# More Patterns…

Acknowledgement: Head-first design patterns

# Chain of Responsibility

Acknowledgement: Head-first design patterns

# Problem

**Scenario:** *Paramount Pictures* has been getting more email than they can handle since the release of the Java-powered Ironman game. From their analysis they get four kinds of email:

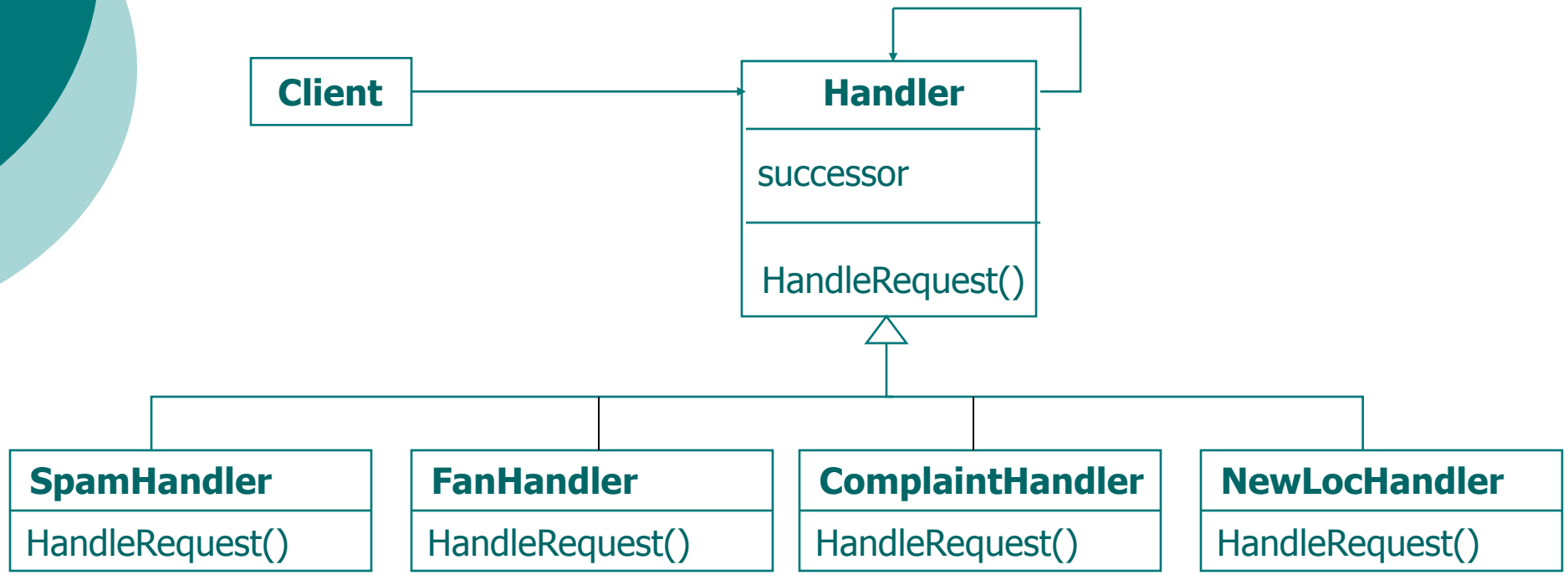
- Fan mail from customers that love the new 1 in 10 game
- Complaints from parents whose kids are addicted to the game
- Requests to put machines in new locations
- Spam

**Task:** They need you to create a design that can use the detectors to handle incoming email.

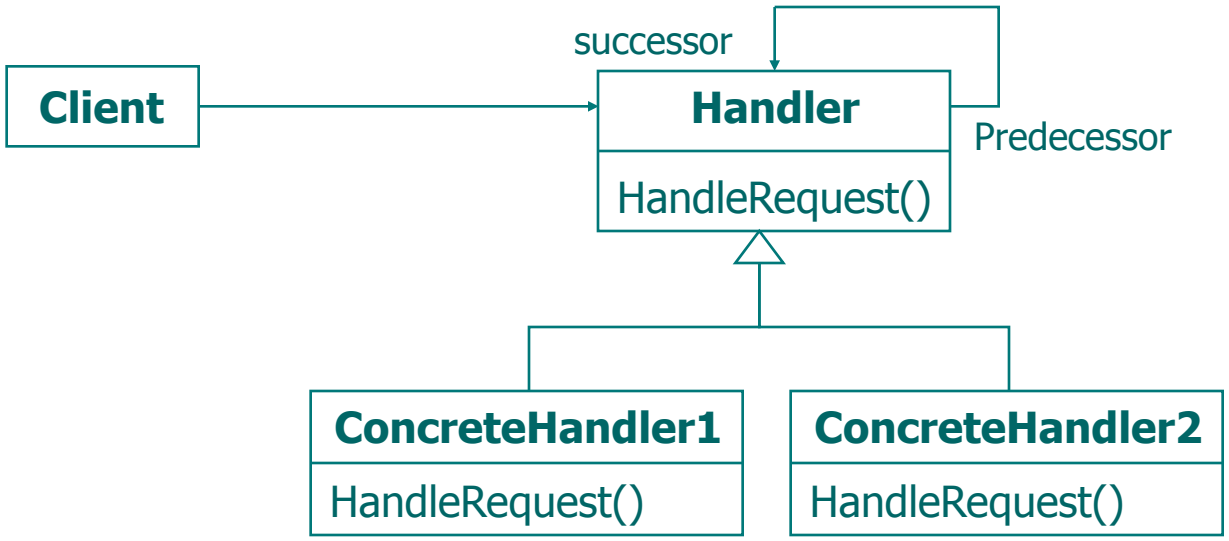
# Chain of Responsibility

- Create a chain of objects that examine a request
- Each object examines the request and handles it, or passes it on to the next object in the chain.

# Chain of Responsibility – Email Handler



# Chain of Responsibility - Structure



# Participants

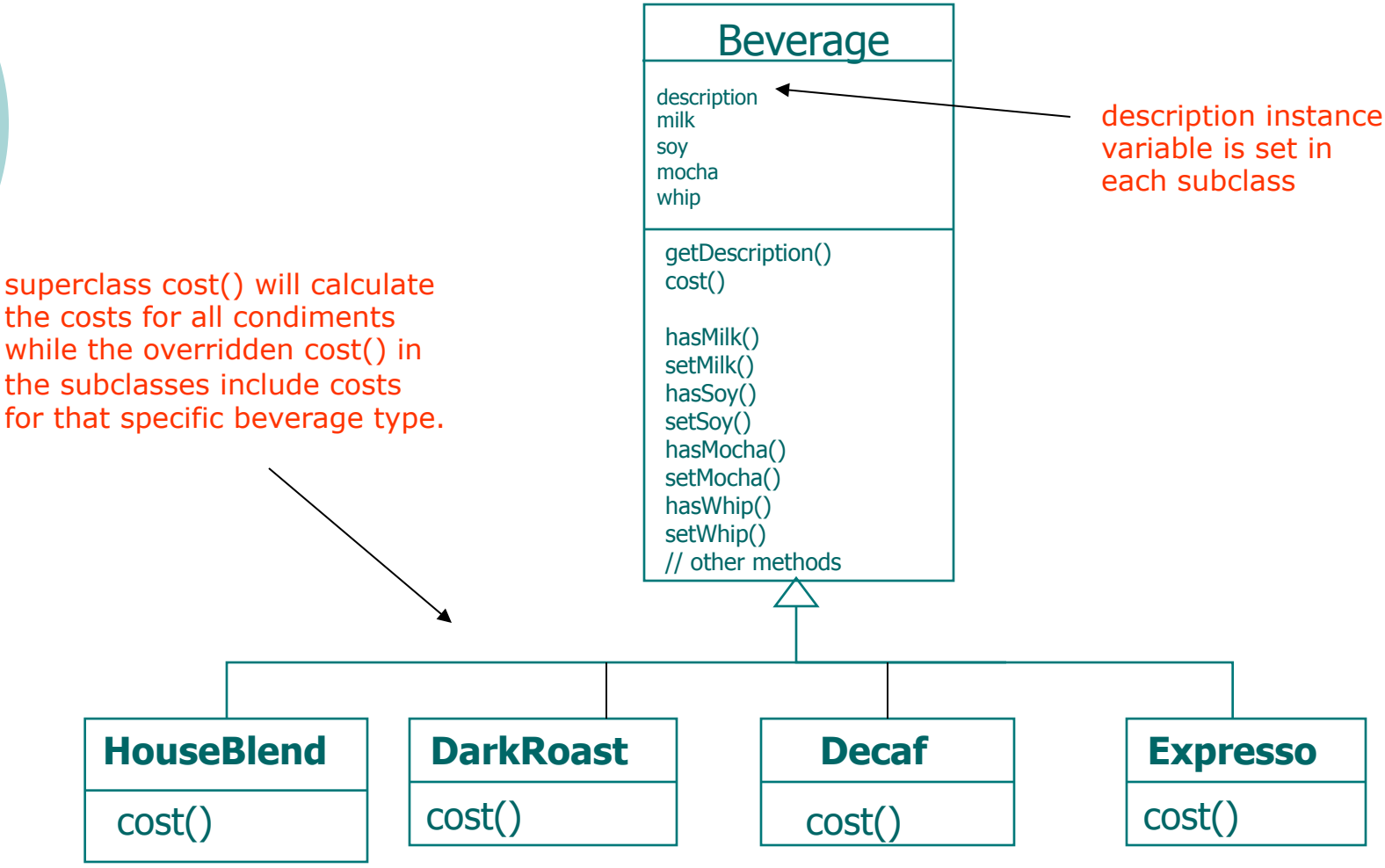
- Handler
  - Defines interface for handling request
  - Implements successor link
- ConcreteHandler
  - Handles all requests for which it is responsible
  - Has access to successor
  - Delegates request to successor when not handled directly
- Client
  - Initiates request to a ConcreteHandler

# Benefits and Drawbacks

- Decouples the sender of the request and its receivers
- Simplifies the object because it doesn't have to know the chain' entire structure
- Allows for adding or removing responsibilities dynamically by changing the members or order of the chain
- Commonly used in windows systems to handle events like mouse clicks and keyboard events
- Execution of a request isn't guaranteed
- Can be hard to observe the runtime characteristics and debug

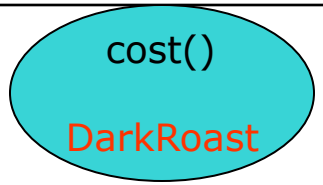
# Decorator Pattern

# Example - Starbuzz Coffee

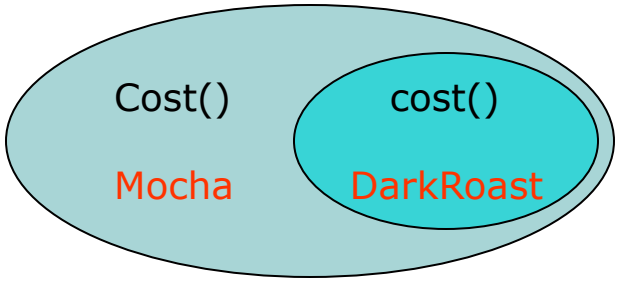


# Constructing a drink order with Decorators

- Start with the DarkRoast Object

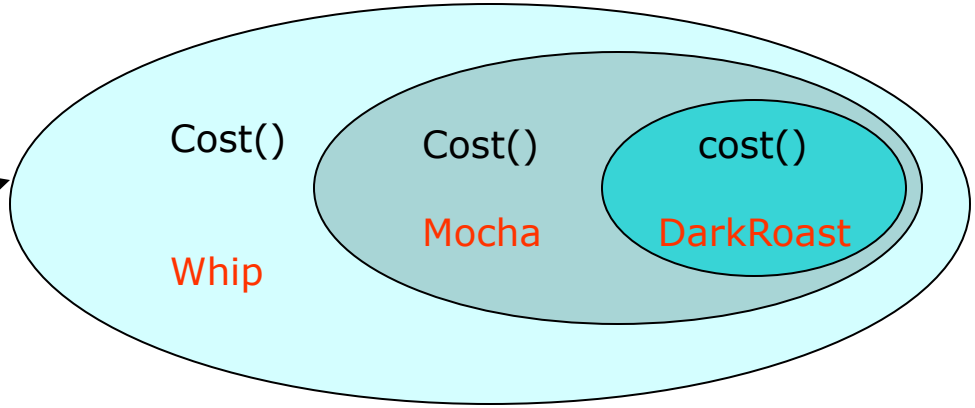



- Customer wants Mocha




Mocha object type mirrors the object it is decorating.

- Customer wants whip

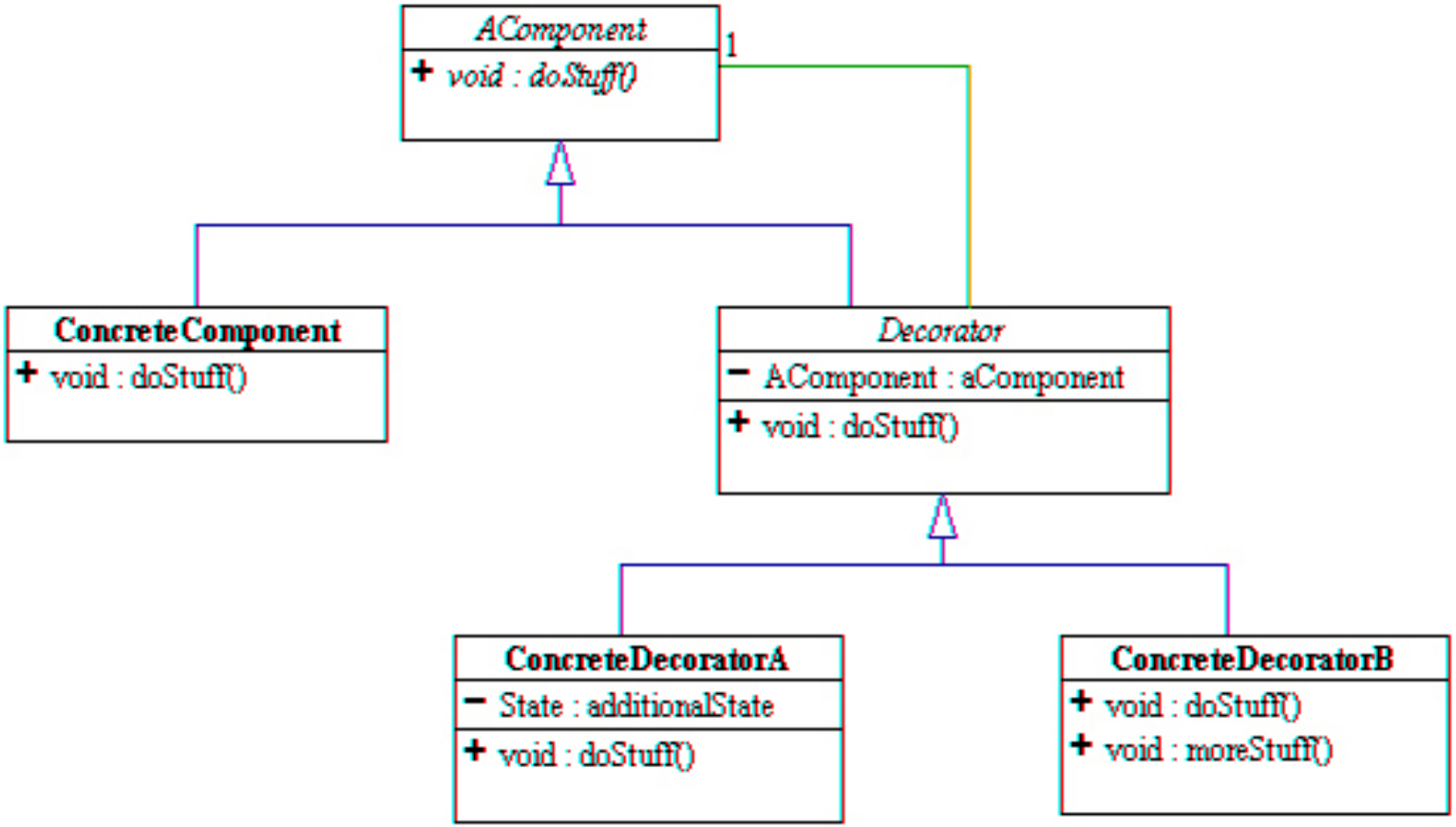



Whip object type mirrors the object it is decorating.

# Decorator pattern

- Decorators have the same super type as the objects they decorate
- You can use one or more decorators to wrap an object
- Given that the decorator has the same super type as the object it decorates, we can pass around a decorated object in place of the original object
- The decorator adds its own behavior either before and/or after delegating to the object it decorates to do the rest of the job
- Objects can be decorated at any time, so we can decorate objects dynamically at runtime with as many decorators as we like

# General Structure



# Mediator Pattern

# Problem

Scenario: Bob has an amazing auto-house. When Bob stops hitting the snooze button, his alarm clock tells the coffee maker to start brewing.

Bob is looking to add additional features:

- No coffee on the weekends…
- Turn of the sprinkler 15 min. before a shower is scheduled…
- Set the alarm early on trash days…

# Problem Representation

**Alarm**

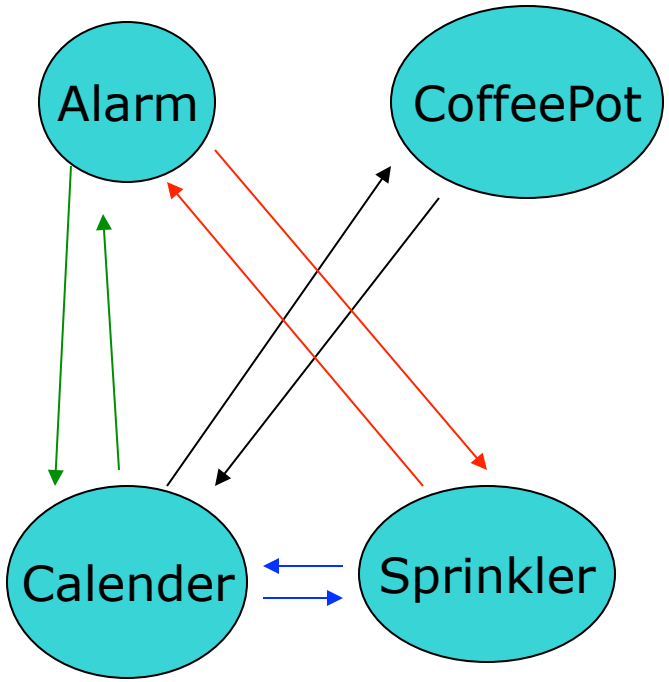
```
onEvent() {
  checkCalender()
  checkSprinkler()
  startCoffee()

 // more stuff
}
```

**CoffeePot**

```
onEvent() {
  checkCalender()
  checkAlarm()
  startCoffee()

 // more stuff
}
```




**Calender**

```
onEvent() {
  checkDayofWeek()
  doSprinkler()
  doCoffee()
  doAlarm()

 // more stuff
}
```

**Alarm**

```
onEvent() {
  checkCalender()
  checkShowerr()
  checkTemp()
  checkWeather()

 // more stuff
}
```

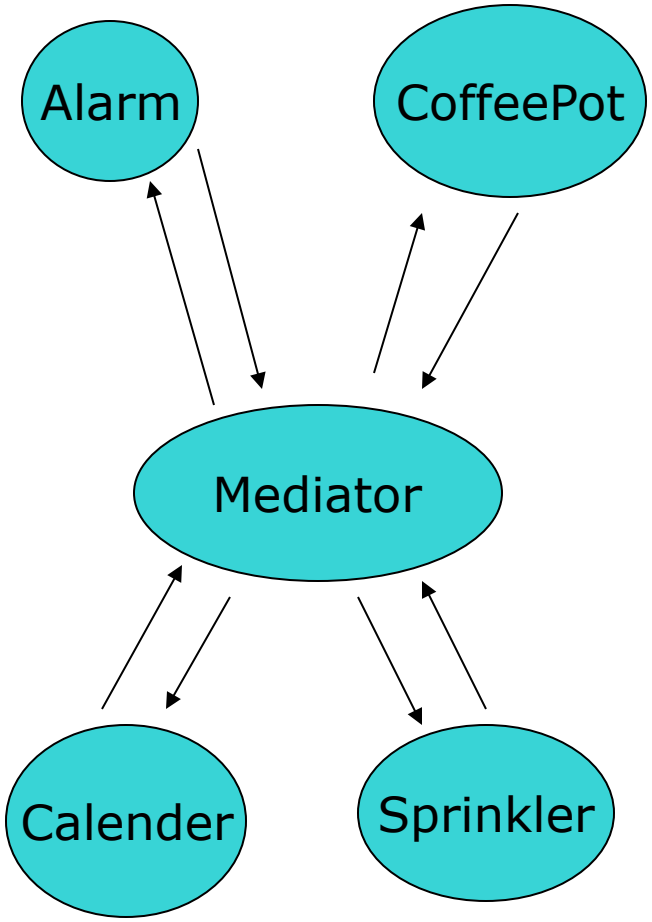
# Mediator in action…

**Mediator**

```
if (alarmEvent) {
  checkCalender()
  checkShower()
  checkTemp()
}

if (weekend) {
  checkWeather()
  // do more stuff
}

if (trashDay){
  resetAlarm()
  //do more stuff
}
```



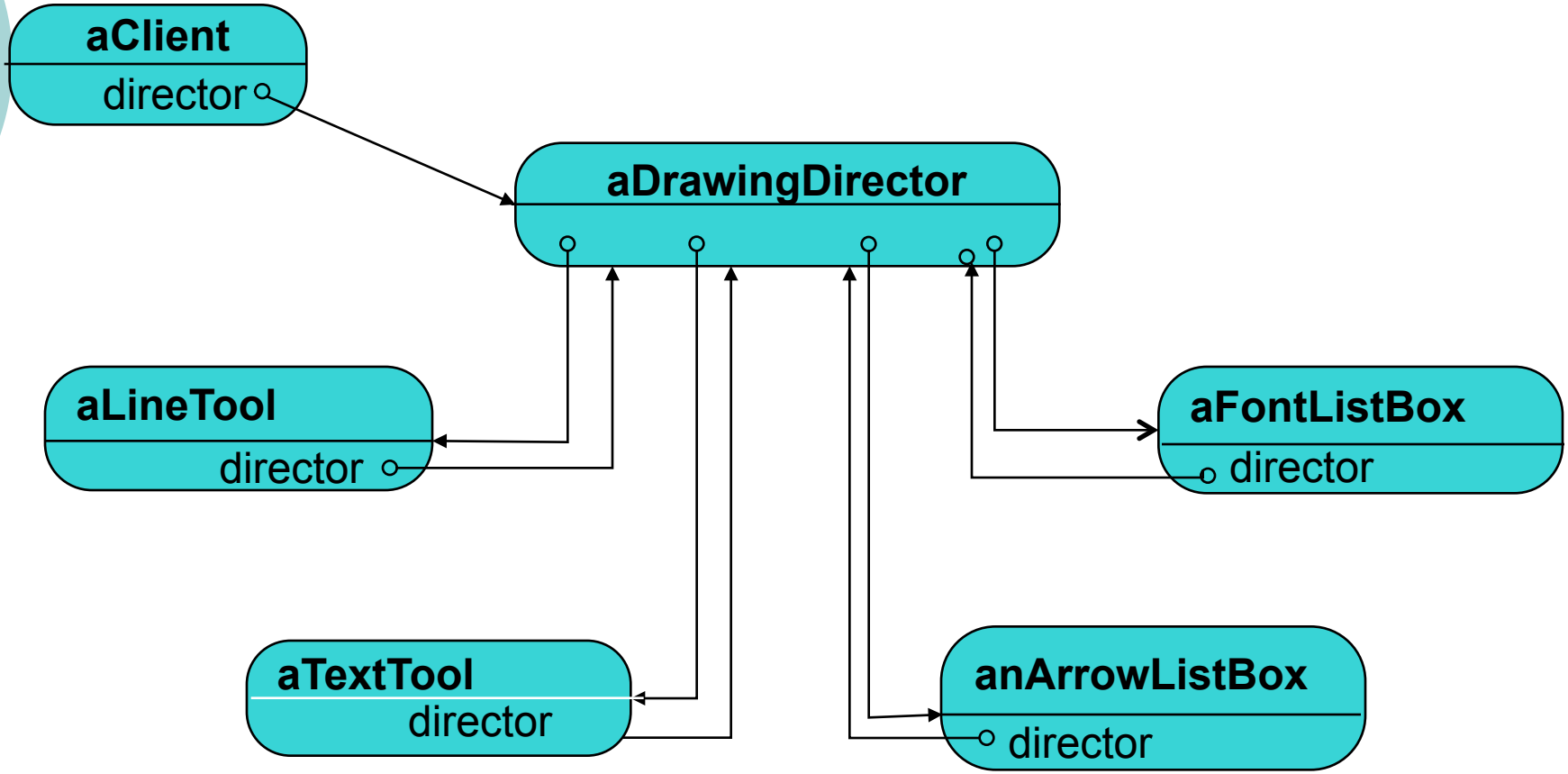
# Example: Drawing Editor

- The GUI widgets for a drawing editor are interrelated
- Examples:
  - Insert text enables font widgets and fill widgets
  - Insert 3-D object enables fill-in oriented widgets
  - Insert line or arc enables arrowhead options.
- How to coordinate all these widgets?
- Do not want to have every widget know about every other widget!

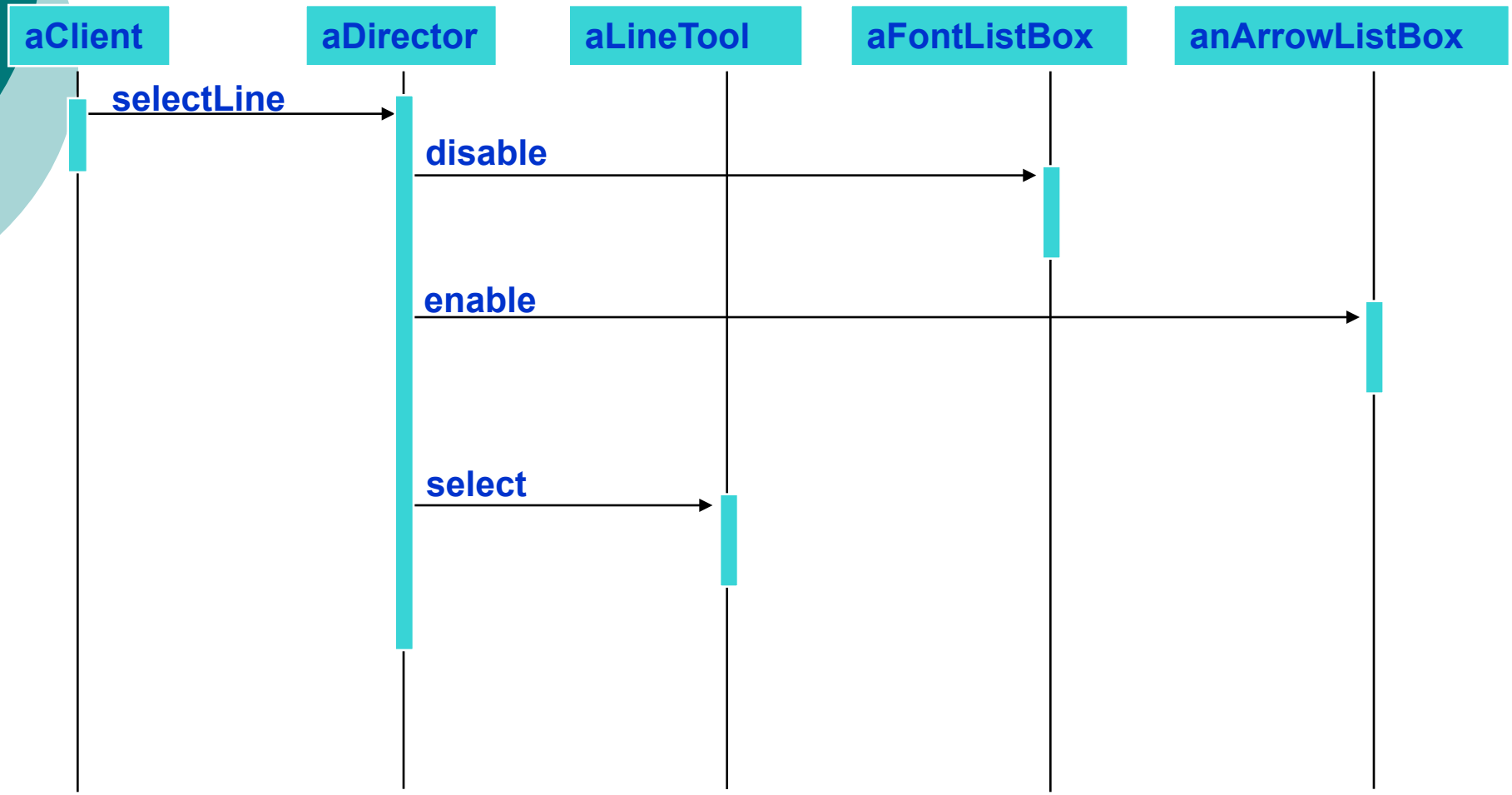
# Solution: Mediator Pattern

- Mediator/director objects synchronize object collections.
- Mediator observes all widgets it coordinates
  - Example: for drawing, every change to drawing tool is broadcasted by mediator to other connected widgets
- More efficient and comprehensible than every widget watching every other one.
- Centralized coordination.

# Example: Drawing Director



# Interaction Chart



# Participants

- Mediator
  - Interface for communications among colleague objects
  - Knows all the colleagues
  - Implements cooperative behavior
- Colleagues
  - Know their director / mediator
  - Communicate with mediator to distribute information to other colleagues

# Benefits and Drawbacks

- Increases the reusability of the objects supported by the Mediator by decoupling them from the system
- Simplifies maintenance of the system by centralizing logic
- Simplifies and reduces the variety of messages sent between objects in the system
- Without proper design, the Mediator object itself can become overly complex

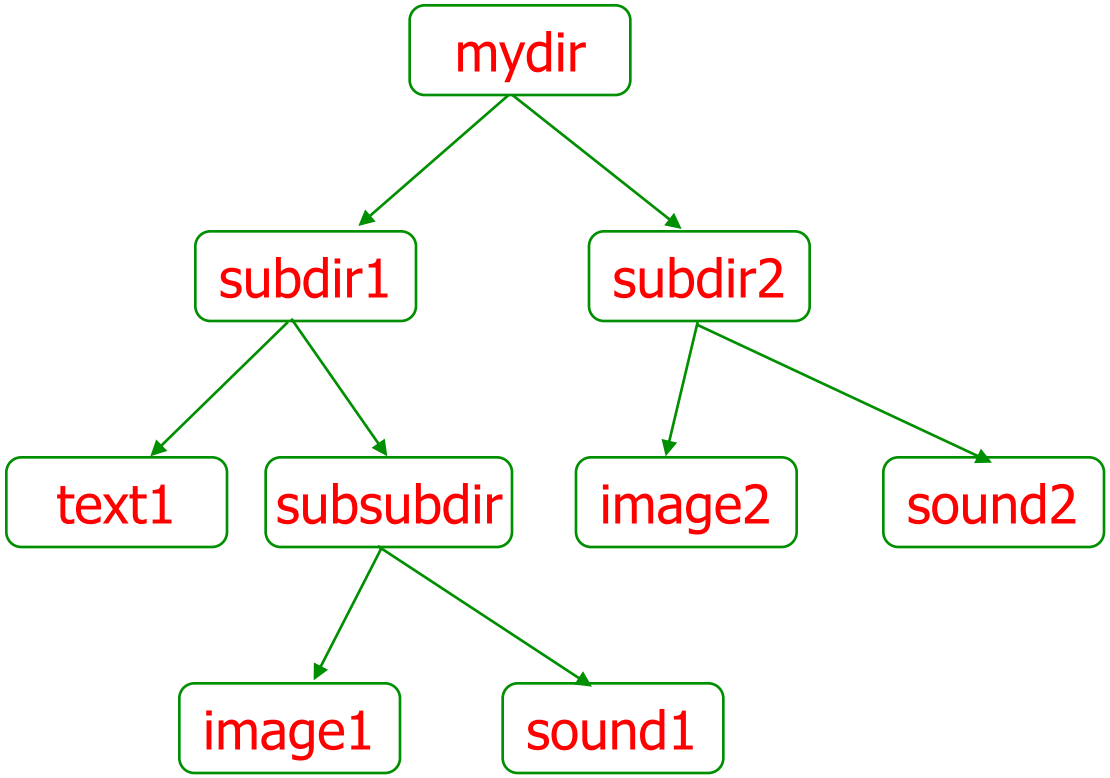
# Visitor Pattern

# The Problem

- Situation:
  - Operations to be performed on the elements of an object structure
- Desire:
  - Add new operations without changing the classes
  - Perform operation in a type-safe manner

# Example – File System

- Basic Types
  - Directories
  - Text Files
  - Image Files
  - Sound Files
- Operations
  - Total size
  - SearchImage



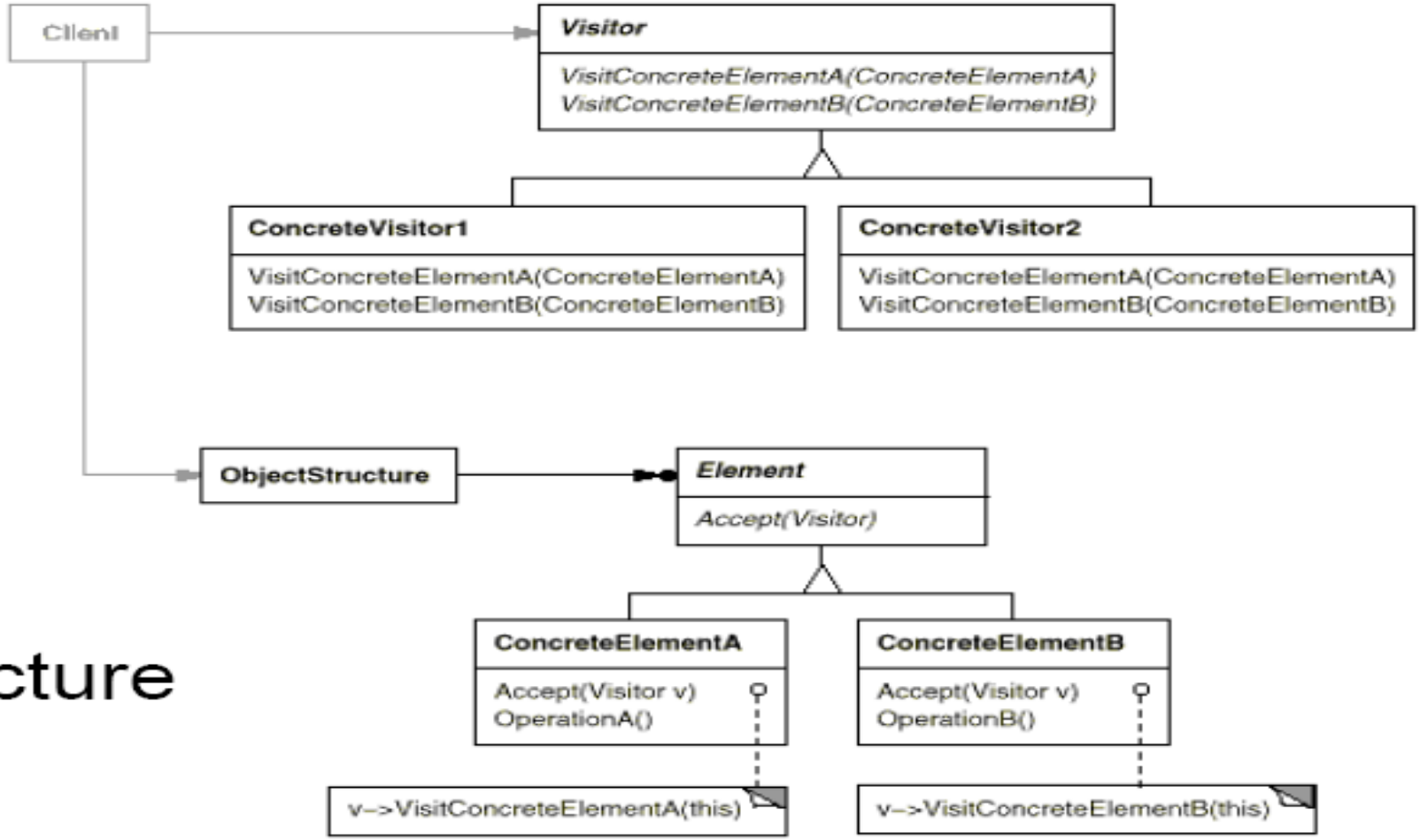
# Approach #1 – Operations in Interface

- Define all operations at topmost interface
- Override as necessary in each subclass
- Issue: adding new operations
  - Requires change to top most class
  - Requires *at least* recompilation of all subclasses
  - Typically adds code to each subclass
  - “Method bloat”

# Approach #2 - Visitor

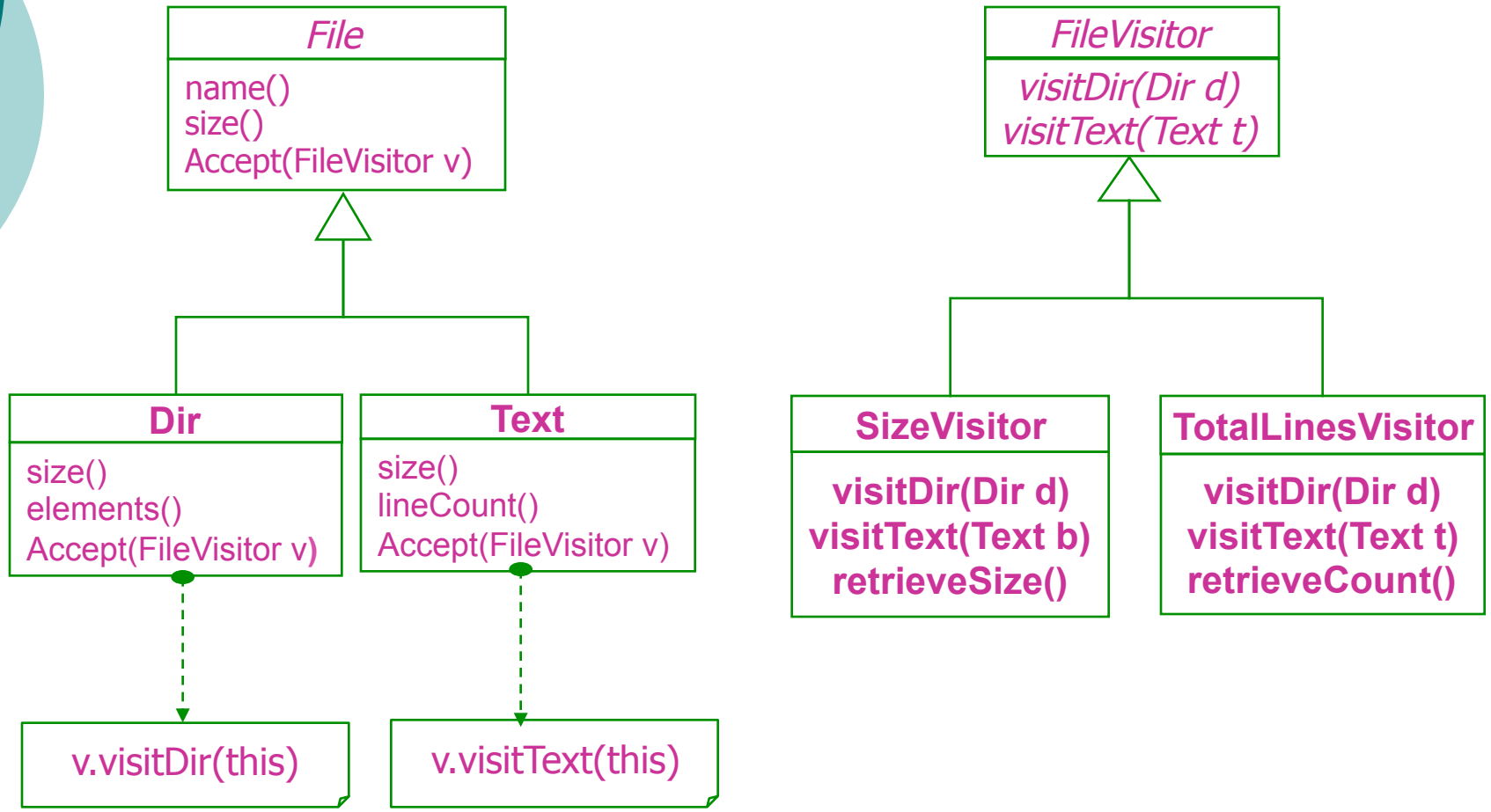
- Define Visitor interface with one method per subclass type (e.g., visitDirectory, visitImage)
- Define **Accept** method in topmost Element interface
- **Accept** method provided a Visitor object
- Each class's **Accept** method calls that class's specific method in the Visitor object.
- Inside each visitor's method(s) we have access to all functionality of the object being visited

# Structure




Structure

# Class Diagram



# Applicability

- Many distinct, unrelated operations are to be performed
- You want to minimize the common interface
  - Isolates conceptual operation in its own class
  - Supports different operations in different applications that share the overall object structure.
- Result
  - The object structure rarely change
  - The operations over the structure change frequently

# Drawbacks

Adding new concrete element classes is difficult:

- **All visitors must change**
- **Here putting functionality in structure probably better**

| New Classes? / New Ops? | Rare | Frequent |
|---|---|---|
| Rare | **Either** | **Structure** |
| Frequent | **Visitor** | **Hard** |