Behavioral Patterns

Chain of Responsibility

Mediator

Decouple sender and handler

- Observer
- Command
- mediator
- Chain of Responsibility

Chain of Responsibility

- When you want to give more than one object a chance to handle a request

Example scenario: Email sorter

- There are 3 types of incoming emails: spam, complaint, request.
- Each will be put into a proper directory.
  - Assume there is a detector that can tell the email category.
- Create a design that can use the detectors to sort the incoming emails

Attempt 1:

```
public class Inbox{
 public void receivedEmail(Email
e){
    if (detector.isSpam(e)){
          //code to move to spam
folder
    }else if(detector.isRequest(e){
         // move to request folder
    }else if (detector.isComplaint(e
){
         //move to complaint folder
    } else {}
      //do nothing leave at inbox
 }
…
}
```

- Ugly
- Problems?
  - OCP
  - Lack of cohesion
  - Single responsibility

Attempt 2: Single responsibility

```
public class Inbox{
 private SpamFilter sf;
 private RequestFilter rf;
 private ComplaintFilter cf;
….
 public void receivedEmail(Email
e){
     if (detector.isSpam(e)){
          sf.sort(e);
     }else if(detector.isRequest(e){
           rf.sort(e );
     }else if (detector.isComplaint(e
){
           cf.sort (e );
   } else { }//do nothing leave at
inbox
```

```
public class SpamFilter{
  public void sort(Email e){
     if (detector.isSpam()){
     //code to save it to spam folder
     }
}}
public class RequestFilter{
  public void sort(Email e){
    if (detector.isRequest()){
     //code to save it to request
folder
    }
}}
public class ComplaintFilter{…}
```

Problems?

Attempt 2: Single responsibility

```
public class Inbox{
 private SpamFilter sf;
 private RequestFilter rf;
 private ComplaintFilter cf;
....
 public void receivedEmail(Email
e){
     if (detector.isSpam(e)){
          sf.sort(e);
     }else if(detector.isRequest(e){
           rf.sort(e );
     }else if (detector.isComplaint(e
){
           cf.sort (e );
   } else { }//do nothing leave at
inbox
```

Problems:

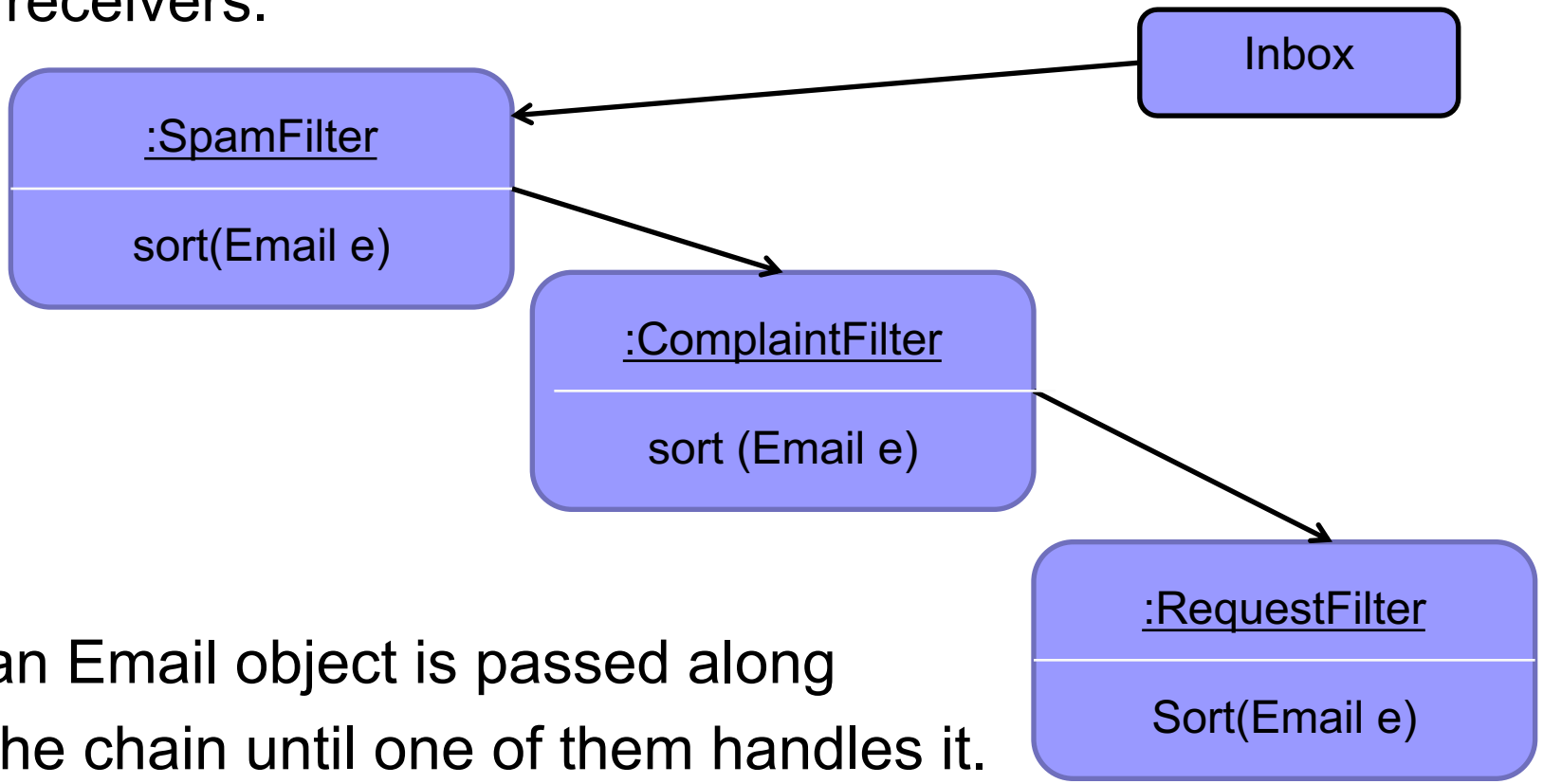
- High coupling
- OCP?
  - Extend with new filters
  - Change the filtering order

Example: Email sorter

- We want to give the Spam, Complaint, and Request filters a chance to save an email into an appropriate directory
  - More than 1 object can handle an email
  - Main application should not be coupled with these handlers as you may introduce new emailing rules
- Solution: Create a chain of filters, each examines the email and then handles or passes to the next filter

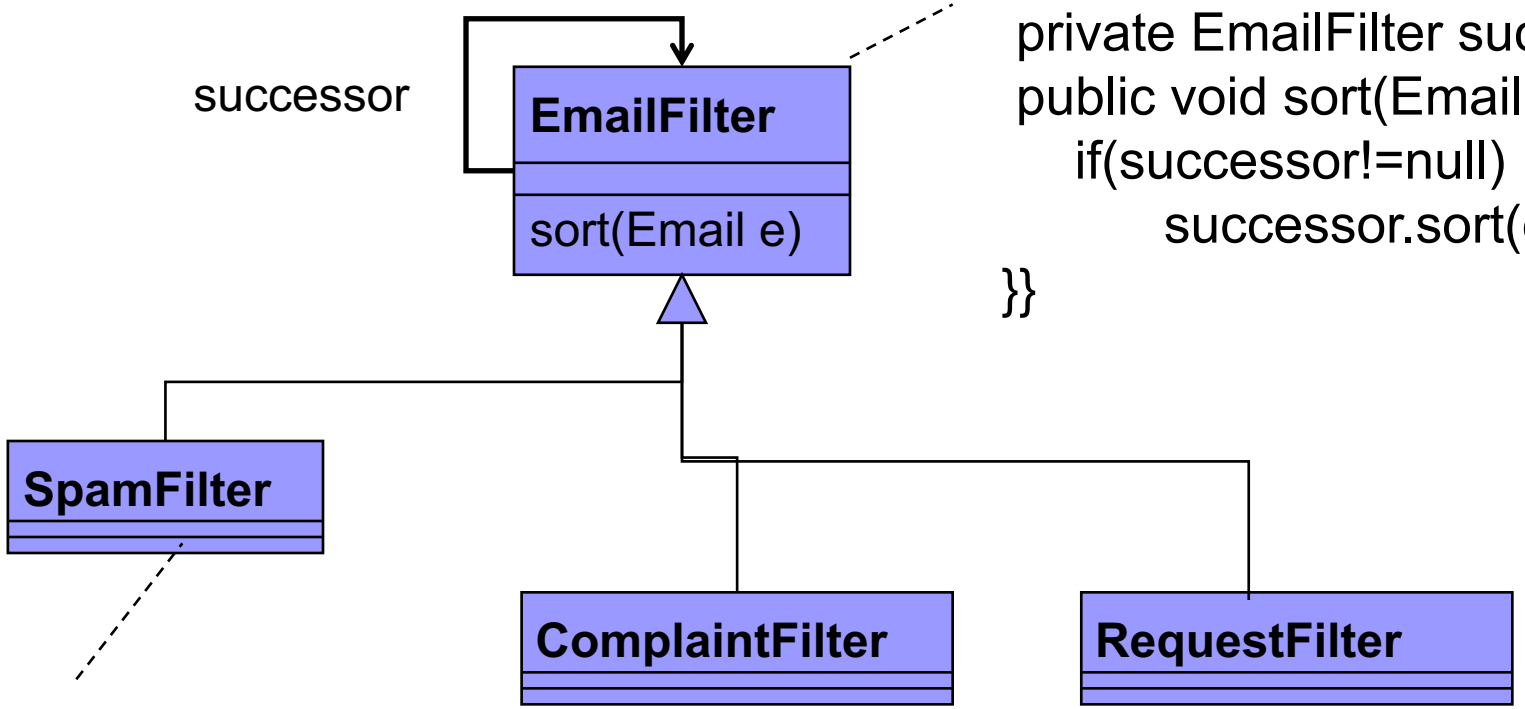
Chain of Filters (handlers)

Use a chain of objects to decouple the senders from the receivers.



an Email object is passed along the chain until one of them handles it.

Email handling chain




```
public class EmailFilter{
   private EmailFilter successor;
   public void sort(Email email){
      if(successor!=null)
           successor.sort(email);
}}
```

```
If (email is spam){
    //put it into spam directory
} else{
   super.sort(Email email);
}
```

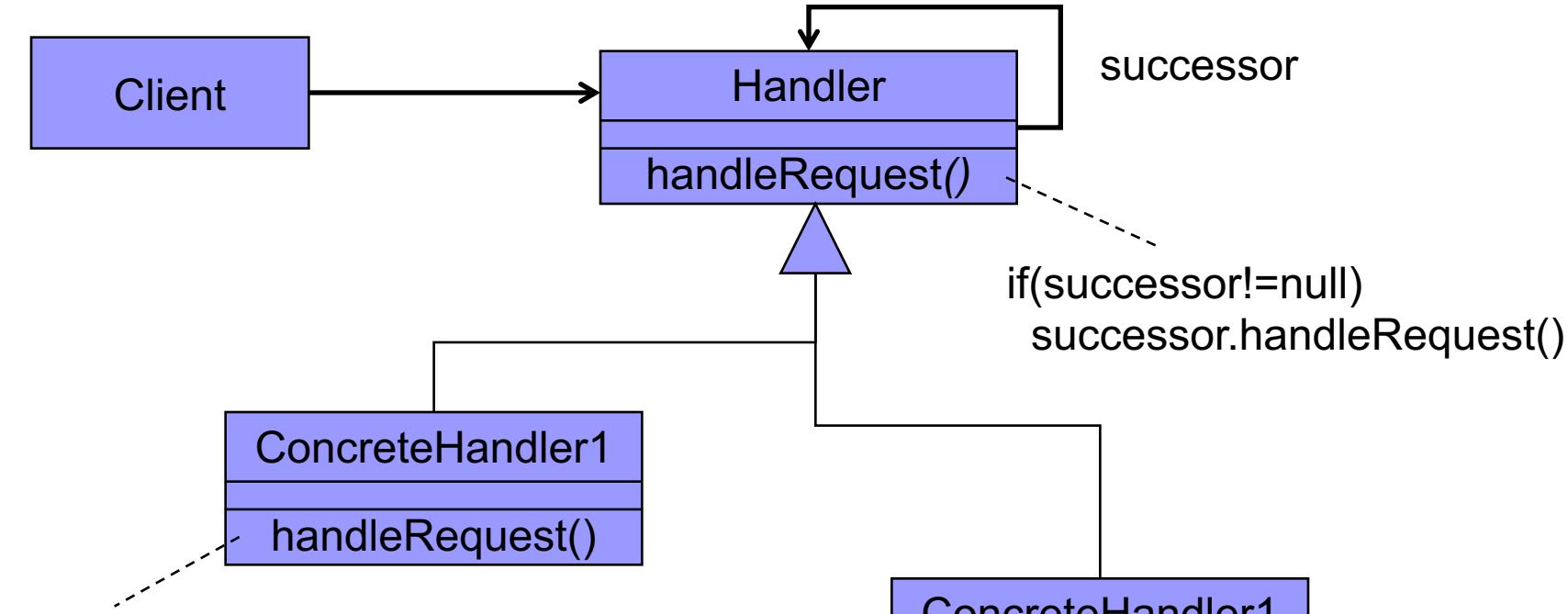
Chain of Responsibility

- **Intent**
  - Avoid coupling requester and provider by giving more than one object chance to handle
- **Applicability**
  - Multiple objects, determined at runtime, are candidates to handle a request
    - Who will handle it is not known beforehand
    - you want to issue a request to one of several objects without specifying the receiver explicitly

Chain of Responsibility

- **Applicability**
  - Multiple objects, determined at runtime, are candidates to handle a request
    - Who will handle it is not known beforehand
    - want to issue a request to one of several objects without specifying the receiver explicitly
  - Execute several handlers in a particular order.
  - and their order are supposed to change at runtime.
  - Don't want to coupled with each handlers explicitly in your code

Structure – Chain of Responsibility




```
if (can handle){
    //handle the request
} else{
   super.handleRequest();
}
```

Attempt 3: Email Sorter with CoR

```
public class Inbox{
 private EmailFilter chain;

 public void receivedEmail(Email
e){
     chain.sort(e);
     if (detector.isSpam(e)){
          sf.sort(e);
     }else if(detector.isRequest(e){
          rf.sort(e );
     }else if (detector.isComplaint(e
){
          cf.sort (e );
   } else { }//do nothing leave at
inbox
 }
```

```
public class SpamFilter{
  public void sort(Email e){
     if (detector.isSpam()){
     //code to save it to spam folder
     } else super.sort(e );
}}
public class RequestFilter{
  public void sort(Email e){
     if (detector.isRequest()){
     //code to save it to request
folder
    }else super.sort(e );
}}
public class ComplaintFilter{…}
```

Email Sorter with CoR

```
//main method or some init
EmailFilter spam = new SpamFilter();
EmailFilter request = new RequestFilter();
EmailFilter complaint = new ComplaintFilter();
// Build the chain by setting successors,
//order defines the priority.
spam.setSuccessor(request);
request.setSuccessor(complaint);
//give it to CoR client
Inbox inbox=new Inbox(spam);
// triggering the chain
inbox.receivedEmail(someEmail);
```

```
public class Inbox{
   private EmailFilter chain;
   public Inbox(EmailFilter f){
       chain=f;
   }
   public void
       receivedEmail (Email e){
     chain.sort(e);
   }
}
```

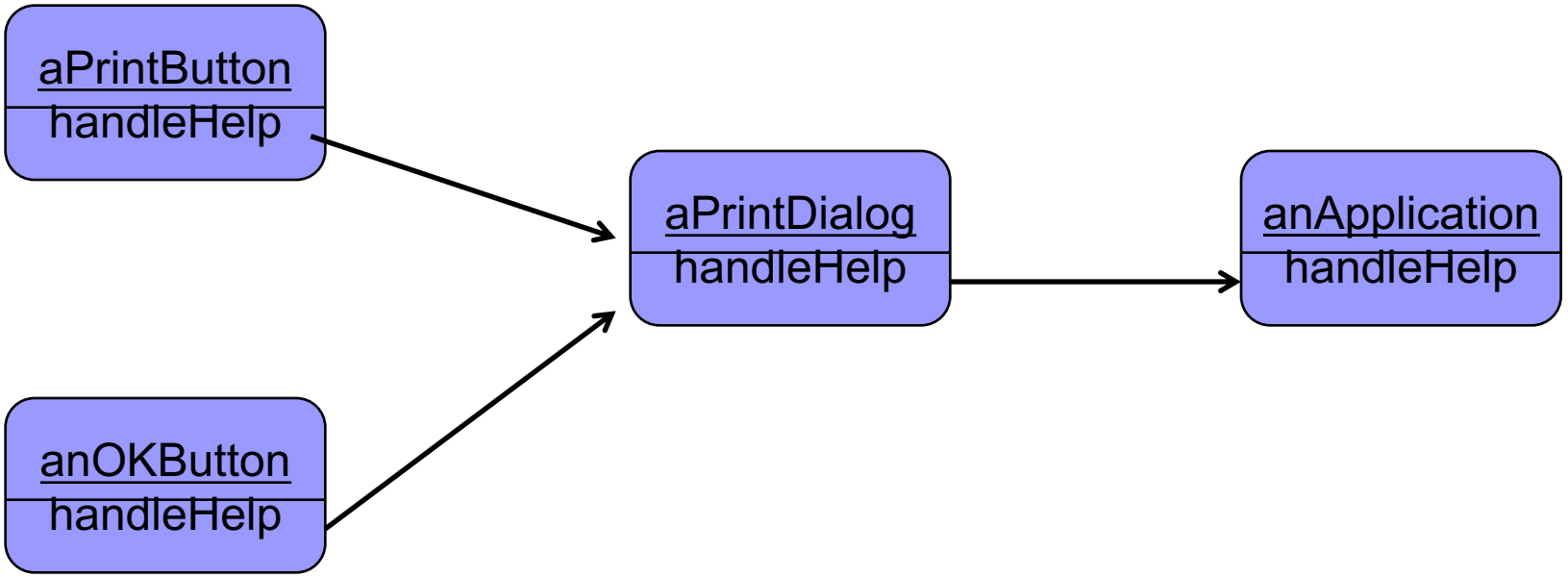
Benefits and Drawbacks

- Benefits
  - Simplifies objects because they don't have to know the chain's structure
    - Inbox does not keep direct references to members of the chain
    - Each member is unaware of the others
  - Allows to add or remove responsibilities dynamically by changing members
  - Allows to change the order of responsibilities at runtime
- Drawbacks
  - execution of the request is not guaranteed. It may fall off the end of chain
  - can be hard to observe the runtime characteristics and debug

Example II

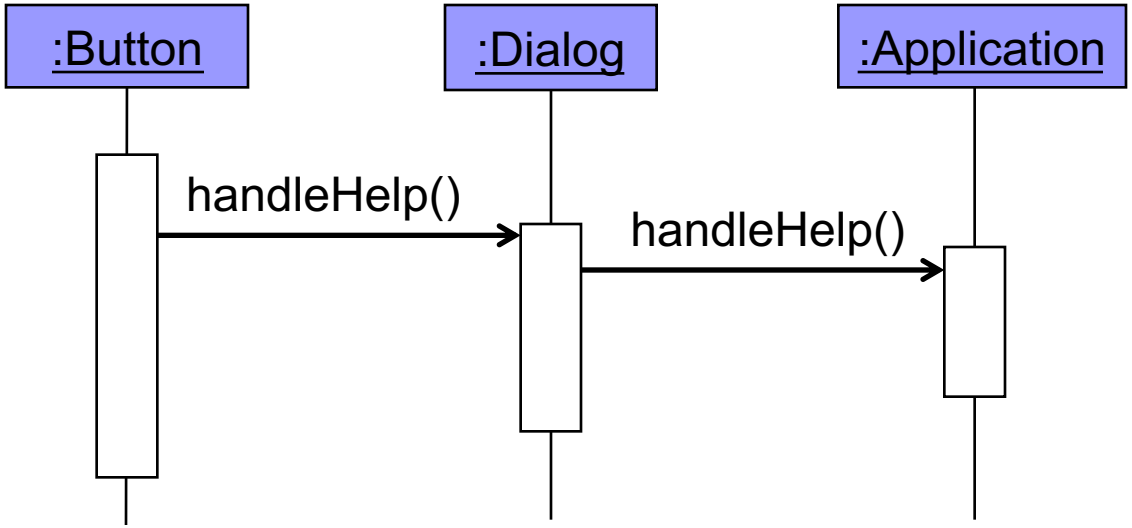
- Context sensitive help facility
  - Help on any part of the user interface
  - Help information depends on the selected parts and its context
    - Button in main window have different help information than a button in a dialog box
  - If no help information available for that part, display the help for the immediate context
- The object that will provide Help is not known explicitly to the requester object (e.g. the button)

A sample object diagram



Example II

- Decouple requester and provider via **Chain of responsibility**



- Request will be passes along a chain
  - 1st object receives the help information request.
  - Either handles it or pass it to the 2nd object
  - Follow the chain until one object handles it

Abstract Handler

```
typedef int Topic;
const Topic NO_HELP_TOPIC = -1;

class HelpHandler {
public:
    HelpHandler(HelpHandler* = 0, Topic = NO_HELP_TOPIC);
    virtual bool HasHelp();
    virtual void SetHandler(HelpHandler*, Topic);
    virtual void HandleHelp();
private:
    HelpHandler* _successor;
    Topic _topic;
};
```

- Defines interface for handling help request.
- Initially help topic is empty (-1).

- Subclasses override HandleHelp().
- HasHelp is a convenience operation for checking whether there is an associated help topic.

```
HelpHandler::HelpHandler (
    HelpHandler* h, Topic t
) : _successor(h), _topic(t) { }

bool HelpHandler::HasHelp () {
    return _topic != NO_HELP_TOPIC;
}

void HelpHandler::HandleHelp () {
    if (_successor != 0) {
        _successor->HandleHelp();
    }
}
```

Handler Widget

```
class Widget : public HelpHandler {
 protected:   Widget(Widget* parent, Topic t = NO_HELP_TOPIC);
 private:   Widget* _parent;
 };
 Widget::Widget (Widget* w, Topic t) : HelpHandler(w, t) { _parent =
w
```

```
class Button : public Widget {
public:
    Button(Widget* d, Topic t = NO_HELP_TOPIC);

    virtual void HandleHelp();
    // Widget operations that Button overrides...
};
Button::Button (Widget* h, Topic t) : Widget(h, t) { }
void Button::HandleHelp () {
    if (HasHelp()) {
        // offer help on the button
    } else {
        HelpHandler::HandleHelp();
    }
}
```

Concrete Handlers

```
void Dialog::HandleHelp () {
    if (HasHelp()) {
        // offer help on the dialog
    } else {
        HelpHandler::HandleHelp();
    }
}
Dialog::Dialog (HelpHandler* h,  Topic t) : Widget(0) {
    SetHandler(h, t);
}
```

Dialog will be the root. Its next handler will be the Application

```
class Application : public HelpHandler {
public:
    Application(Topic t) : HelpHandler(0, t) { }

    virtual void HandleHelp();
    // application-specific operations...
};

void Application::HandleHelp () {
    // show a list of help topics
}
```

Application is not a Widget, but a HelpHandler. When a help request propagates to this level, the application can supply information on the application in general, or it can offer a list of different help topics:

Setting up

```
const Topic PRINT_TOPIC = 1;
const Topic PAPER_ORIENTATION_TOPIC = 2;
const Topic APPLICATION_TOPIC = 3;

Application* application = new Application(APPLICATION_TOPIC);
Dialog* dialog = new Dialog(application, PRINT_TOPIC);
Button* button = new Button(dialog, PAPER_ORIENTATION_TOPIC);
….
button->HandleHelp();
```

Implementation issues

- Request structure
  - When request is data
    - Handlers with the same method name to work with it
    - Handlers with a set of method names, client tells which one and sends the request data
  - When request is an action
    - Encoding with string, enum, or int – not safe
    - Command pattern (next)
- Variation: pipe and filter
- Using existing links or defining new links for successor (next)
- Broken chain (next next)

CoR variations: Strict CoR

- Strict CoR : First-Handler Model
- Rule: Exactly **one** handler in the chain processes the request.
- How it works: A handler checks if it can process the request.
  - If YES, it handles it and **stops** the chain.
  - If NO, it passes the request to its successor.
- E.g. Context Sensitive Help
  - It only calls HelpHandler::HandleHelp() in the else block .

CoR variations: Pipe-and-Filter

- Rule: All (or many) handlers in the chain get a chance to process the request.
- How it works: A handler performs its action (e.g., logging) and then unconditionally passes the request to the next handler to continue processing.
- E.g. javax.servlet.Filter. A request (like AuthenticationFilter -> LoggingFilter) must pass through all of them.
  - conceptually very similar to Decorator
- we will use Strict CoR in this course.

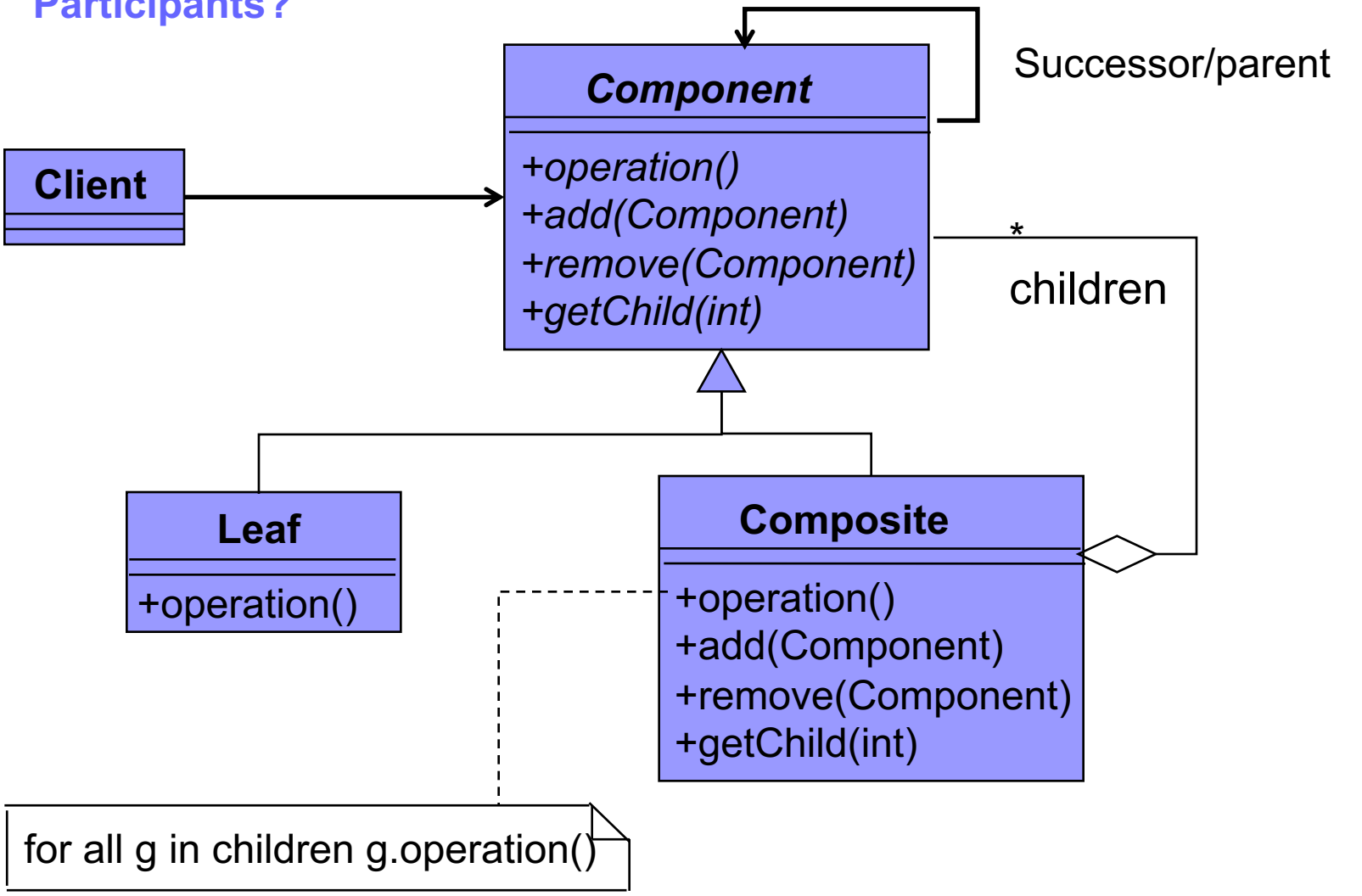
CoR and Composite

- Using existing links of Composite
- Chain of Responsibility is often used in conjunction with Composite.

- When a leaf component gets a request, it may pass it through the chain following the parent links of the components up to the root of the object tree.
  - E.g. in a GUI leaf component (a button) passes the help request to its container (parent). It is passed up until someone handles it.

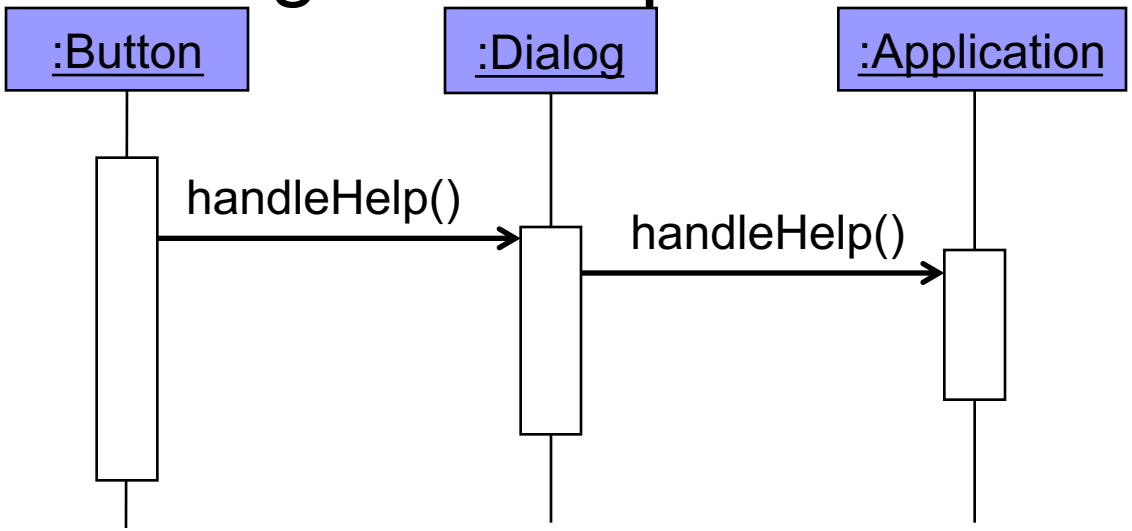
Composite with CoR

Participants?



CoR + Composite Example

- Recall the context sensitive help
- Make Widgets composite



- Request will be passes along a chain
  - 1st object receives the help information request.
  - Either handles it or pass it to the 2nd object
  - Follow the chain until one object handles it

CoR with Composite

```
class Widget : public HelpHandler, Composite {
  protected:        Widget(Widget* parent, Topic t = NO_HELP_TOPIC);
 };
 Widget::Widget (Widget* w, Topic t) : HelpHandler(w, t) { _parent =
w;}

class Composite{
 protected: std::vector<Component *> children;
                    Component *_parent;
public:
 virtual void add(Component *c) {children.push_back(c); c-
>parent=this};
 virtual void remove(Component *c){children.remove(c)};
 virtual Component *get(int i) const {
      if (i<children.size()) return children[i]; return 0;}
```

Broken chain

- What if programmer of a Handler does not pass control to the successor?
  - When the handler cannot process the request, the request must follow the chain
  - May omit to call super.handle(request)
  - Abstract Handler does not implement the default behavior and handler omits successor.handle(request)
- What is invariant of handling a request?
  - Do it
  - If not, pass it on
- Implement the invariant once. Which pattern?

Broken chain – solution

- Template Method to rescue.

Participants?

```
abstract class Handler{
   public final void handleReq(Request r){
        bool isHandled=handleRequest(r);
        if(successor!=null && !isHandled)
              successor.handleReq(r);
   }
   public abstract boolean handleRequest(Request r);
   private Handler successor=null;
   public void setSuccessor(Handler h){
        this.successor=h;}
}
```

Consequences -1

- Reduced coupling
  - simplifies objects because they don't have to know the chain's structure and do not keep direct references to its members
- Flexibility in assigning responsibilities
  - allows to add or remove responsibilities dynamically by changing members or the chain order
- OCP and Single Responsibility
  - Handlers can be added/removed without altering existing code (OCP), and each handler focuses only on its processing logic (SRP).

Consequences -2

- Some requests might not get handled
  - execution of the request is not guaranteed.
  - if may fall off the end of chain
- Can be hard to observe the runtime characteristics and debug

Known use

- java.awt used to be CoR but too many requests passed up in the hierarchy, so they switched to Observer pattern

Related patterns

- Observer, Mediator and CoR
  - *Chain of Responsibility* passes a request sequentially along a dynamic chain of potential receivers until one of them handles it.
  - *Mediator* eliminates direct connections between senders and receivers, forcing them to communicate indirectly via a mediator object.
  - *Observer* lets receivers dynamically subscribe to and unsubscribe from receiving requests.
- Read https://gameprogrammingpatterns.com/observer.html#it-does-too-much-dynamic-allocation
- Do you see the chain of responsibility?

Related patterns

- Composite together with CoR
  - Parent links serve as successor
  - The structural parent/child links within the Composite tree are *reused* to serve as the successor links in the Chain
  - Difference: Composite is about **part-whole hierarchy** for recursive operations. CoR is about **delegation for singular handling**.
- Command to implement requests as objects
- Decorator vs CoR
  - **add responsibilities** vs **find a single handler** that stops the chain

Decorator vs CoR

- Decorator object diagram is almost the same as Chain of Responsibility object diagram
  - Chain of handlers vs linked list of decorators

  However:
  - Decorator has the real subject at the end,
  - CoR all handlers are at the same level, no big object at the end

Decorator vs CoR -difference

- With the decorator, all classes handle the request;
  - Every decorator in the chain is executed
  - **Pipe-and-Filter is conceptually similar to Decorator**

- While for the chain of responsibility, **exactly one** of the classes in the chain handles the request (Strict CoR).
  - This is a strict definition of the Responsibility concept in the GoF book. We will use strict CoR.
  - One handler stops passing the request to the next one.

Related Patterns –Request Handling

Pattern	How it Decouples (The Key Difference)	CoR vs. Alternative
Observer	**Broadcasting/1-to-N:** Receivers dynamically subscribe and unsubscribe. All subscribed receivers get the request/notification.	CoR passes the request sequentially for **exclusive handling** (Strict CoR)/ Observer broadcasts for **shared notification. (**request volume).
Command	**Request Encapsulation:** Encapsulates the request as an object, which is then passed to a chain or queue.	CoR is the dispatcher; Command is the payload. Command is often used *with* CoR to implement requests as objects:safer than encoding requests as strings.
Mediator	**Centralized Hub:** Eliminates direct connections between senders and receivers.	CoR is **distributed and self-chaining**. Mediator is **centralized and explicit**.

CoR with Command

- Command is the “request”

```
// A concrete command
public class AuthCommand implements Command {
    private User user;
    public AuthCommand(User u) { this.user = u;    }
    public void execute() {
        System.out.println("Authenticating " + user.name);   // ... auth logic
    }
}
public class LogCommand implements Command {
    private String message;
    public LogCommand(String msg) { this.message = msg;   }
    public void execute() {
        System.out.println("LOG: " + message); // ... logging logic ...
    }
}
```

CoR with Command

- Handler executes the command

```
// A concrete command
public class AuthCommand
     implements Command {
 private User user;
 public AuthCommand(User u) {
       this.user = u;    }
 public void execute() {
      // ... auth logic
    }
}
```

```
public class AuthHandler extends Handler


 protected boolean
    canHandle  (Command cmd) {
       // only handles AuthCommands
      if (cmd instanceof AuthCommand) {
            cmd.execute();
            return true; // Stop the chain
       }
      return false; // Pass to successor
    }
}
```

```
public class Application {
    public static void main(String[] args) {
        // 1. Build the chain
        Handler auth = new AuthHandler();
        Handler log = new LogHandler();
        auth.setSuccessor(log);
        // auth -> log -> null

        // 2. Create command objects
        Command login = new AuthCommand(new User("grad_student"));
        Command greeting = new LogCommand("User logged in");
        Command invalid = new PaymentCommand(200.0); // Not shown
```

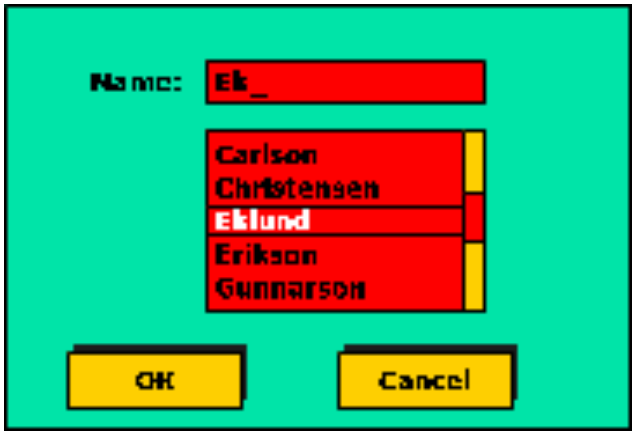
```
    // 3. Send commands to the chain root
        System.out.println("--- Sending Auth Command ---");
        auth.handle(login);
        // Output: "Authenticating grad_student"

        System.out.println("--- Sending Log Command ---");
        auth.handle(greeting);
        // Output: "LOG: User logged in"

        System.out.println("--- Sending Invalid Command ---");
        auth.handle(invalid);
        // Output: (nothing)
    }
}
```

Decouple sender and handler

- Observer
- Command
- mediator
- Chain of Responsibility

Example: GUI coordination

- A From dialog window



- Multiple dependencies
  - When an item of list is selected,
    - the selection will be displayed on the entry field
    - Ok button will be enabled, etc…
  - Change in the FilterPanel triggers updates in
    - ProductList, Pagination, and SummaryLabel

Motivation

- It is hard to keep track which rules reside in which objects, and how objects should relate to each other
- Chaotic many-to-many dependency

- If the objects interact with each other *directly*, the system components are *tightly* coupled with each other making higher maintainability cost and not hard to extend

Could CoR solve it?

- coordinate complex updates between a FilterPanel, SearchBox, and ProductList.

- GUI Needs Simultaneous Coordination
  - A change in the FilterPanel must trigger updates in the ProductList, Pagination, and SummaryLabel **at the same time.**

Could CoR solve it?

- CoR is for Sequential Delegation
  - If the filter panel became the head of the chain, the request would hit the SearchBox first.
  - The SearchBox would have no reason to approve or stop the request, so it would pass it down, adding unnecessary overhead.
- CoR fails: problem is not about giving one object a chance to handle the request;
- It is about managing a complex fan-out of required actions triggered by a single event.

Observer?

- Observer eliminates the direct link between a **Subject** and its observers, but in complex GUIs …
- 1) Chaotic Many-to-Many Dependencies
  - Everyone subscribes to everyone
  - a tangled web of subscriptions.
- 2) Notification Cascade
  - If the FilterPanel (Subject) broadcasts a change, the ProductList, SummaryLabel, and Pagination all receive it.
  - If the ProductList then changes the page count, it has to notify *other* observers, which can trigger **circular** or **cascading** updates.
    - **circular updates:** A notifies B, B's update notifies A, etc..

Observer?

3) Low cohesion in Observers

```
public void update(Subject changedSubject) {
    // I have to figure out WHO notified me...
    if (changedSubject == filterPanel) {
        // ...and then run the specific logic
        runFilterLogic();
    } else if (changedSubject == searchBox) {
        runSearchLogic();
    } else if (changedSubject == pagination) {
        runPaginationLogic();
    }
    // This is complex, brittle, and violates SRP!
}
```

Observer?

- Notification vs. Coordination
- Observer: 1-to-N broadcasting.
  - "Does anyone need to *know* I changed?“
- GUI Problem: COORDINATION
  - N-to-N interaction.
  - "When this component changes, *what complex logic* must I execute to orchestrate all other components?“
- Observer fails because it *distributes* the coordination logic, leading to chaos. We need a pattern that *centralizes* it

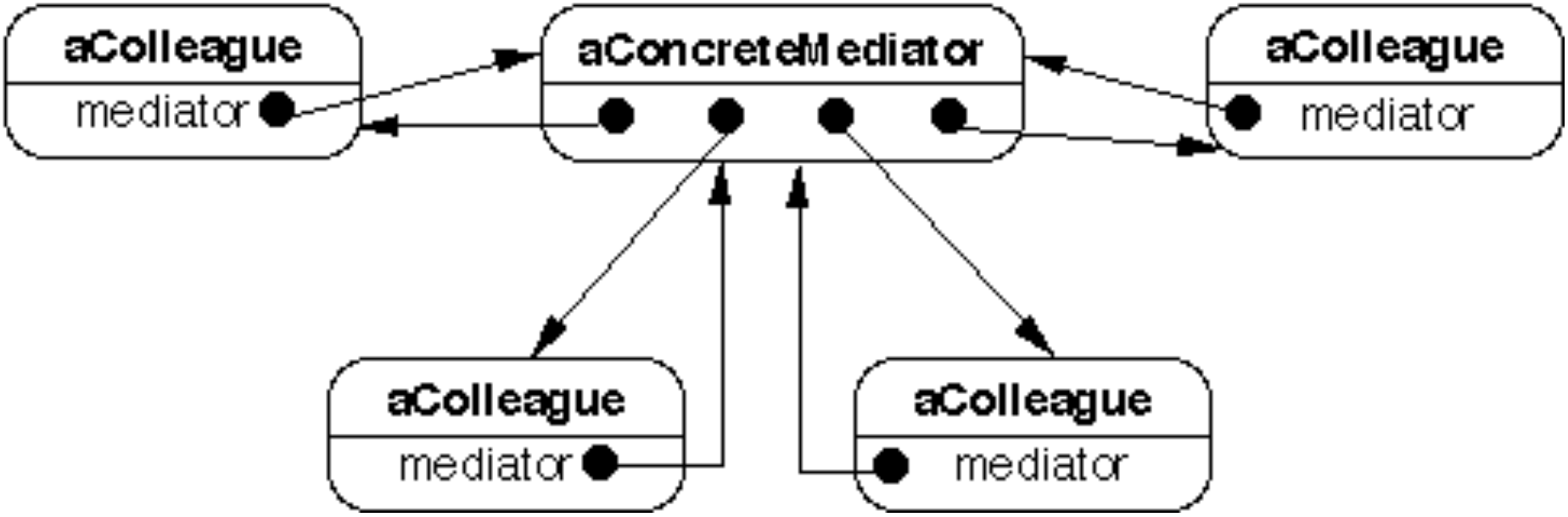
Motivation

- Chaotic many-to-many dependency

Solution: Mediate the communication

- Centralize complex communication and control between related objects
  - colleagues tell the mediator when their state changes
  - they respond to requests from the mediator

Mediator Structure



Mediator Pattern

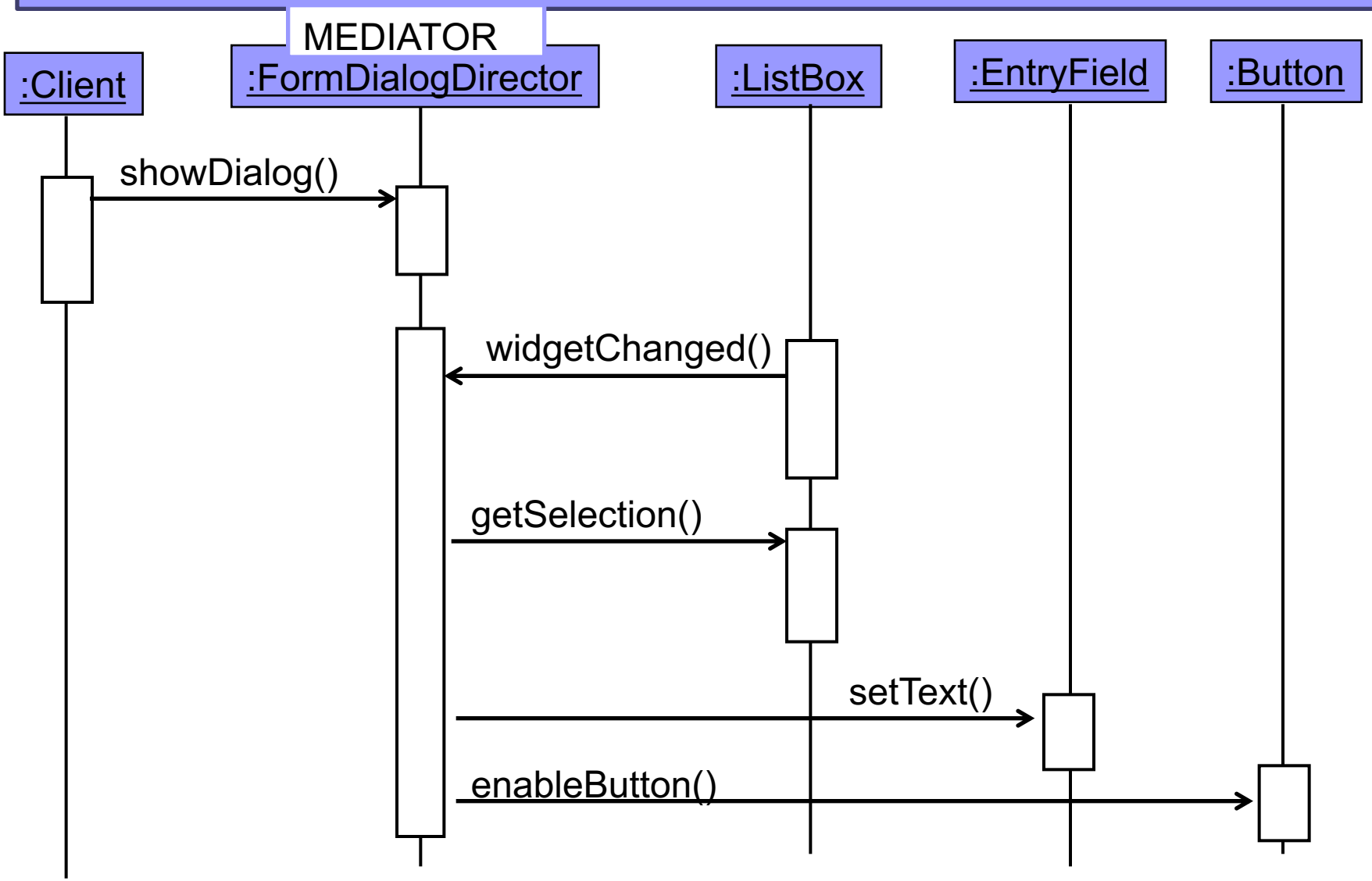
- A Form dialog window

- A mediator: FormDialogDirector
  - control and coordinate the interactions
  - objects only know the Mediator
    - reduced number of interactions

- *Commonly used to coordinate related GUI components*

See https://refactoring.guru/design-patterns/mediator/java/example

Sequence Diagram



Mediator Pattern

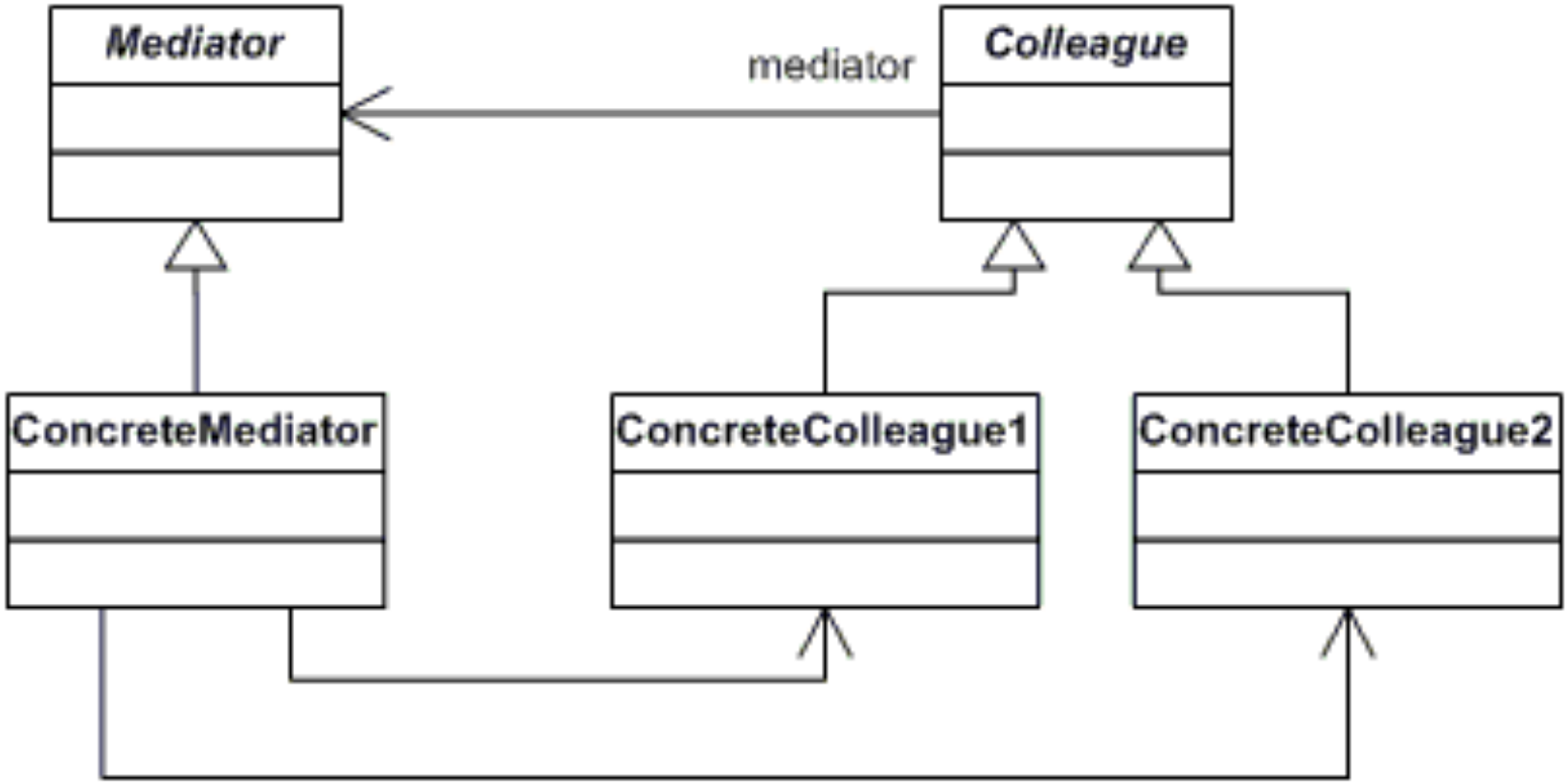
- A mediator is responsible for controlling and coordinating the interactions of a group of objects
- Putting the logic in one mediator object to manage state changes of others
  - instead of distributing the logic over the other objects
  - *trading complexity of interaction for complexity of mediator*
- More cohesive implementation of the logic
- Decreased coupling between the other objects

Mediator Pattern

- **Intent:** Define an object that encapsulates how a set of objects interact.

  Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently

Mediator Structure



As long as the colleague works with its mediator via the generic interface, we can link it with a different implementation of the mediator.

Participants

- **Mediator interface**: defines the interface between the objects.
  - In most cases, a single method for receiving notifications from components is sufficient
- **ConcreteMediator**: stores references to colleagues. Implements the communication protocol and coordinates colleagues
- **Colleague:** has reference to the mediator. Communicates with its mediator whenever it would have otherwise communicated with another colleague.
  - When the colleague works with its mediator via the generic interface, we can link it with a different implementation of the mediator.

Centralized Control

- Benefits
  - colleagues focus on their operations
  - interaction protocol becomes visible
  - when the rule changes only change the mediator
  - when a new colleague is added, change only the mediator
  - participants are not aware of the existence of other objects
- Drawback
  - overkill for simple communication

Applicability –when…

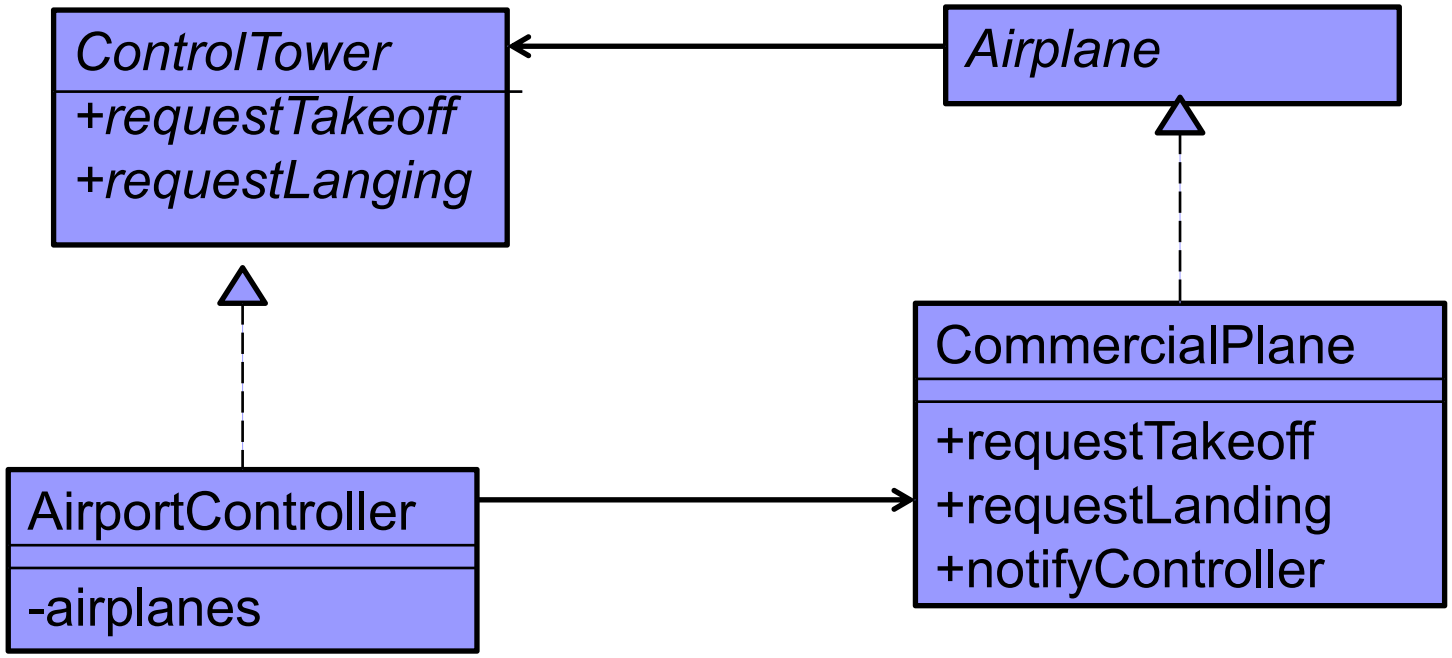
- A set of objects communicate in a well-defined but complex ways. The interdependencies are unstructured and difficult to understand.
- It is hard to change some of the classes due to tight coupling with other classes
  - Also, when some objects are hard to reuse because it refers to and communicates with many other objects
- A behavior that's distributed between several classes should be customizable without a lot of subclassing
  - Each participant’s subclass vs Mediator subclass

Example 1: Air Traffic

- There are multiple airplanes that need to communicate and coordinate their movements to avoid collisions and ensure safe takeoffs and landings.
  - Direct communication between airplanes is error-prone
  - Collision risk when the status of all airplanes in the air space is unknown

Source: https://www.geeksforgeeks.org/mediator-design-pattern/

Example: Air Traffic



Implementing Colleagues

```
// Colleague Interface
public interface Airplane {
    void requestTakeoff();
    void requestLanding();
    void notifyAirTrafficControl(String message);
}
// Concrete Colleague
public class CommercialAirplane implements Airplane {
    private AirTrafficControlTower mediator;
    public CommercialAirplane(AirTrafficControlTower mediator) {
        this.mediator = mediator;}
    public void requestTakeoff() {
        mediator.requestTakeoff(this);}
    public void requestLanding() {
        mediator.requestLanding(this); }
    public void notifyAirTrafficControl(String message) {
        System.out.println("Commercial Airplane: " + message);}
```

Implementing Mediator

```
public interface ControlTower { // Mediator Interface
        void requestTakeoff(Airplane airplane);
        void requestLanding(Airplane airplane);
}
// Concrete Mediator
public class AirportController implements ControlTower {
    public void requestTakeoff(Airplane airplane) {
        // Logic for coordinating takeoff
        airplane.notifyAirTrafficControl("Requesting takeoff
clearance.");
    }
   public void requestLanding(Airplane airplane) {
        // Logic for coordinating landing
        airplane.notifyAirTrafficControl("Requesting landing
clearance.")  }
```

Implementation issues-1

Colleague-Mediator communication

- Choice1: Mediator observing colleagues (subjects)
  - Each time a change is made in the state of a colleague object, the mediator gets notified
  - Then, mediator notifies all other colleague objects.
- Choice 2: When communicating with the mediator, a colleague passes itself as an argument, allowing the mediator to identify the sender.
  - Airplane example
- Message queue in mediator

Implementation issues -2

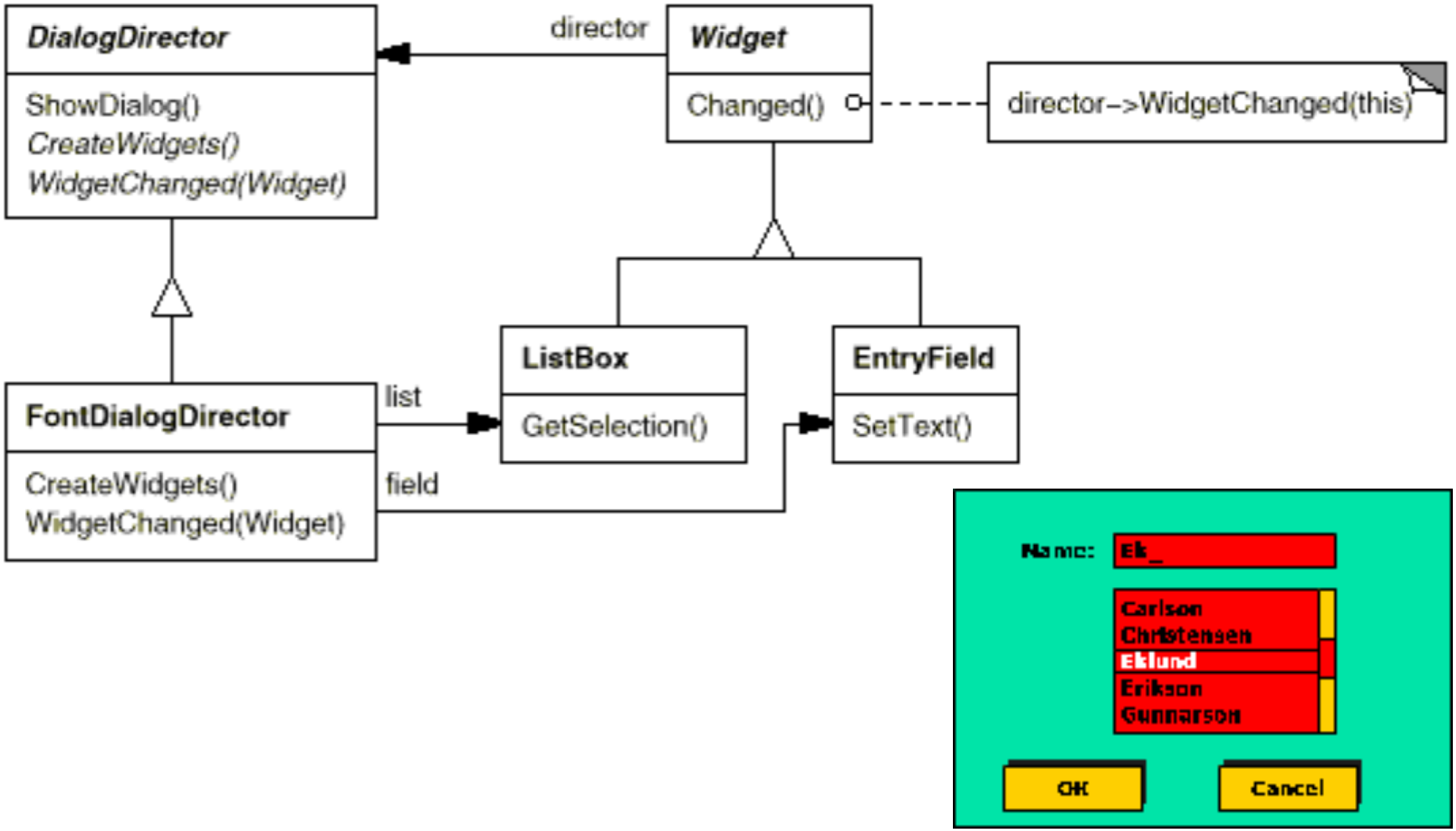
- If colleagues are going to use only one mediator, no need to define Abstract Mediator.
  - The definition of an abstract Mediator is required only if the colleagues needs to work with different mediators.
- Complexity of the mediator could be a problem when there are a lot of participants
- When not all colleagues are of the same abstract type
  - Mediator becomes tightly coupled
  - Mediator logic becomes complex
  - Adapter?

Back to Example 1: DialogDirector

When an item in the list box is selected

1. The list box tells its director that it's changed.
2. The director gets the selection from the list box.
3. The director passes the selection to the entry field.
4. Now that the entry field contains some text, the director enables button(s) for initiating an action

Example 1: DialogDirector



Example1: Colleagues

```
class Widget {
public:
    Widget(DialogDirector*);
    virtual void Changed();

    virtual void HandleMouse(MouseEvent& event);
    // ...
private:
    DialogDirector* _director;
};

void Widget::Changed () {
    _director->WidgetChanged(this);
}
```

Concrete Colleagues

```
class Button : public Widget {
public:
    Button(DialogDirector*);

    virtual void SetText(const char* text);
    virtual void HandleMouse(MouseEvent& event);
    // ...
};

void Button::HandleMouse (MouseEvent& event) {
    // ...
    Changed();
}
```

```
class ListBox : public Widget {
public:
    ListBox(DialogDirector*);

    virtual const char* GetSelection();
    virtual void SetList(List<char*>* listItems);
    virtual void HandleMouse(MouseEvent& event);
    // ...
};
```

Example1: Mediator

```
class DialogDirector {
public:
    virtual ~DialogDirector();

    virtual void ShowDialog();
    virtual void WidgetChanged(Widget*) = 0;

protected:
    DialogDirector();
    virtual void CreateWidgets() = 0;
};
```

- Do I see a Factory Method?

Example1:Concrete Mediator

```
class FontDialogDirector : public DialogDirector {
public:
    FontDialogDirector();
    virtual ~FontDialogDirector();
    virtual void WidgetChanged(Widget*);

protected:
    virtual void CreateWidgets();

private:
    Button* _ok;
    Button* _cancel;
    ListBox* _fontList;
    EntryField* _fontName;
};
```

Example1:Concrete Mediator

```
void FontDialogDirector::WidgetChanged (
    Widget* theChangedWidget
) {
    if (theChangedWidget == _fontList) {
        _fontName->SetText(_fontList->GetSelection());

    } else if (theChangedWidget == _ok) {
        // apply font change and dismiss dialog
        // ...

    } else if (theChangedWidget == _cancel) {
        // dismiss dialog
    }
}
void FontDialogDirector::CreateWidgets () {
    _ok = new Button(this);
    _cancel = new Button(this);
    _fontList = new ListBox(this);
    _fontName = new EntryField(this);

    // fill the listBox with the available font names
```

Consequences: Benefits

- Improves comprehension of the system by centralized control logic
  - Hence simplifies maintenance
  - Also, when this logic need to be extended only the mediator class need to be extended.
- The colleague classes are totally decoupled.
  - Adding a new colleague class is very easy due to this decoupling level.
  - increases reusability of the colleagues of the mediator by decoupling them from the system
- Thus, increase cohesion and reduce coupling

Consequences -2

- The colleague objects must communicate only with the mediator objects.
  - Reduction from many-to-many to one-to-many and many-to-one.
  - *It simplifies object protocols.* A mediator replaces many-to-many interactions with one-to-many interactions between the mediator and its colleagues. One-to-many relationships are easier to understand, maintain, and extend.
- Reuse participants easily

Consequences-3

- *Limits subclassing.* A mediator localizes behavior that otherwise would be distributed among several objects. Changing this behavior requires subclassing Mediator only; Colleague classes can be reused as is.

- *Abstracts how objects cooperate.* Making mediation an independent concept and encapsulating it in an object lets you focus on how objects interact apart from their individual behavior. That can help clarify how objects interact in a system.

Consequences - Drawbacks

- Trades complexity of interaction for complexity in the mediator.
  - Because a mediator encapsulates protocols, it can become more complex than any individual colleague.
  - This can make the mediator itself a monolith that's hard to maintain.
- Overkill for simple interaction
- Without a proper design, the mediator itself becomes overly complex
  - Do not add extra responsibilities to mediator
  - E.g. DialogDirector class should not contain code which is not a part of the coordination.

Mediator – Known uses

- `java.awt.Container`: This class acts as a mediator between components and the specific layout manager

- `java.util.concurrent.Executor`: This interface is a mediator between a bunch of tasks that need to be run and a bunch of threads in which the tasks are run1.

Mediator vs Façade

- Both abstract functionality of existing classes.
- Mediator abstracts/centralizes arbitrary communication between colleague objects
  - routinely "adds value",
  - Known and referenced by the colleague objects
    - i.e. it defines a multidirectional protocol
- Facade defines a simpler interface to a subsystem,
  - doesn't add new functionality
  - not known by the subsystem classes

Observer vs Mediator

Mediator and Observer are competing patterns.

- communication:
  - Observer distributes communication by introducing "observer" and "subject" objects
  - Mediator object encapsulates the communication between other objects.
- Observer is one-to-many, Mediator is many-to-many
- They may work together
  - Mediator observing colleagues if something changed
  - Colleagues may subscribe to mediator