **blurs its primary responsibility**
- Some algorithms might be **tailored** for a specific application
- The client code **may not care about** how the elements are stored

[Figures in this slide are extracted from https://refactoring.guru/design-patterns/iterator]

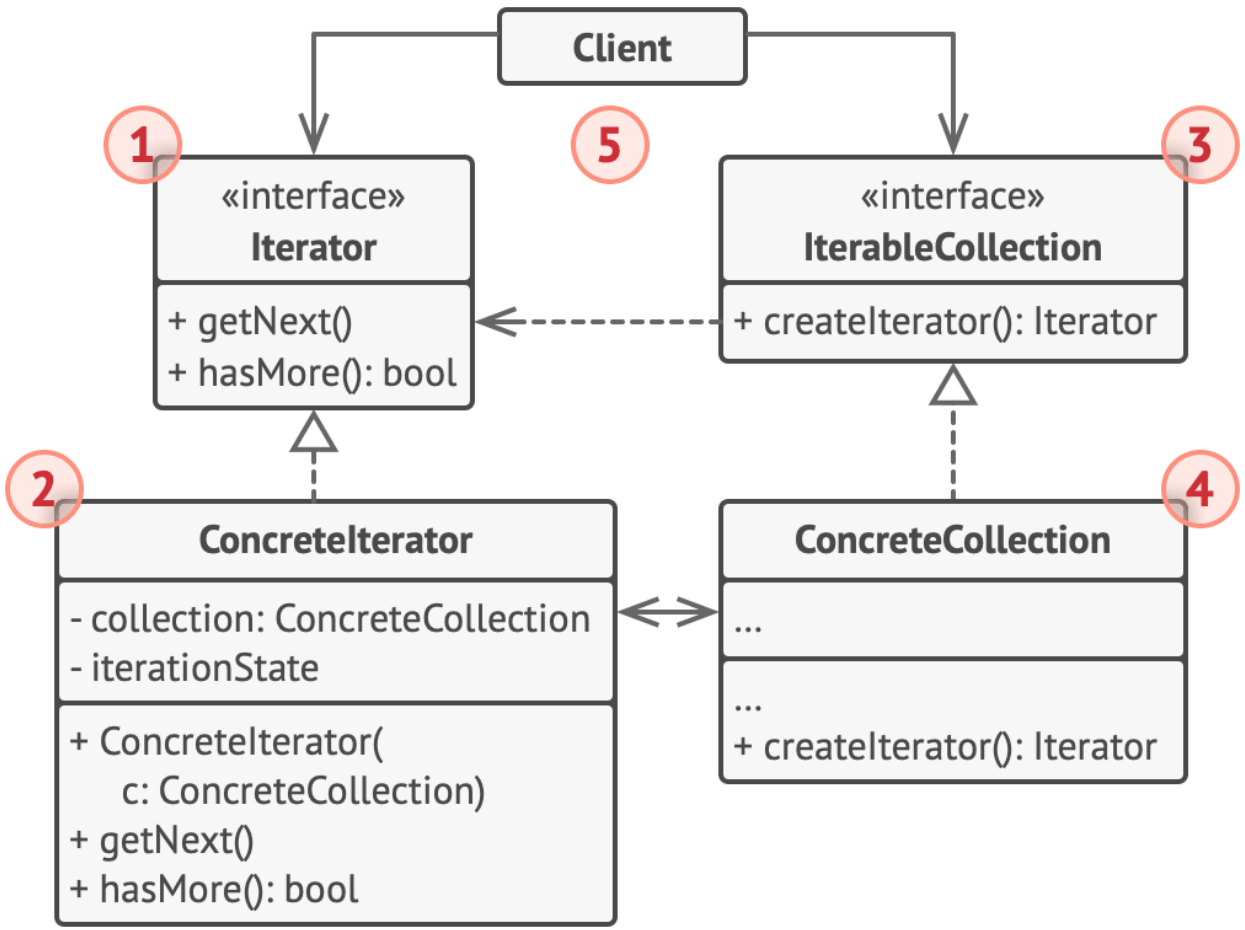
Iterator: Solution

- Main idea: extract the **traversal behavior** of a collection into a separate object called an **iterator**
- Iterator encapsulates all traversal details
- Several iterators can go through the same collection at the same time
- Providing one primary method for fetching elements from the client
- All iterators implement the same interface



[Figures in this slide are extracted from https://refactoring.guru/design-patterns/iterator]

Iterator: Structure

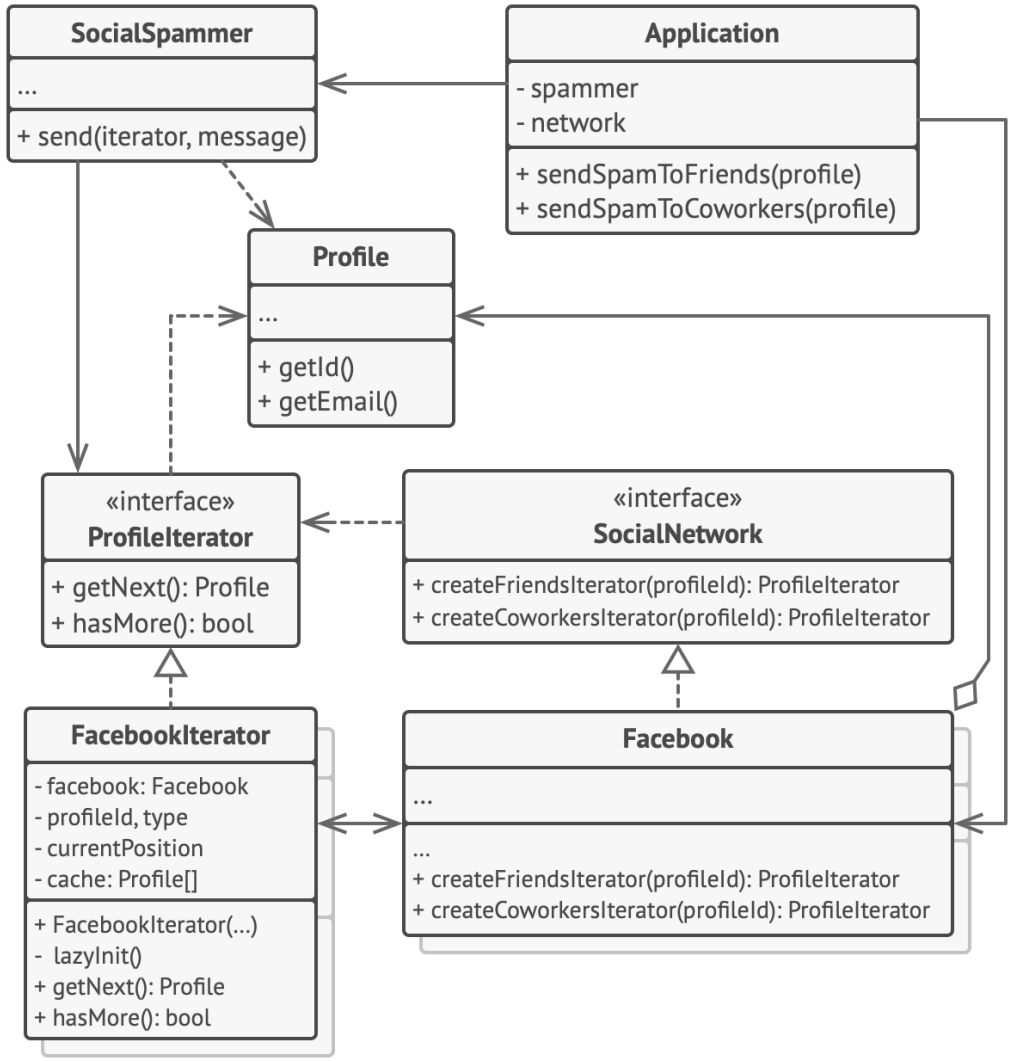


1. **Iterator:** interface declaring traversal operations
2. **Concrete Iterators:** implementing specific traversal algorithms, managing traversal progress
3. **Collection:** declaring methods for getting compatible iterators
4. **Concrete Collections:** returning new instances of particular iterator classes

[Figures in this slide are extracted from https://refactoring.guru/design-patterns/iterator]

Iterator: Example

- Walking through a special kind of collection which encapsulates access to Facebook's social graph
- The client utilizes the iterator rather than the whole collection
- The way the client works with the collection can be changed at runtime



[Figures in this slide are extracted from https://refactoring.guru/design-patterns/iterator]

Iterator: Applicability

- When the collection has a complex data structure, but you want to hide its complexity from clients (either for convenience or security)
  - The iterator provides the client with several simple methods of accessing the collection elements
- Reduce duplication of the traversal code
  - The code of non-trivial iteration algorithms may blur the responsibility of the original code and make it less maintainable
- Traverse different (and even unknown) data structures
  - A couple of generic interfaces for both collections and iterators

Iterator: Implementation

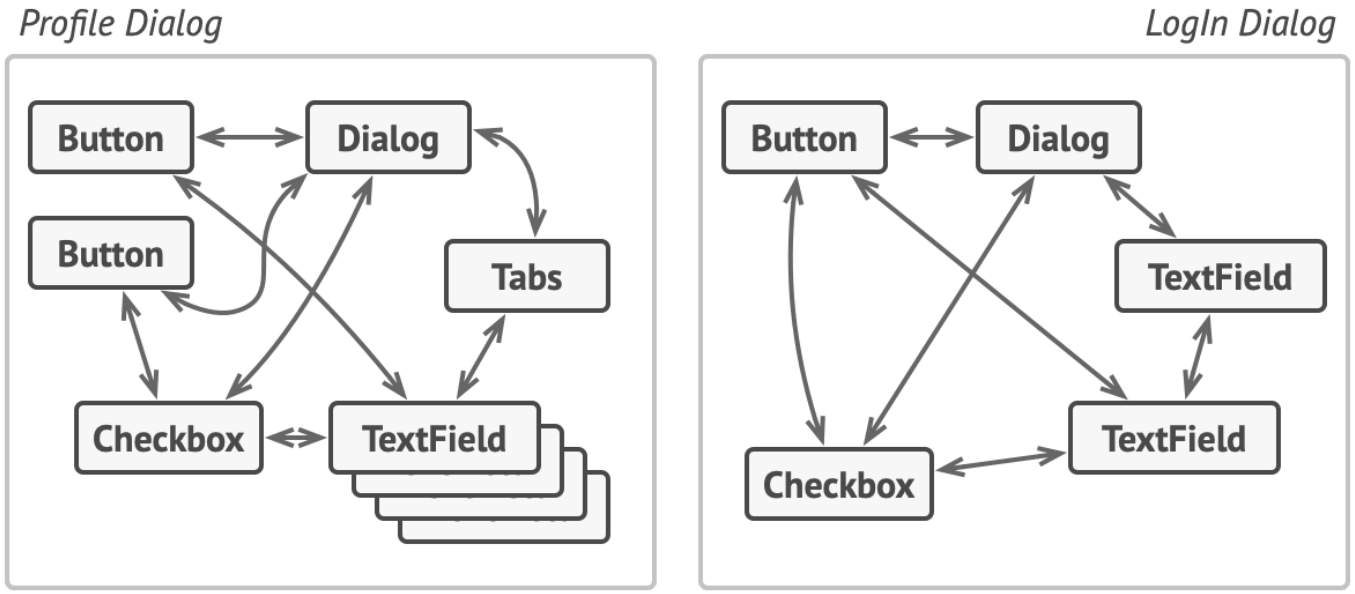
1. Declare the **iterator interface**, with at least one method for fetching the next element from a collection
   - May be extended: fetching the previous element, tracking the current position, and checking the end of the iteration, etc.
2. Declare the **collection interface** and describe a method for fetching iterators
3. Implement **concrete iterator classes** for the collections
   - An iterator object must be linked with a single collection instance
4. Implement the collection interface
   - Provide the client with a shortcut for creating iterators, tailored for a particular collection class
   - The collection object must pass itself to the iterator's constructor to establish a link
5. The client fetches a new iterator object each time it needs to iterate over the collection

Iterator: Pros and Cons

- **Pros**
  - Single Responsibility Principle: separate traversal algorithms from collections
  - Open/Closed Principle: implementing new types of collections and iterators without changing existing code
  - Iterate over the same collection in parallel
  - Delay an iteration and continue it when needed
- **Cons**
  - Can be an overkill if the app only works with simple collections
  - May be less efficient than going through elements directly

Mediator: Problem

- **Example: a dialog for creating and editing customer profiles**

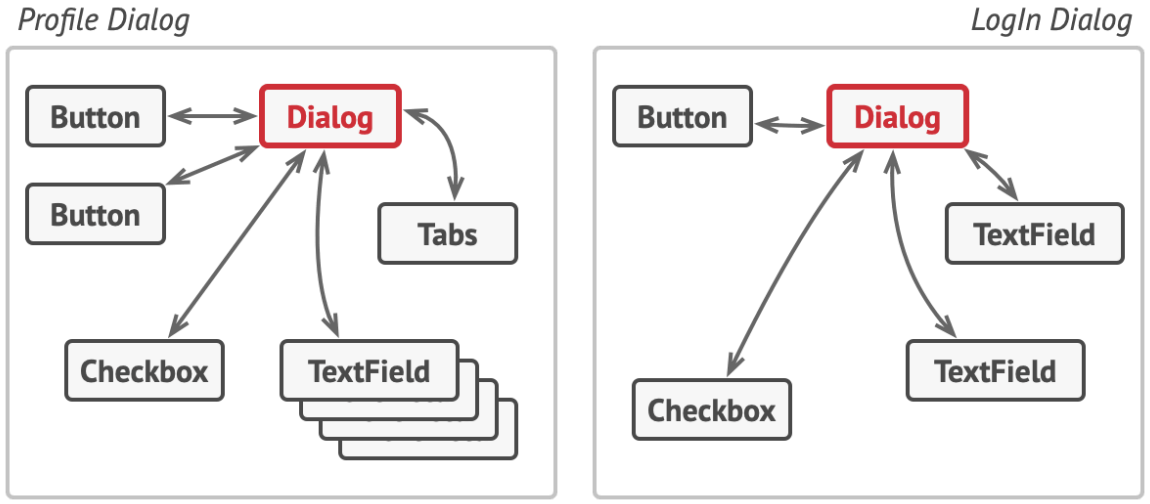


  - Elements may interact with each other
  - Changes to one element may affect others
  - Elements become harder to reuse

[Figures in this slide are extracted from https://refactoring.guru/design-patterns/mediator]

Mediator: Solution

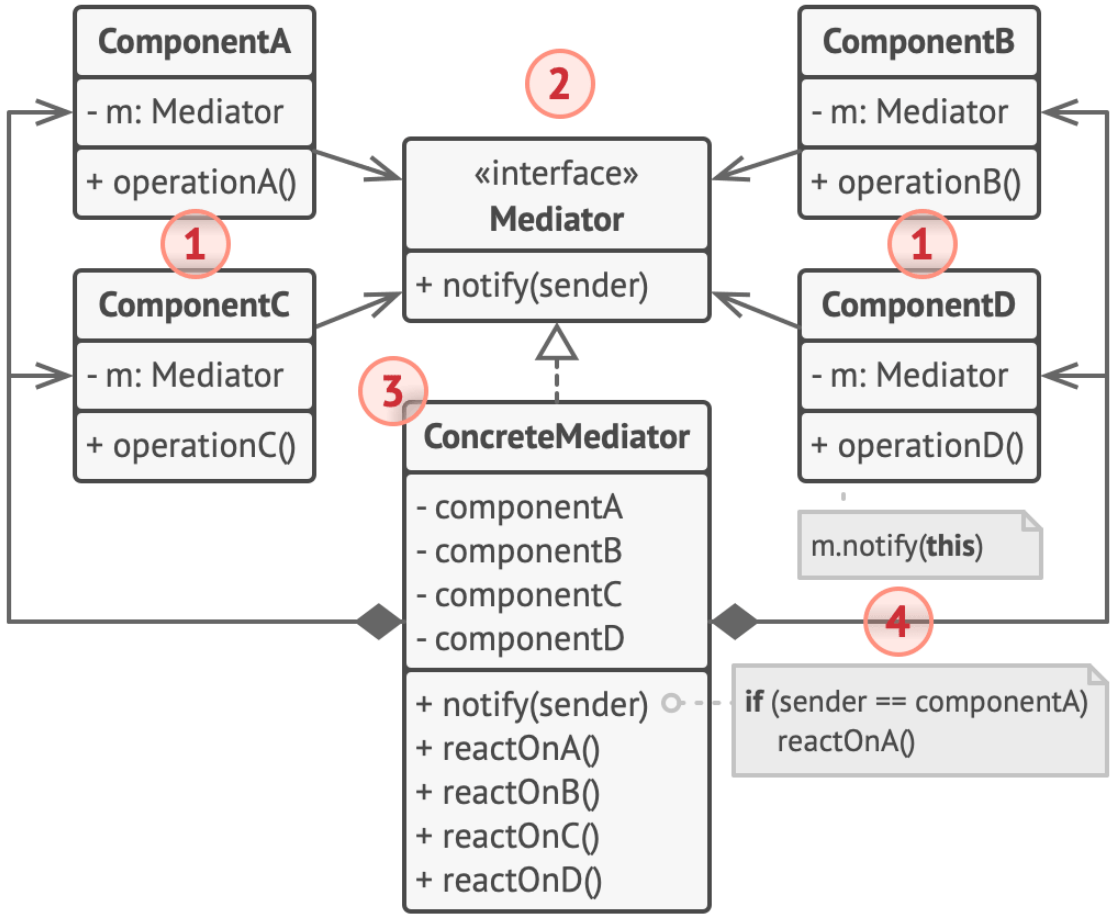
- **Mediator** (aka Intermediary or Controller)
  - Cease direct communications among components
  - Collaborate indirectly by calling a special mediator object



- **Significant changes** happen in the form elements
  - Notify the dialog about the event
  - Dialog preforms tasks, or pass tasks to other elements
- **Further improvement**
  - Make the dependency looser by extracting the common interface for all dialogs

[Figures in this slide are extracted from https://refactoring.guru/design-patterns/mediator]

Mediator: Structure

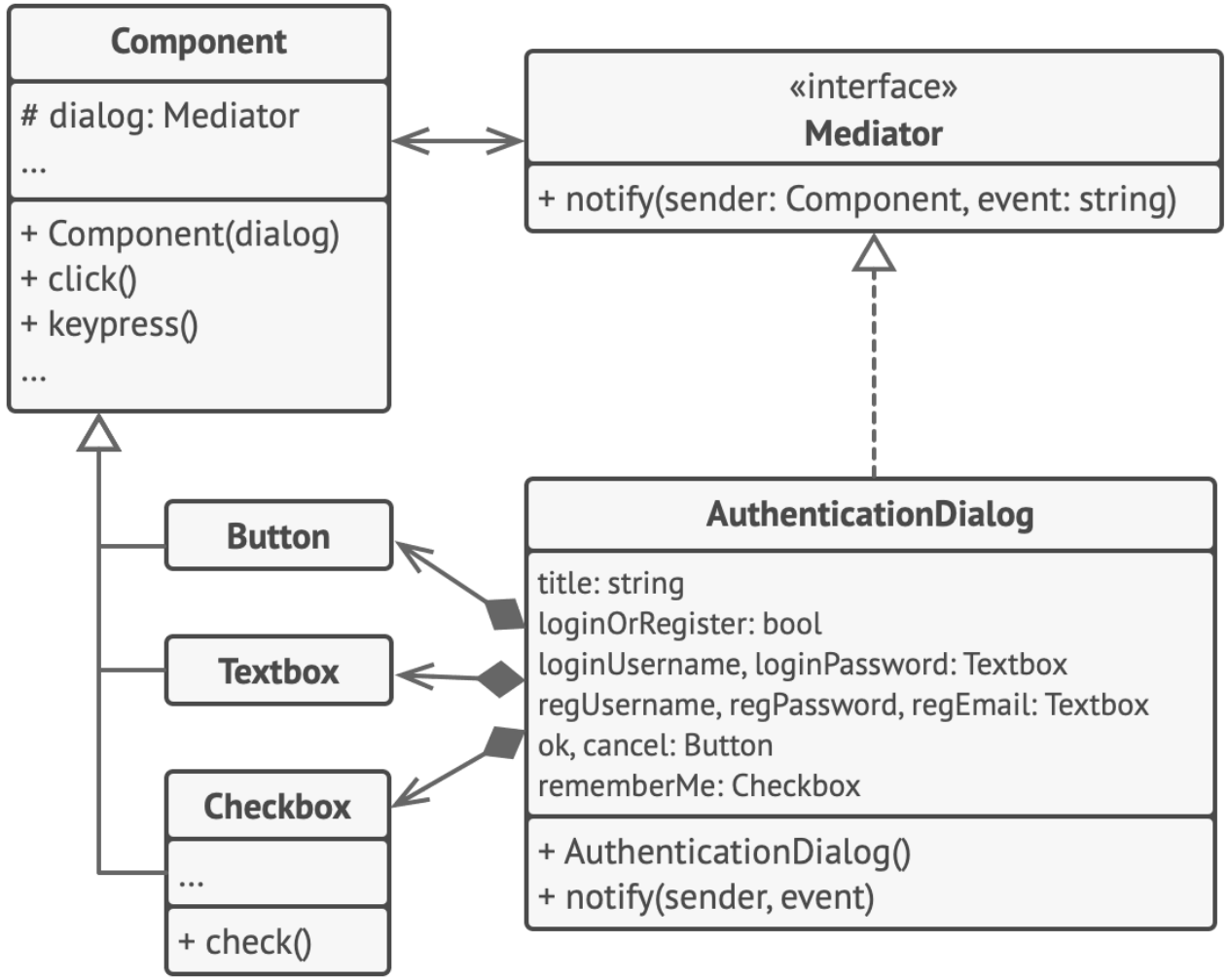


1. **Components:** business logic
2. **Mediator:** interface, declaring methods of communication with components
   - Usually, just a single notification method
   - Components may pass context as arguments
3. **Concrete Mediators:** encapsulating relations between components
   - Keep references to all components, and even manage their lifecycles

- Components not aware of others
- A black box from a component's perspective

[Figures in this slide are extracted from https://refactoring.guru/design-patterns/mediator]

Mediator: Example



- The authentication dialog acts as the mediator
- Upon receiving a notification about an event, it decides what element should address the event

[Figures in this slide are extracted from https://refactoring.guru/design-patterns/mediator]

Mediator: Applicability

- When it is hard to change some classes because they are tightly coupled to others
  - Extract all relationships between classes into a separate class, isolating changes to a specific component from the rest
- When you cannot reuse a component because it is too dependent on others
  - Individual components become unaware of the others, and communicate indirectly
- When you create tons of component subclasses just to reuse some basic behavior in various contexts
  - Define entirely new ways for components to collaborate by introducing new mediator classes, without changing the components

Mediator: Implementation

1. Identify a group of tightly coupled classes which would benefit from being more independent
2. Declare the **mediator interface** and describe the desired **communication protocol** between mediators and various components
3. Implement the **concrete mediator class**, with references to all components
4. Optional: make the mediator responsible for the creation and destruction of component objects (like a factory or facade)
5. Components store a reference to the mediator
6. Change the components' code so that they call the mediator's method instead of methods on other components
   - Extract the code that involves calling other components into the mediator class

Mediator: Pros and Cons

- **Pros**
  - Single Responsibility Principle: extracting communications into a single place
  - Open/Closed Principle: introducing new mediators without changing components
  - Reduce coupling among components
  - Reuse individual component more easily
- **Cons**
  - A mediator may evolve into a God Object

Combinations and Comparisons

- **Iterators** can be used to traverse **Composite** trees
- **Factory Method** can be used along with **Iterator**

- **Chain of Responsibility, Command, and Mediator**
  - Chain of Responsibility: passes a request sequentially along a dynamic chain
  - Command: establishes unidirectional connections between senders and receivers
  - Mediator: eliminates direct connections between senders and receivers
- **Facade and Mediator**
  - Façade: a simplified interface to a subsystem of objects, without new functionality
    - The subsystem is unaware of the façade
    - Objects within the subsystem communicate directly
  - Mediator: centralizes communication between components
    - Components only know about the mediator, and communicate indirectly