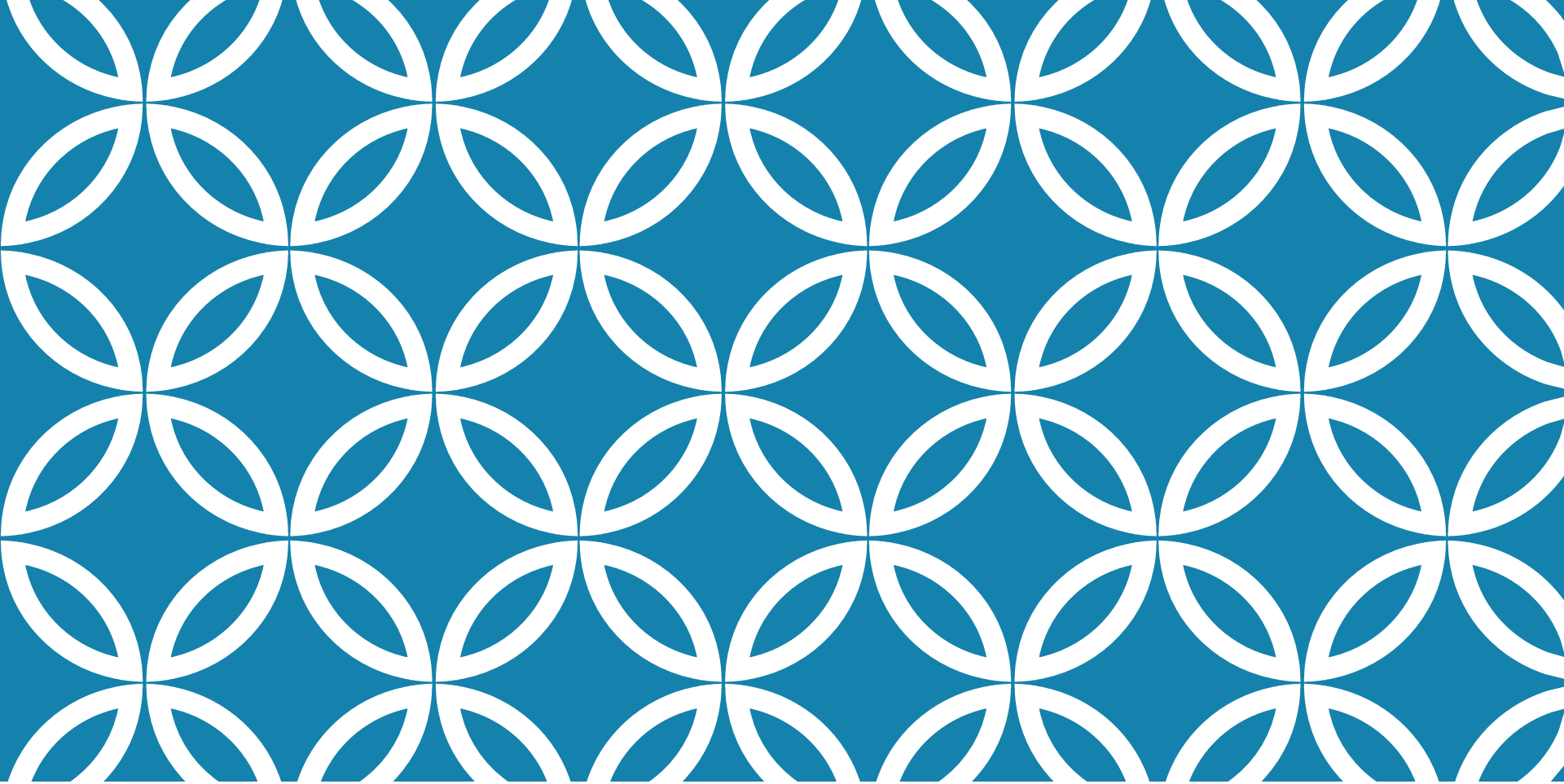


BEHAVIOURAL DP

Lecture 11

CONTENT

Design Patterns

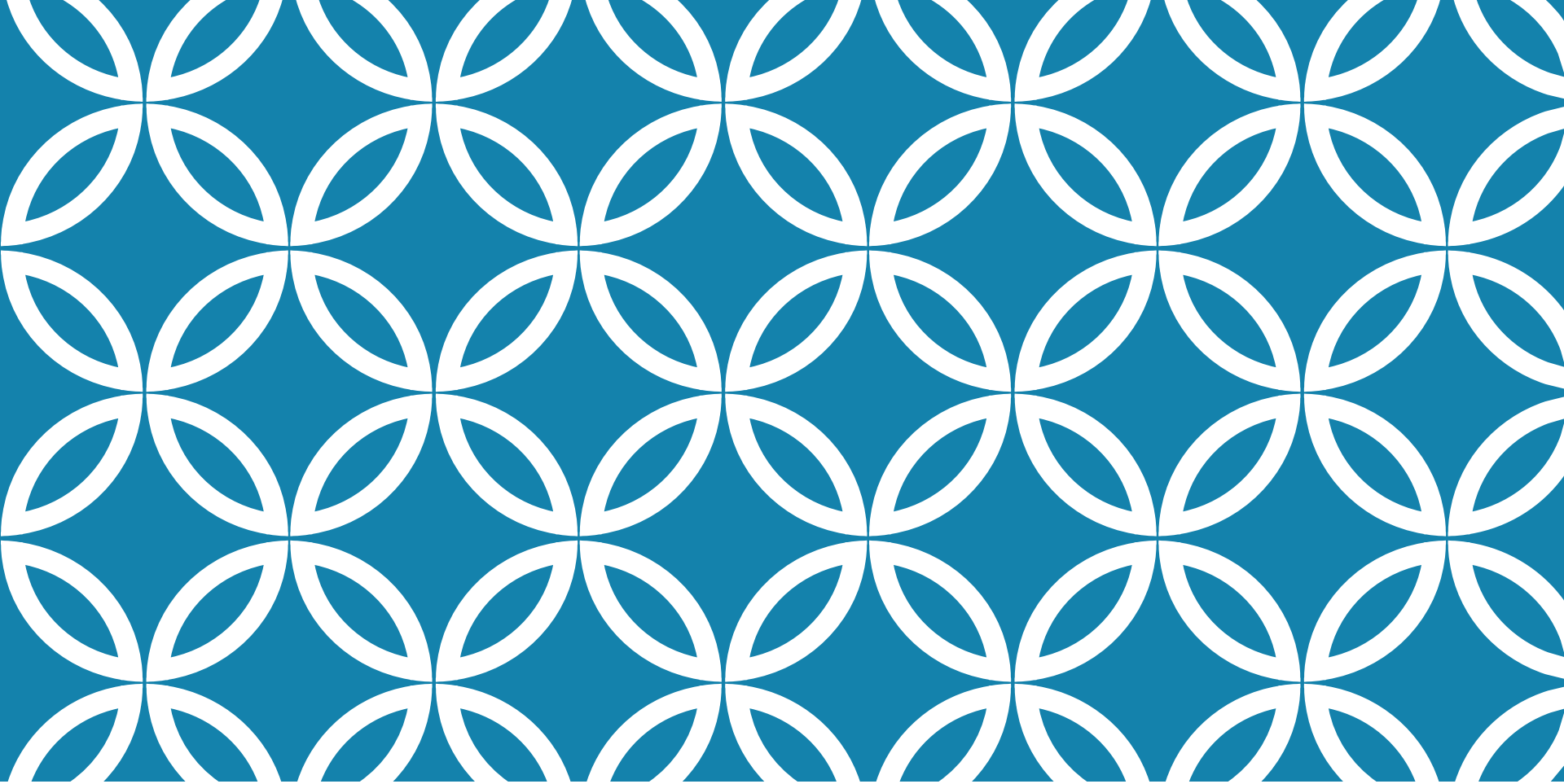
- Creational Patterns
- Structural Patterns
- Behavioural Patterns
  - Observer
  - Strategy
  - State
  - Command
  - Chain of Responsibility

REFERENCES

Erich Gamma, et.al, Design Patterns: Elements of Reusable Object-Oriented Software, Addison Wesley, 1994, ISBN 0-201-63361-2.

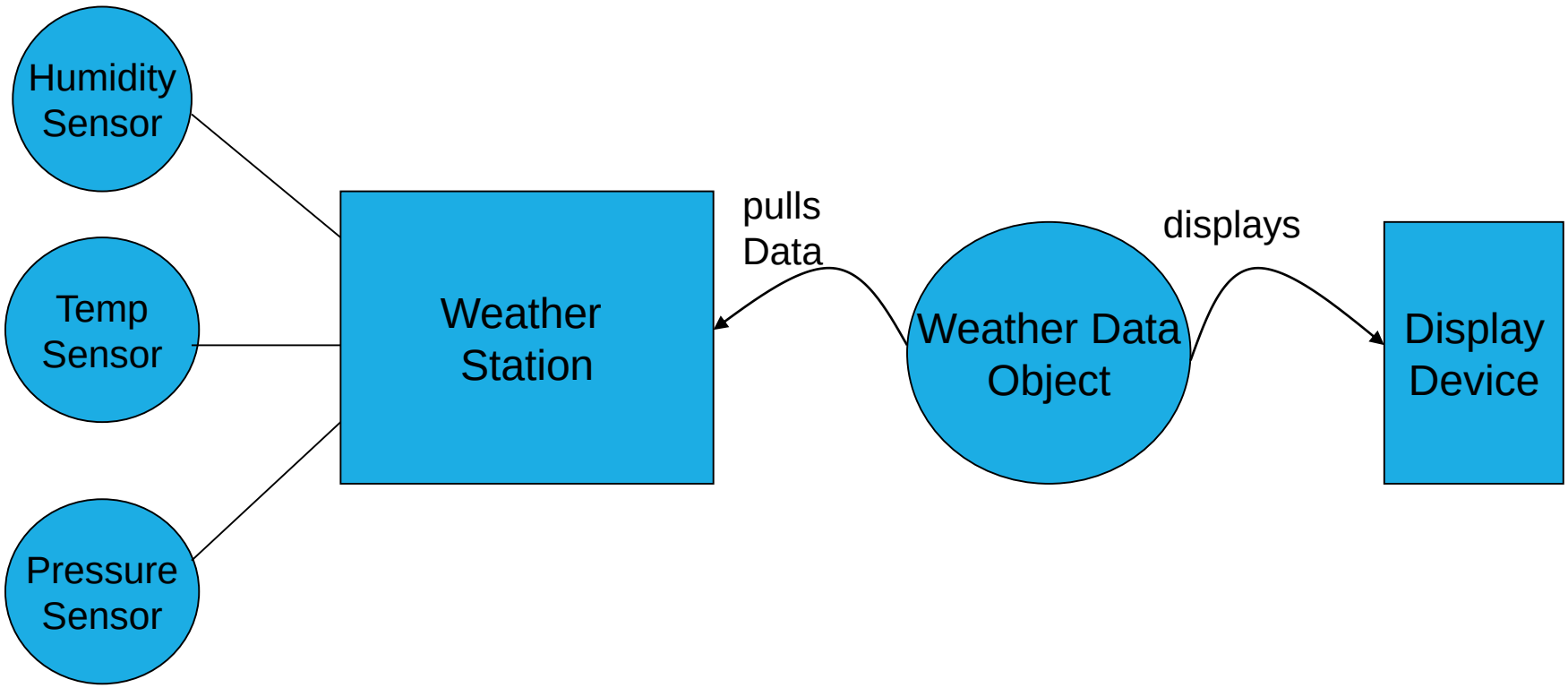
BEHAVIOURAL PATTERNS

- are concerned with **algorithms** and the **assignment of responsibilities** between objects.
- describe **patterns of communication** between classes/objects.
- **class** patterns use inheritance to distribute behavior between classes.
- **object** patterns use object composition rather than inheritance.

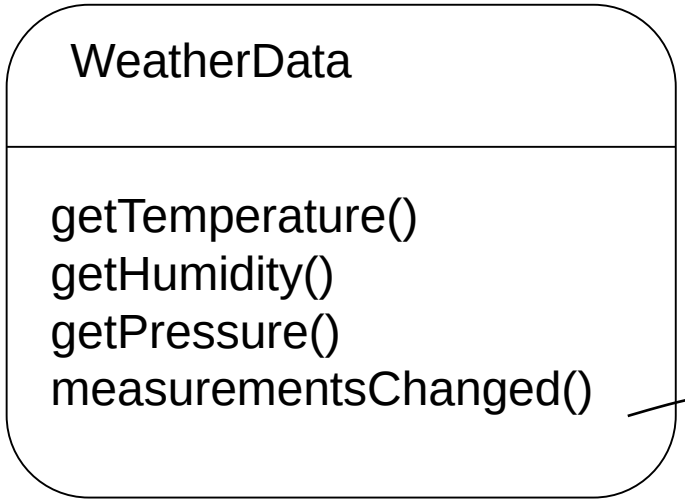


OBSERVER PATTERN

WEATHER MONITORING APPLICATION



WHAT NEEDS TO BE DONE?




```
/* Call this method whenever
 measurements are updated
*/

public void measurementsChanged() {
 // your code goes here
}
```

Update three different displays

PROBLEM SPECIFICATION

- WeatherData class has three getter methods
- measurementsChanged() method called whenever there is a change
- Three display methods needs to be supported:
  - current conditions,
  - weather statistics and
  - simple forecast
- System should be expandable

FIRST IMPLEMENTATION IDEA

```
public class WeatherData {
  public void measurementsChanged(){
        float temp = getTemperature();
        float humidity = getHumidity();
        float pressure = getPressure();
        currentConditionsDisplay.update (temp, humidity,
 pressure);
        statisticsDisplay.update (temp, humidity, pressure);
        forecastDisplay.update (temp, humidity, pressure);
 }
 // other methods
}
```

FIRST IMPLEMENTATION IDEA

```
public class WeatherData {

        public void measurementsChanged(){

        float temp = getTemperature();

        float humidity = getHumidity();

        float pressure = getPressure();


        currentConditionsDisplay.update (temp, humidity,
 pressure);

        statisticsDisplay.update (temp, humidity, pressure);

        forecastDisplay.update (temp, humidity, pressure);

 }

 // other methods

}
```

Area of change which can be Managed better by encapsulation

By coding to concrete implementations there is no way to add additional display elements without making code change

BASIS FOR OBSERVER PATTERN

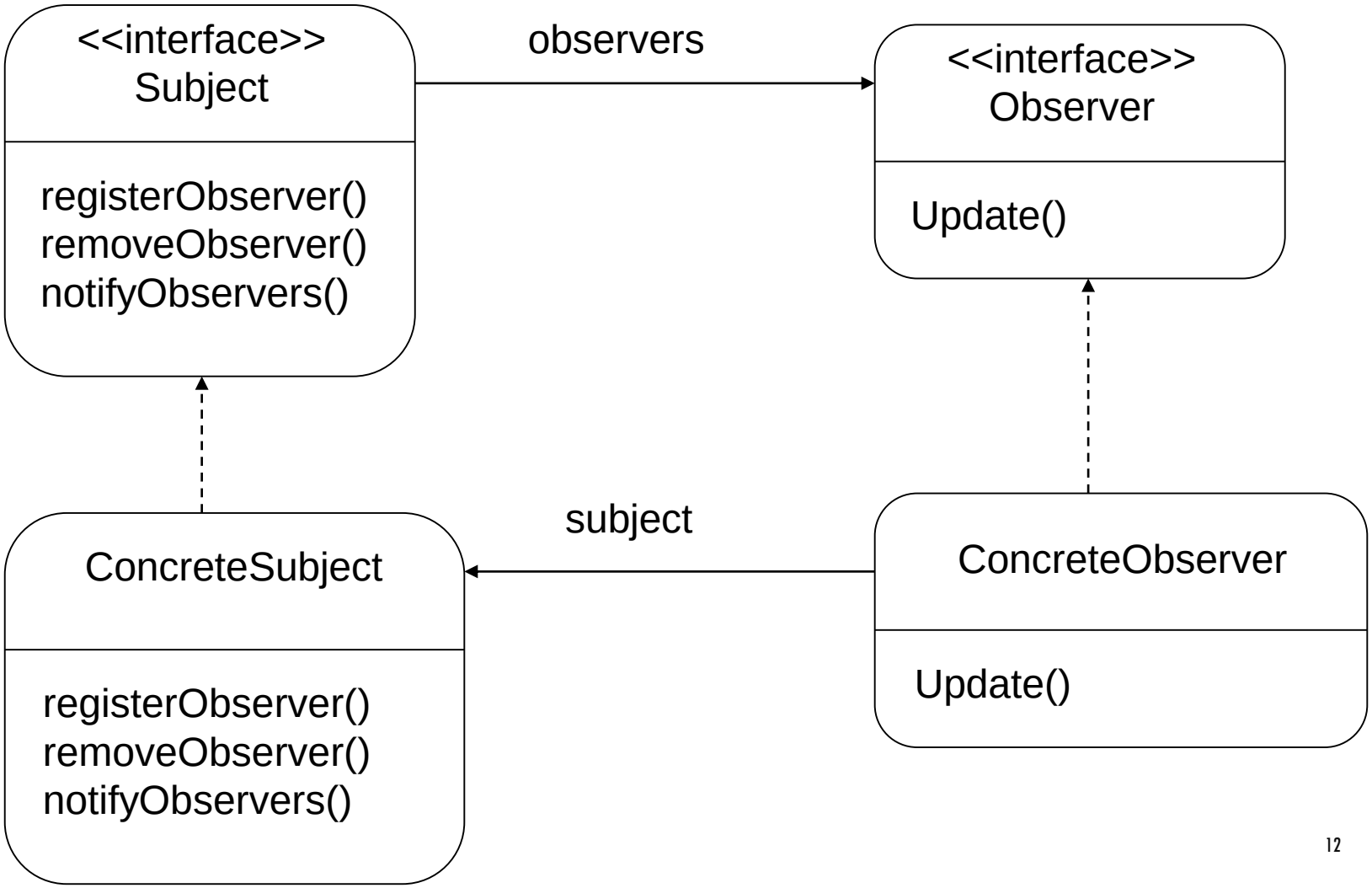
Fashioned after the publish/subscribe model

Works off similar to any subscription model

- Buying newspaper
- Magazines
- List servers

The Observer Pattern defines a one-to-many dependency between objects so that when one object changes state, all of its dependents are notified and updated automatically.

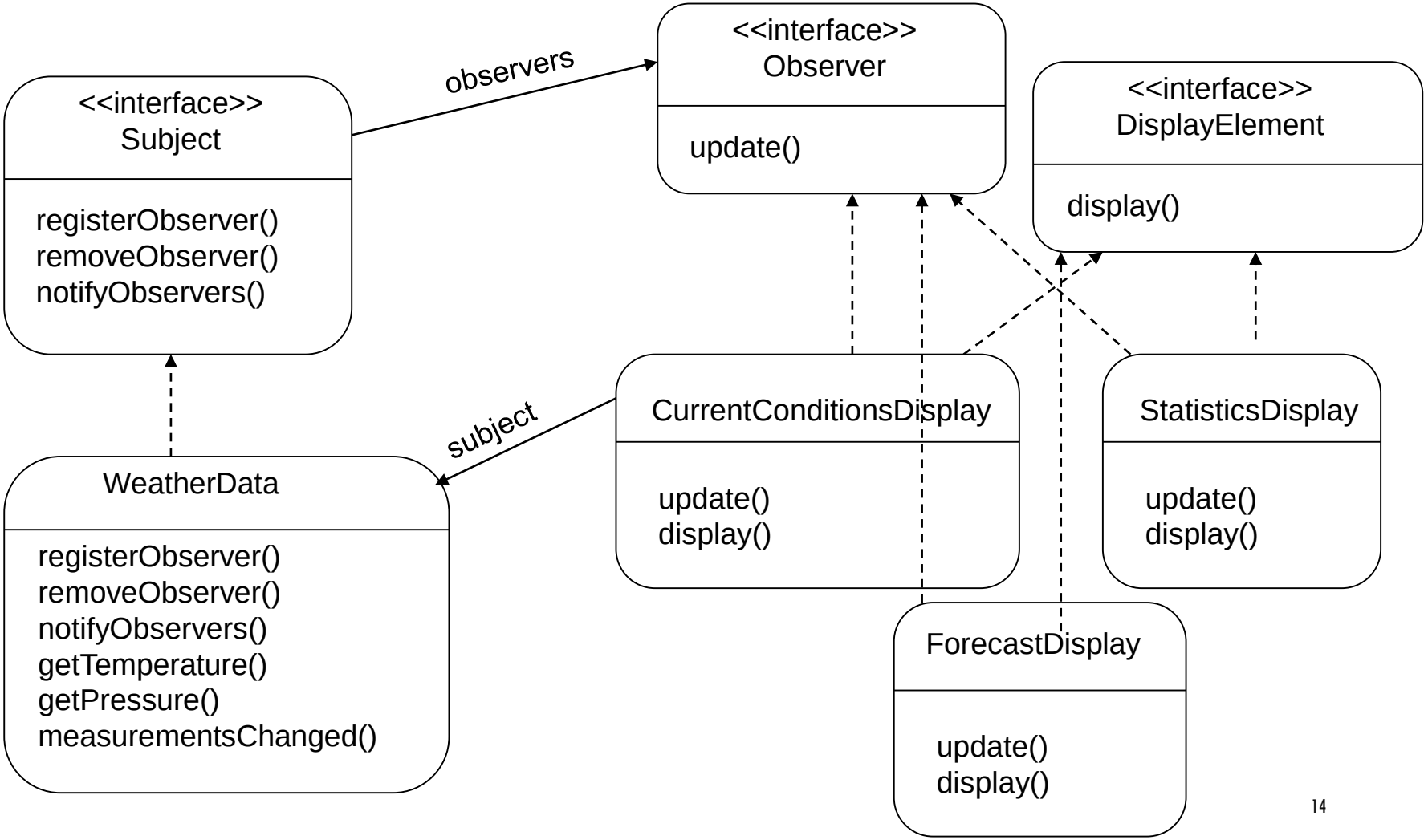
OBSERVER PATTERN – CLASS DIAGRAM



POWER OF LOOSE COUPLING

- The only thing that the subject knows about an observer is that it implements an interface
- Observers can be added at any time and subject need not be modified to add observers
- Subjects and observers can be reused or modified without impacting the other [as long as they honor the interface commitments]

WEATHER DATA REVISITED



WEATHER DATA INTERFACES

```
public interface Subject {
 public void registerObserver(Observer o);
 public void removeObserver(Observer o);
 public void notifyObservers();
}
public interface Observer {
 public void update(float temp, float humidity, float
 pressure);
}
public interface DisplayElement {
 public void display();
}
```

IMPLEMENTING THE SUBJECT INTERFACE

```
public class WeatherData implements Subject {
      private ArrayList observers;
      private float temperature;
      private float humidity;
      private float pressure;

      public WeatherData() {
            observers = new ArrayList();
      }
      …}
```

REGISTER AND UNREGISTER

```
public void registerObserver(Observer o) {
      observers.add(o);
}

public void removeObserver(Observer o) {
      int i = observers.indexOf(o);
      if (i >= 0) {
            observers.remove(i);
      }
}
```

NOTIFY METHODS

```
public void notifyObservers() {
  for (int i=0; i<observers.size(); i++)
   {
     Observer observer = (Observer)observers.get(i);
     observer.update(temperature, humidity, pressure);
   }
}

public void measurementsChanged() {
      notifyObservers()
}
```

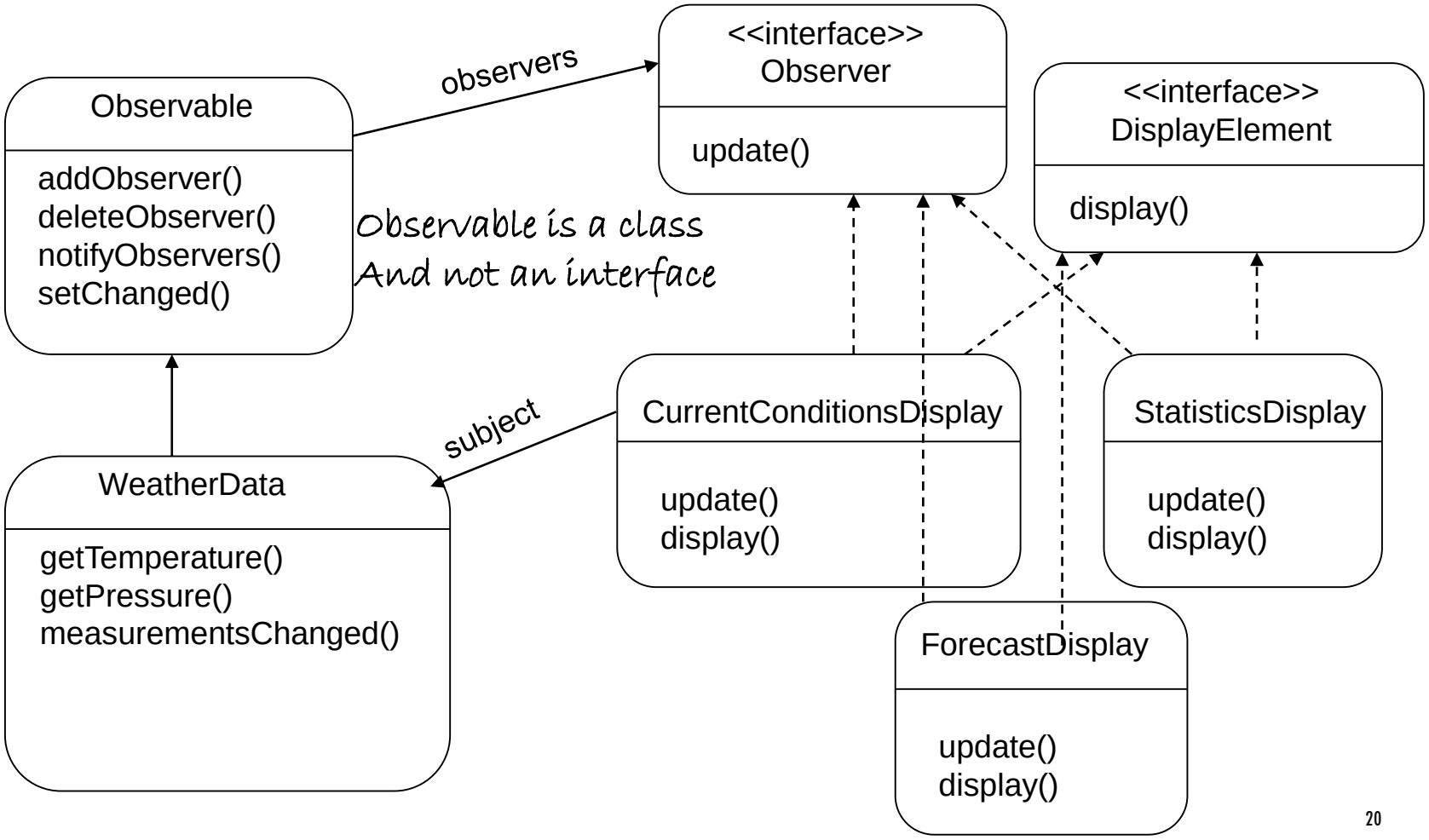
DISCUSSION

- The notification approach used so far **pushes** all the state to all the observers

- One can also just send a notification that some thing has changed and let the observers **pull** the state information

- Java observer pattern support has built in support for both push and pull in notification
  - **java.util.Observable**
  - **java.util.Observer**

JAVA OBSERVER PATTERN – WEATHER DATA



PROBLEMS WITH JAVA IMPLEMENTATION

Observable is a class

- You have to subclass it
- You cannot add observable behavior to an existing class that already extends another superclass
- ***You have to program to an implementation – not interface***

Observable protects crucial methods

- Methods such as setChanged() are protected and not accessible unless one subclasses Observable.
- ***You cannot favor composition over inheritance.***

You may have to write your own observer interface if Java utilities don't work for your application

CHANGING THE "GUTS" OF AN OBJECT ...

Control

- "shield" the implementation from direct access (Proxy)

Decouple

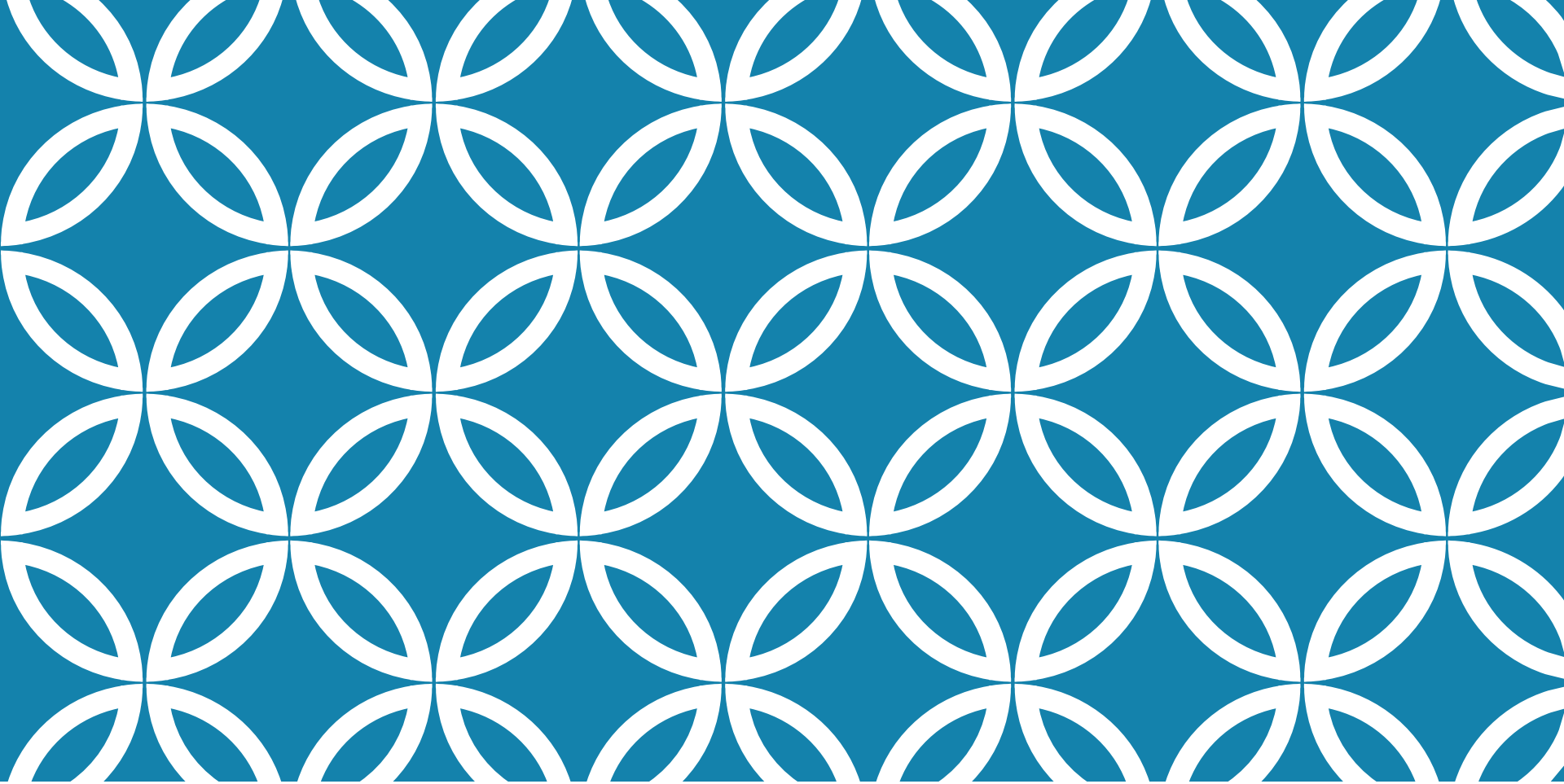
- let abstraction and implementation vary independently (Bridge)

Optimize

- use an alternative algorithm to implement behavior (Strategy)

Alter

- change behavior when object's state changes (State)



STRATEGY PATTERN

JAVA LAYOUT MANAGERS

GUI container classes in Java

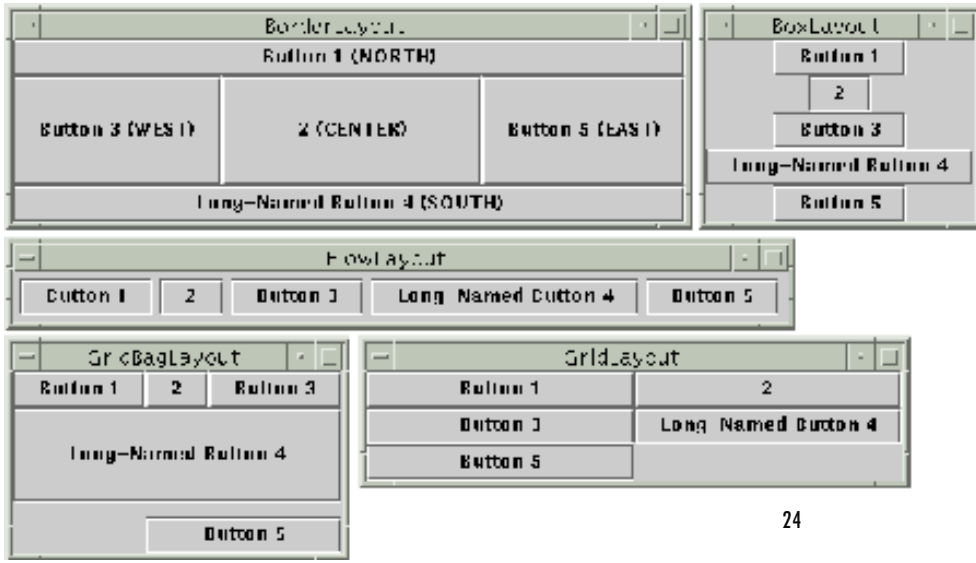
- frames, dialogs (top-level)
- panels (intermediate)

Each container class has a layout manager

- determine the size and position of components
- 20 types of layouts
- ~40 container-types
- imagine to combine them freely by inheritance

Consider also sorting...

- open-ended number of sorting criteria



BASIC ASPECTS

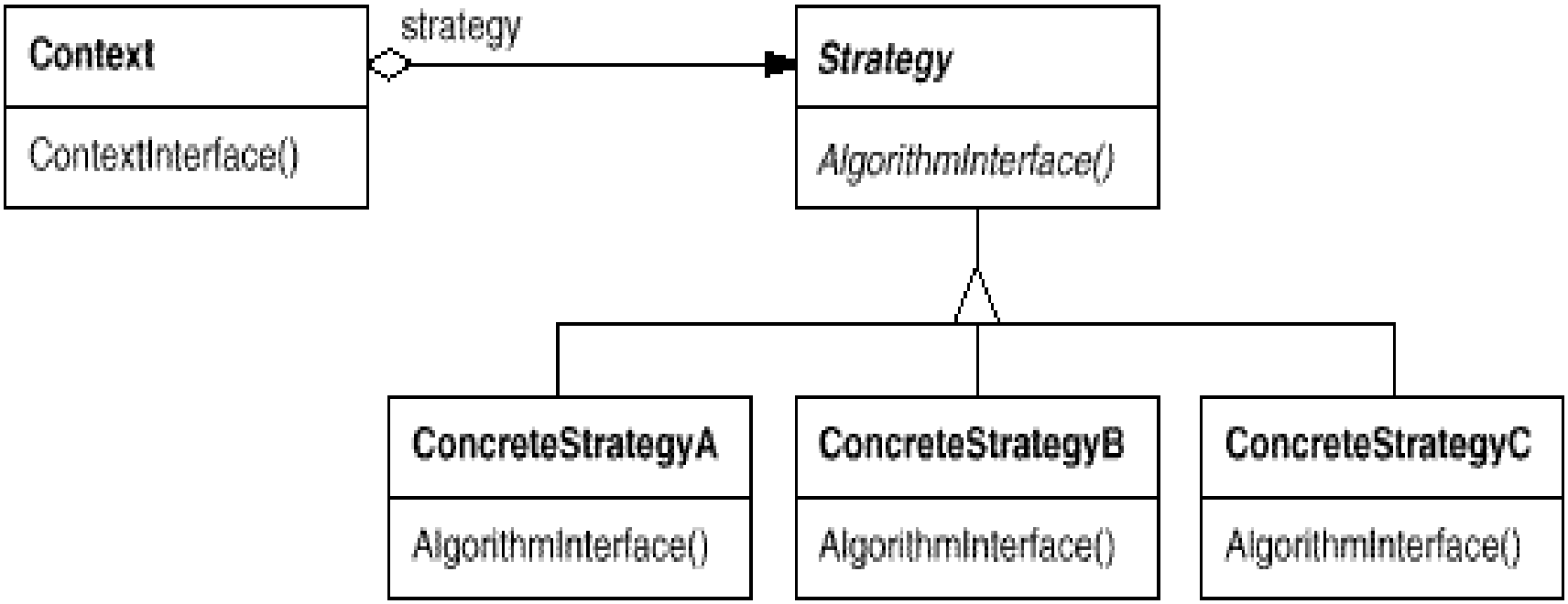
Intent

- Define a family of algorithms, encapsulate each one, and make them interchangeable
- Let the algorithm vary independently from clients that use it

Applicability

- You need different variants of an algorithm
- An algorithm uses data that clients shouldn't know about
  - avoid exposing complex, algorithm-specific data structures
- Many related classes differ only in their behavior
  - configure a class with a particular behavior

STRUCTURE


Context
ContextInterface()
strategy
Strategy
AlgorithmInterface()
ConcreteStrategyA
AlgorithmInterface()
ConcreteStrategyB
AlgorithmInterface()
ConcreteStrategyC
AlgorithmInterface()

PARTICIPANTS

Strategy

- declares an interface common to all supported algorithms.
- Context uses this interface to call the algorithm defined by a ConcreteStrategy

ConcreteStrategy

- implements the algorithm using the Strategy interface

Context

- configured with a ConcreteStrategy object
- may define an interface that lets Strategy objects to access its data

CONSEQUENCES

Families of related algorithms

- usually provide different implementations of the same behavior
- choice decided by time vs. space trade-offs

Alternative to subclassing

- see examples with layout managers
- We still subclass the strategies…

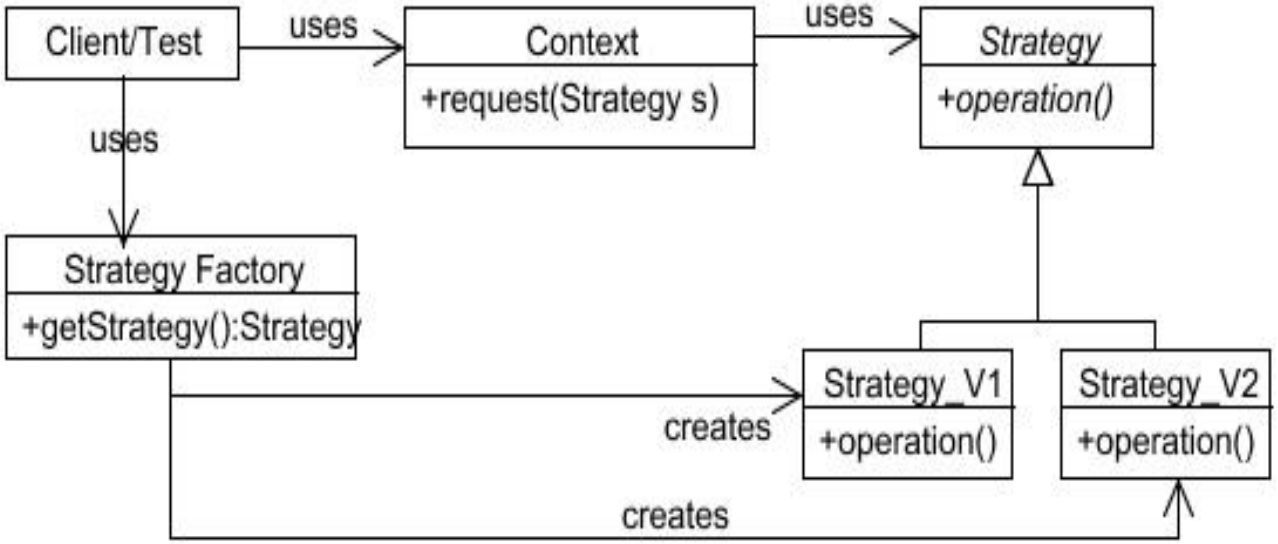
Eliminates conditional statements

- many conditional statements → "invitation" to apply Strategy!

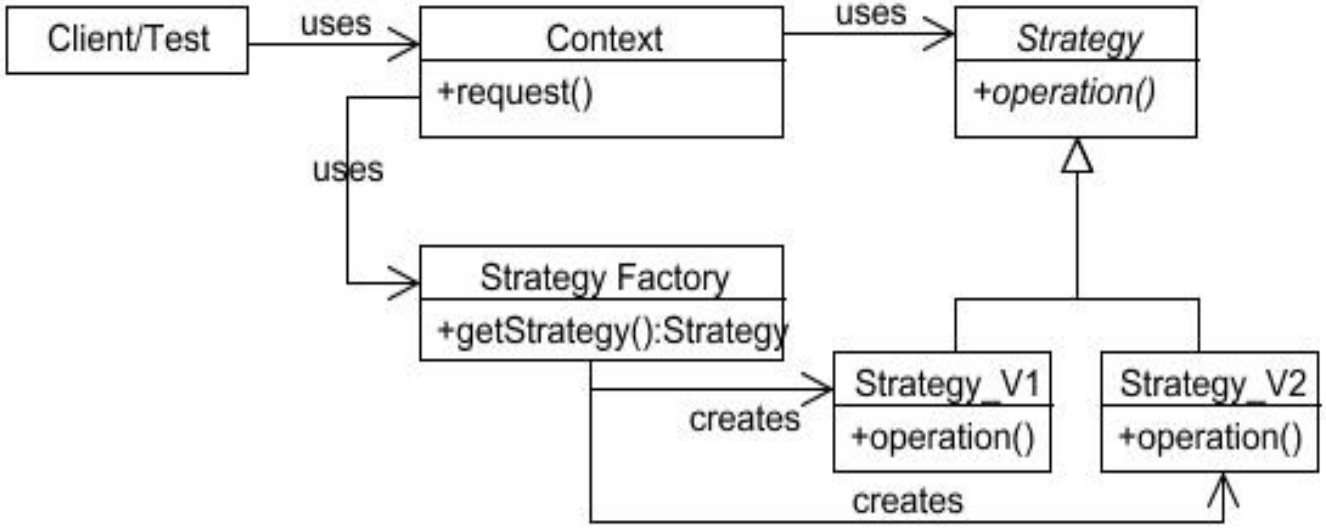
ISSUES

Who chooses the strategy?

Client



Context



IMPLEMENTATION

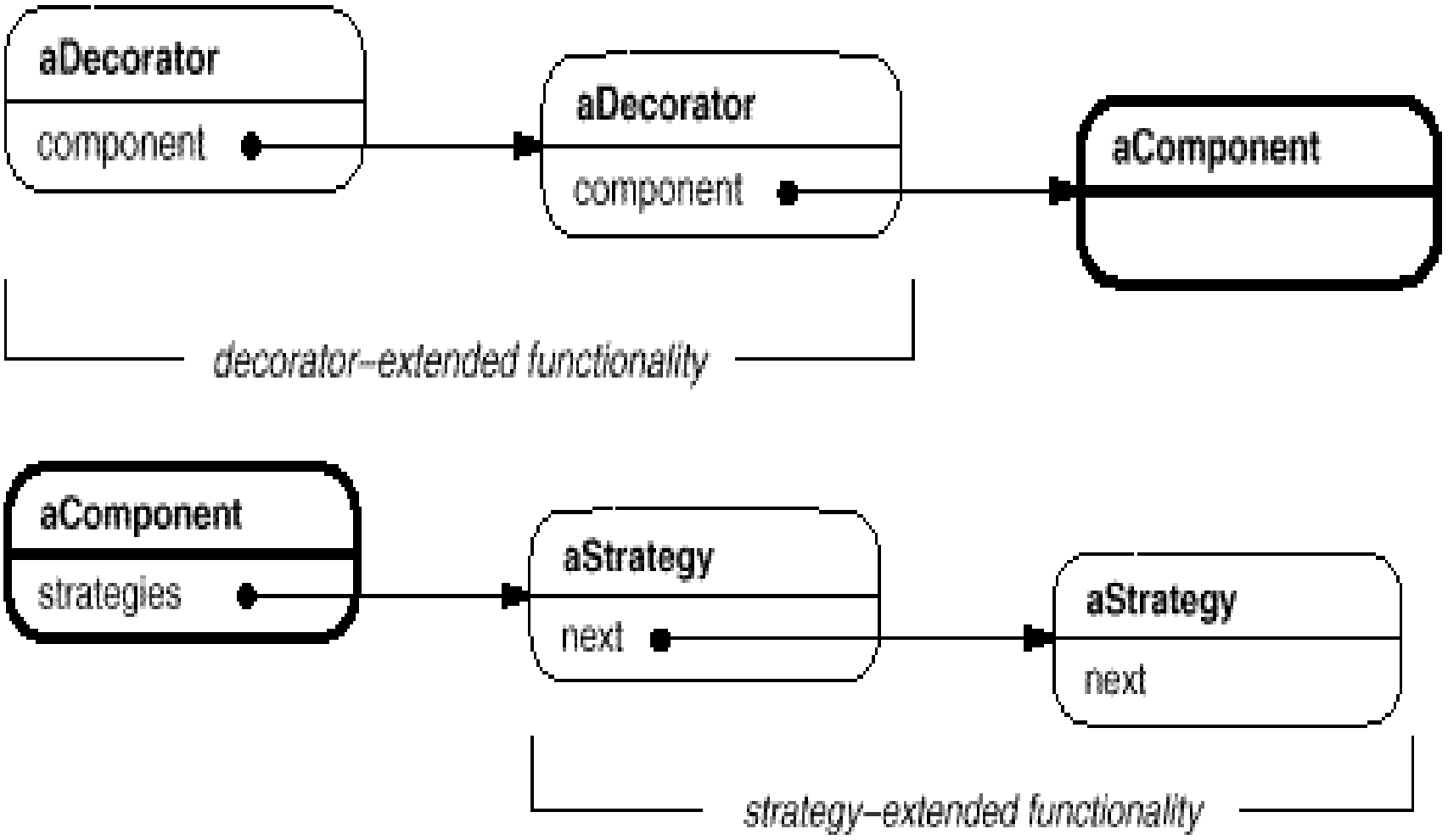
How does data flow between Context and Strategies?

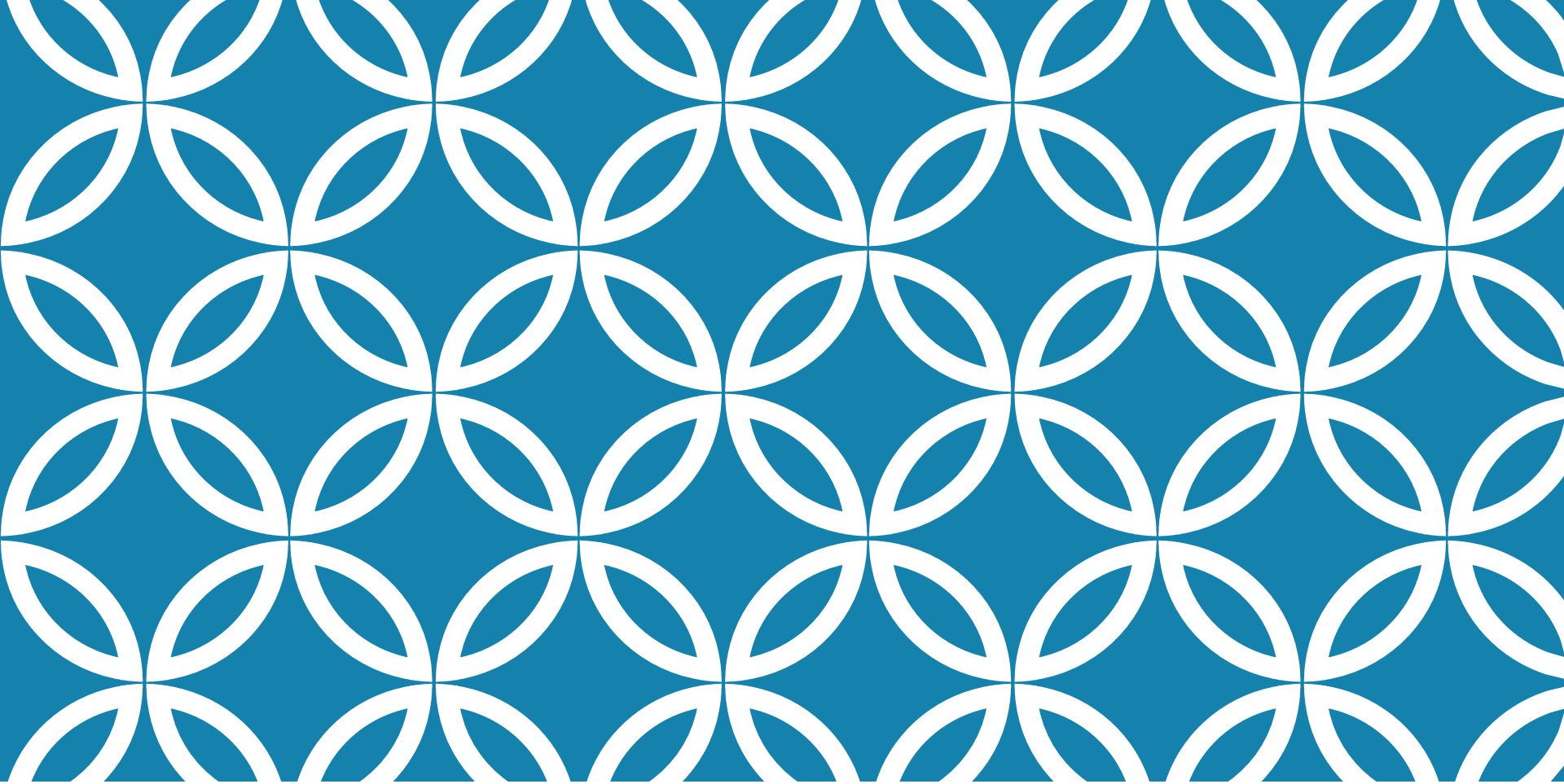
- **Approach 1:** take data to the strategy
  - decoupled, but might be inefficient
- **Approach 2:** pass Context itself and let strategies take data
  - Context must provide a more comprehensive access to its data => more coupled
- In Java, the strategy hierarchy might be inner classes

Making Strategy object optional

- provide Context with default behavior
  - if default used no need to create Strategy object
- don't have to deal with Strategy unless you don't like the default behavior

DECORATOR VS. STRATEGY





STATE PATTERN

EXAMPLE: SPOP

SPOP = **S**imple **P**ost **O**ffice **P**rotocol

- used to download emails from server

SPOP supports the following commands:

- USER <username>
- PASS <password>
- LIST
- RETR <message number>
- QUIT

USER & PASS commands

- USER with a username must come first
- PASS with a password or QUIT must come after USER
- If the username and password are valid, the user can use other commands

SPOP (CONTD.)

LIST command

- Arguments: a message-number (optional)
- Returns: size of message in octets
  - if message number, returns the size of that message
  - otherwise return size of all mail messages in the mail-box

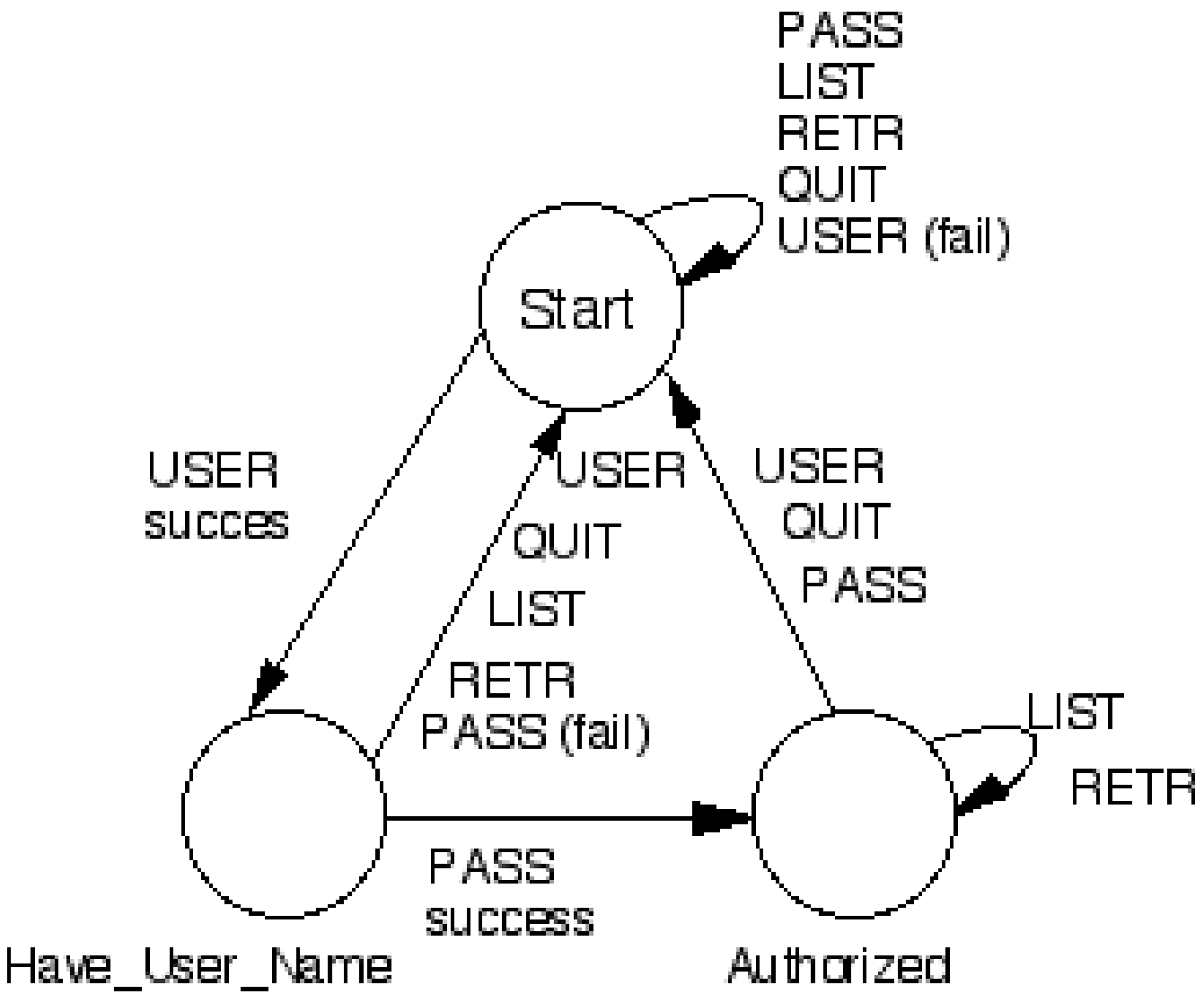
RETR command

- Arguments: a message number
- Returns: the mail message indicated by that number

QUIT command

- Arguments: none
- updates mailbox to reflect transactions taken during the transaction state, the logs user out
- if session ends by any method except the QUIT command, the updates are not done

SPOP STATES


PASS
LIST
RETR
QUIT
USER (fail)
Start
USER
succes
USER
QUIT
LIST
RETR
PASS (fail)
USER
QUIT
PASS
LIST
RETR
PASS
success
Have_User_Name
Authorized

THE "DEAR, OLD" SWITCHES IN ACTION

Think about adding a new state to the protocol...

Why?

- object's behavior depends on its state

```
class Spop {
   static final int HAVE_USER_NAME = 2;
   static final int START = 3;
   static final int AUTHORIZED = 4;

   private int state = START;

   String userName;
   String password;

   public void user( String userName ) {
      switch (state) {
         case START:
            this.userName = userName;
            state = HAVE_USER_NAME;
            break;

         case HAVE_USER_NAME:
         case AUTHORIZED:
            endLastSessionWithoutUpdate();
            goToStartState()
      }
   }



   public void pass( String password ) {
      switch (state) {
         case START:
            giveWarningOfIllegalCommand();
        break;
         case HAVE_USER_NAME:
            this.password = password;
            if (validateUser())
```

BASIC ASPECTS OF STATE PATTERN

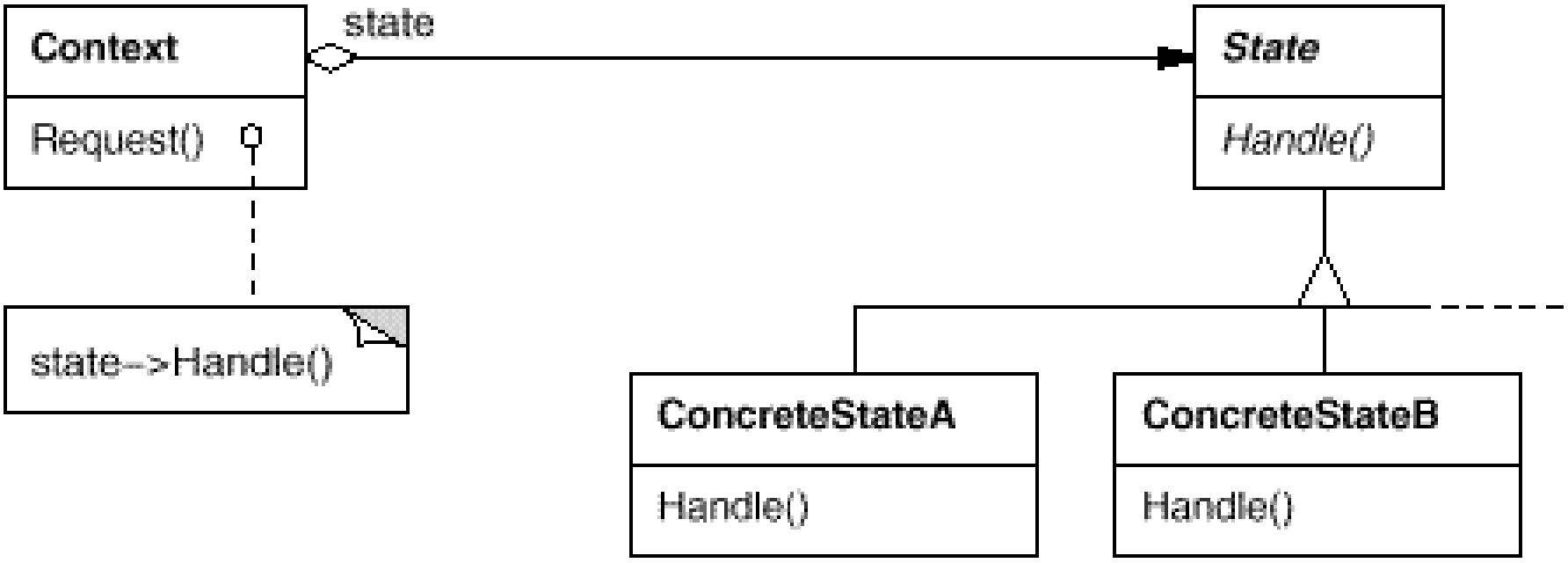
Intent

- allow an object to alter its behavior when its internal state changes
  - object will appear to change its class

Applicability

- object's behavior depends on its state
- it must change behavior at run-time depending on that state
- operations with multipart conditional statements depending on the object's state
  - state represented by one or more enumerated constants
  - several operations with the same (or similar) conditional structure

STRUCTURE


Context
Request()
state
State
Handle()
state->Handle()
ConcreteStateA
Handle()
ConcreteStateB
Handle()

PARTICIPANTS

Context

- defines the interface of interest for clients
- maintains an instance of ConcreteState subclass

State

- defines an interface for encapsulating the behavior associated with a particular state of the Context

ConcreteState

- each subclass implements a behavior associated with a state of the Context

COLLABORATIONS

Context delegates state-specific requests to the State objects

- the Context may pass itself to the State object
  - if the State needs to access it in order to accomplish the request

State transitions are managed either by Context or by State

- see discussion on the coming slides

Clients interact exclusively with Context

- but they might configure contexts with states
  - e.g initial state

CONSEQUENCES

Localizes state-specific behavior and **partitions** behavior for different states

- Put all behavior associated with a state in a state-object
- Easy to add new states and transitions
  - context becomes O-C
- Behavior spread among several State subclasses
  - number of classes increases, less compact than a single class
  - good if many states...

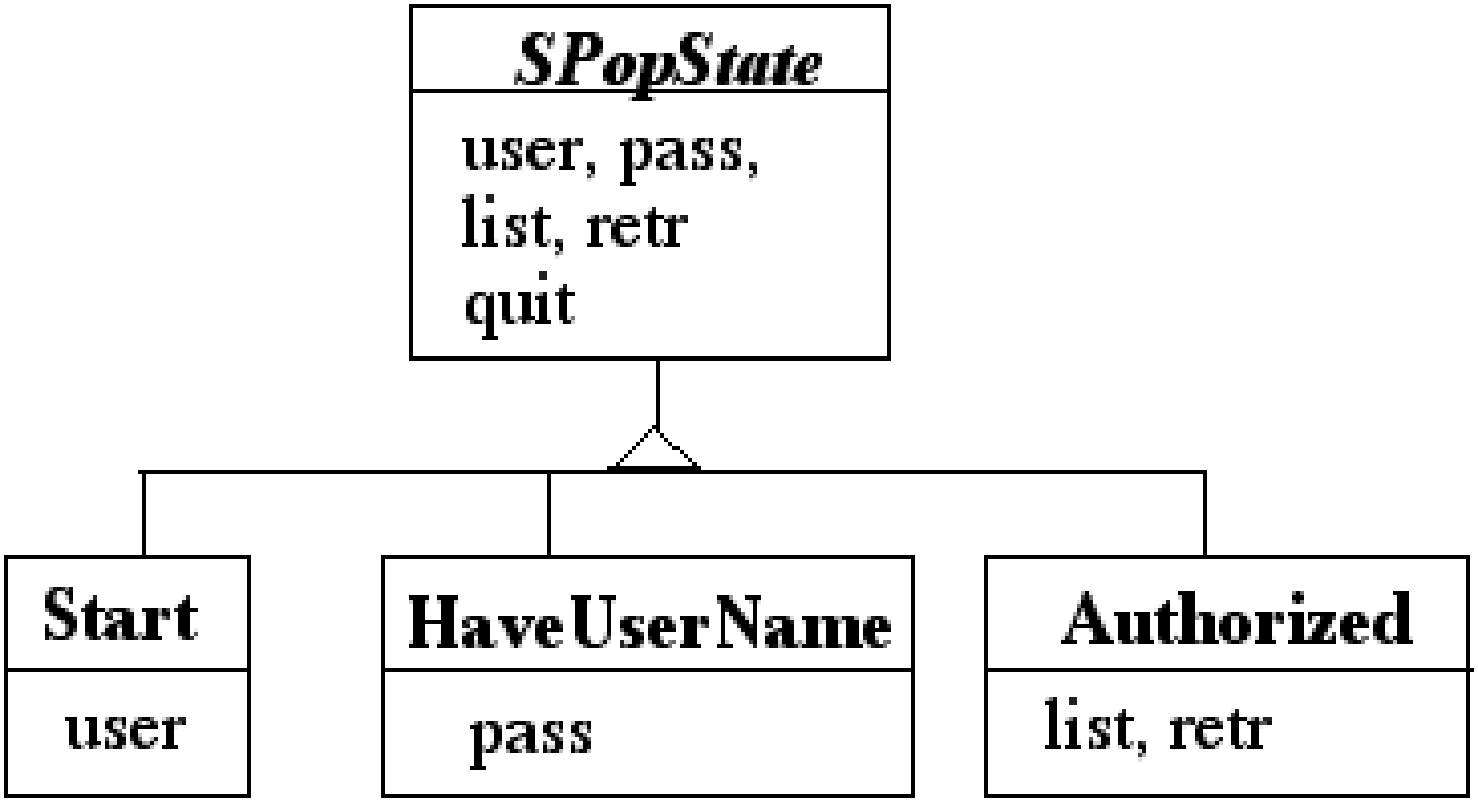
Makes state transitions explicit

- Not only a change of an internal value
- **States receive a full-object status!**
- Protects Context from inconsistent internal states

APPLYING STATE TO SPOP

```
class SPop {
      private SPopState state = new Start();
      public void user( String userName ) {
        state = state.user( userName );
      }
      public void pass( String password ) {
        state = state.pass( password );
      }
      public void list( int messageNumber ) {
        state = state.list( massageNumber );
      }
    //  .  . .
}
```

SPOP STATES


SPopState
user, pass,
list, retr
quit
Start
user
HaveUserName
pass
Authorized
list, retr

HOW MUCH STATE IN THE STATE?

Let's identify the roles...

- **`SPop`** is the Context
- **`SPopState`** is the abstract State
- **`Start, HaveUserName`** are ConcreteStates

All the state and *real* behavior is in SPopState and subclasses

- this is an extreme example

In general Context has data and methods

- besides State & State methods
- this data will not change states

Only some aspects of the Context will alter its behavior

WHO DEFINES THE STATE TRANSITION?

The Context if ...

- ...states will be **reused** in different state machines with different transitions
- ... the criteria for changing states are fixed

```
class Spop {
   private SPopState state = new Start();

   public void user( String userName ) {
      state.user( userName );
      state = new HaveUserName( userName );
   }

   public void pass( String password ) {
      if ( state.pass( password ) )
         state = new Authorized( );
      else
         state = new Start();
   }
```

OR…THE STATES

More flexible to let State subclasses specify the next state

```
class Spop {
   private SPopState state = new Start();

   public void user( String userName ) {
      state = state.user( userName );
   }

   public void pass( String password ) {
      state = state.pass( password );
   }

   public void list( int messageNumber ) {
      state = state.list( massageNumber );
   }

   // . . .
}

class Start extends SPopState {
   public SPopState user( String userName ) {
      return new HaveUserName( userName );
   }
}


class HaveUserName extends SPopState {
   String userName;

   public HaveUserName( String userName ) {
      this.userName = userName;
   }
```

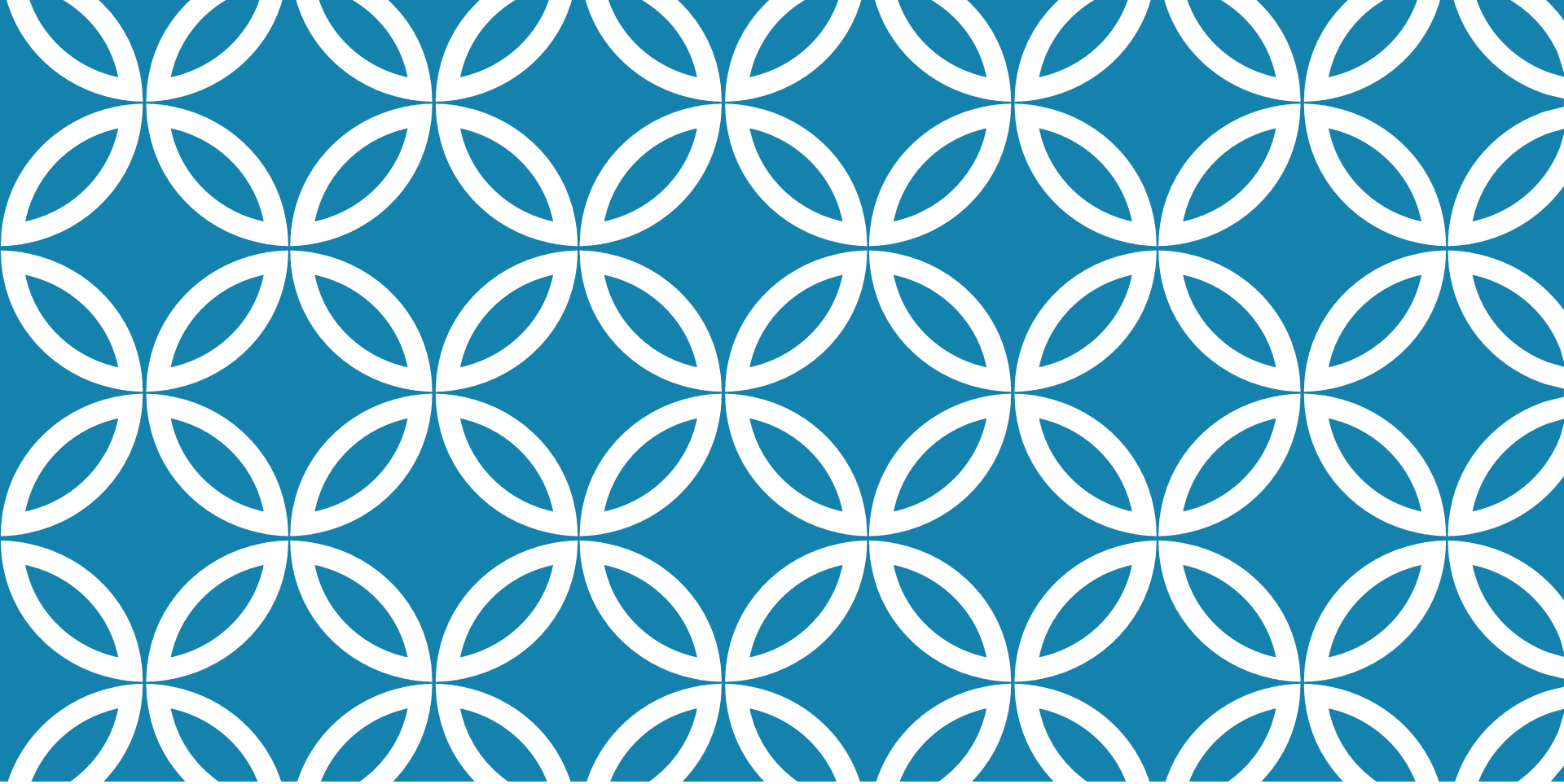
STATE VERSUS STRATEGY

Rate of Change

- Strategy
  - Context object usually contains one of several possible **`ConcreteStrategy`** objects
- State
  - Context object often changes its **`ConcreteState`** object over its lifetime

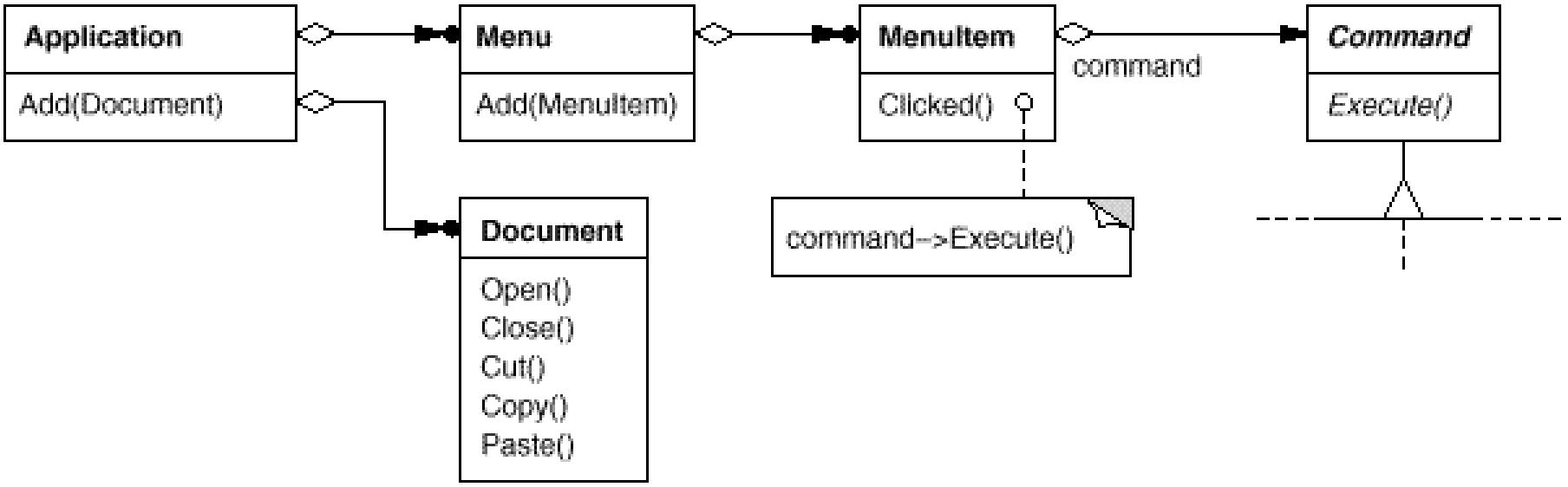
Visibility of Change

- Strategy
  - All **`ConcreteStrategy`** do the same thing, but differently
  - Clients do not see any difference in behavior in the Context
- State
  - **`ConcreteState`** acts differently
  - Clients see different behavior in the Context



COMMAND PATTERN

MENU ITEMS USE COMMANDS



BASIC ASPECTS

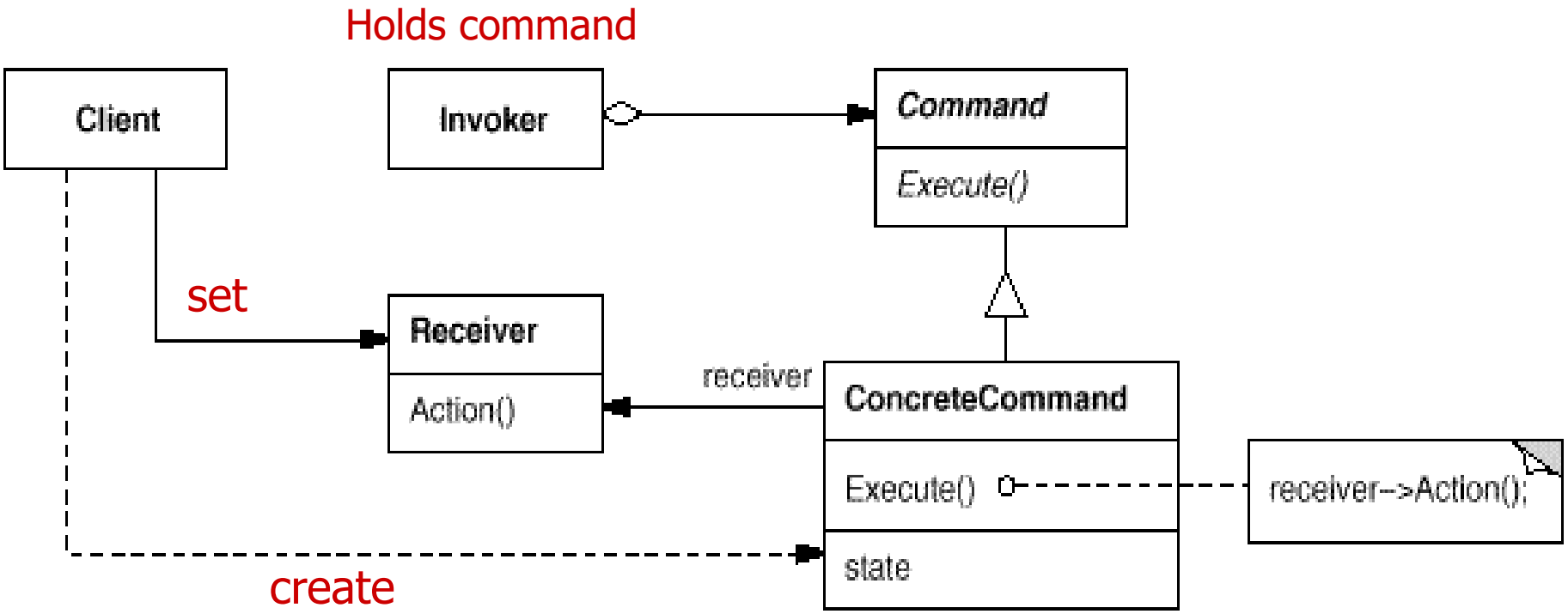
Intent

- Encapsulate requests as objects, letting you to:
  - parameterize clients with different requests
  - queue or log requests
  - support undoable operations

Applicability

- Parameterize objects
- Specify, queue, and execute requests at different times
- Support undo
  - recover from crashes → needs undo operations in interface
- Support for logging changes
  - recover from crashes → needs load/store operations in interface
- Model transactions

STRUCTURE



Transforms: *concreteReceiver*.**action()** in *command*.**execute()**

PARTICIPANTS

Command

- declares the interface for executing the operation

ConcreteCommand

- binds a request with a concrete action

Invoker

- asks the command to carry out the request

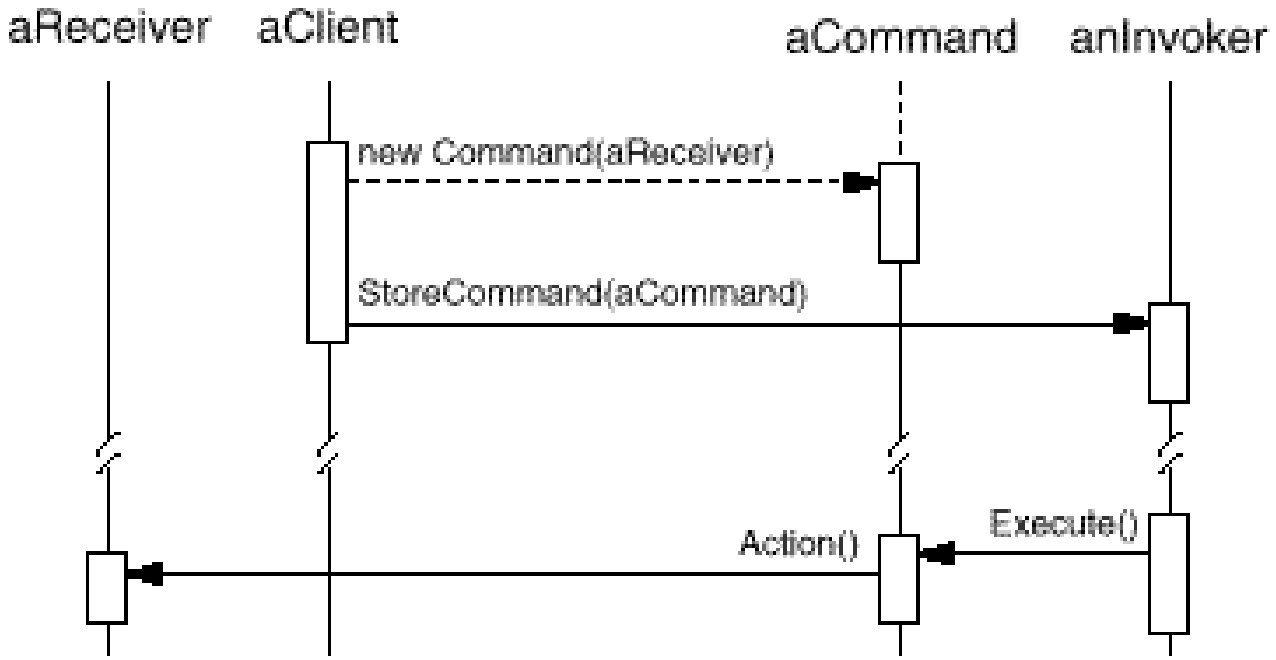
Receiver

- knows how to perform the operations associated with carrying out a request.

Client

- creates a ConcreteCommand and sets its receiver

COLLABORATIONS



Client → ConcreteCommand

- creates and specifies receiver

Invoker → ConcreteCommand

ConcreteCommand → Receiver

CONSEQUENCES

Decouples Invoker from Receiver

Commands are **first-class objects**

- can be manipulated and **extended**

Composite Commands

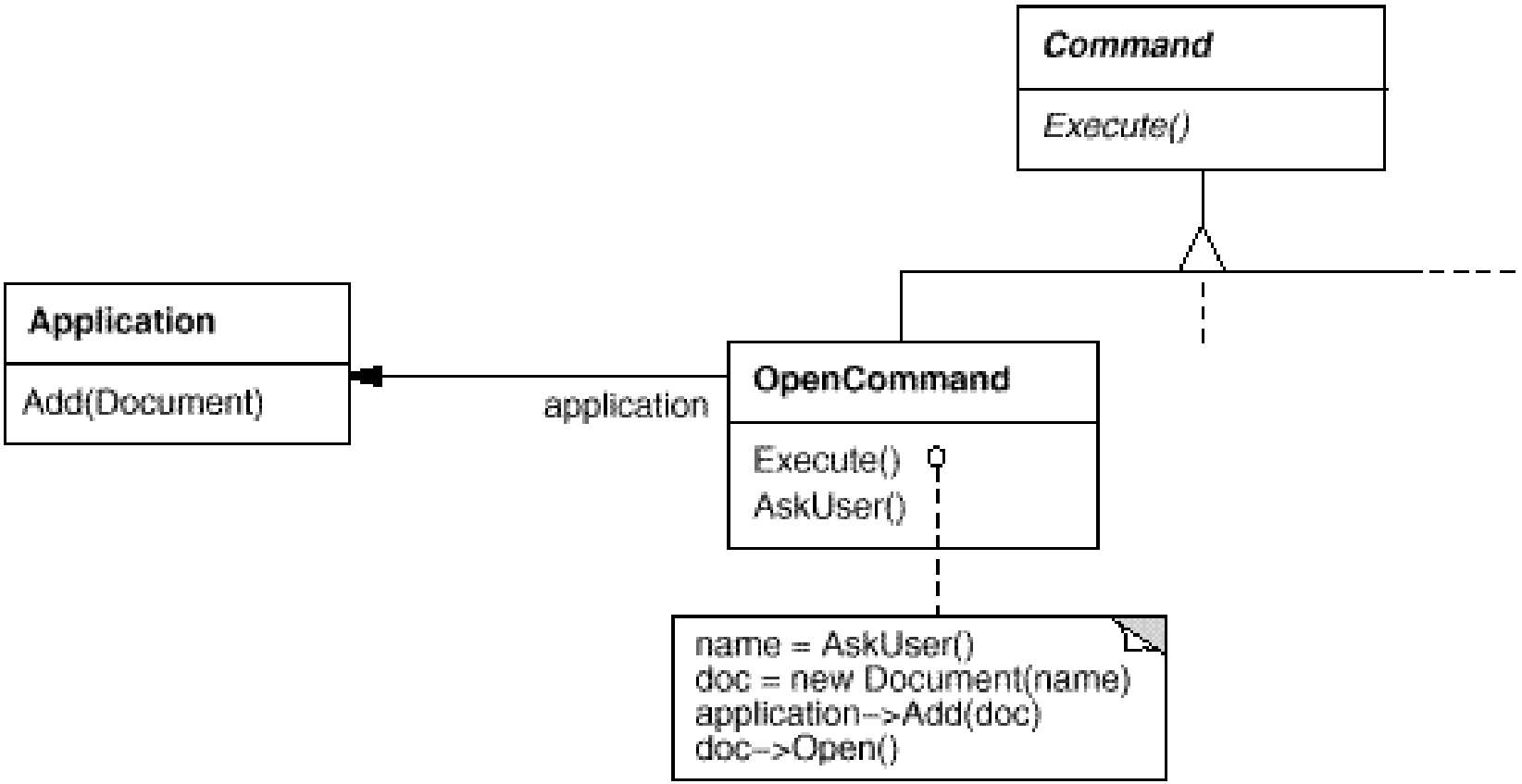
- see also C*omposit*e pattern

Easy to add new commands

- Invoker does not change
- it is Open-Closed

Potential for an excessive number of command classes

EXAMPLE: OPEN DOCUMENT



"INTELLIGENCE" OF COMMAND OBJECTS

"Dumb"

- delegate everything to Receiver
- used just to decouple Sender from Receiver

"Genius"

- does everything itself without delegating at all
- useful if no receiver exists
- let ConcreteCommand be independent of further classes

"Smart"

- find receiver dynamically

UNDOABLE COMMANDS

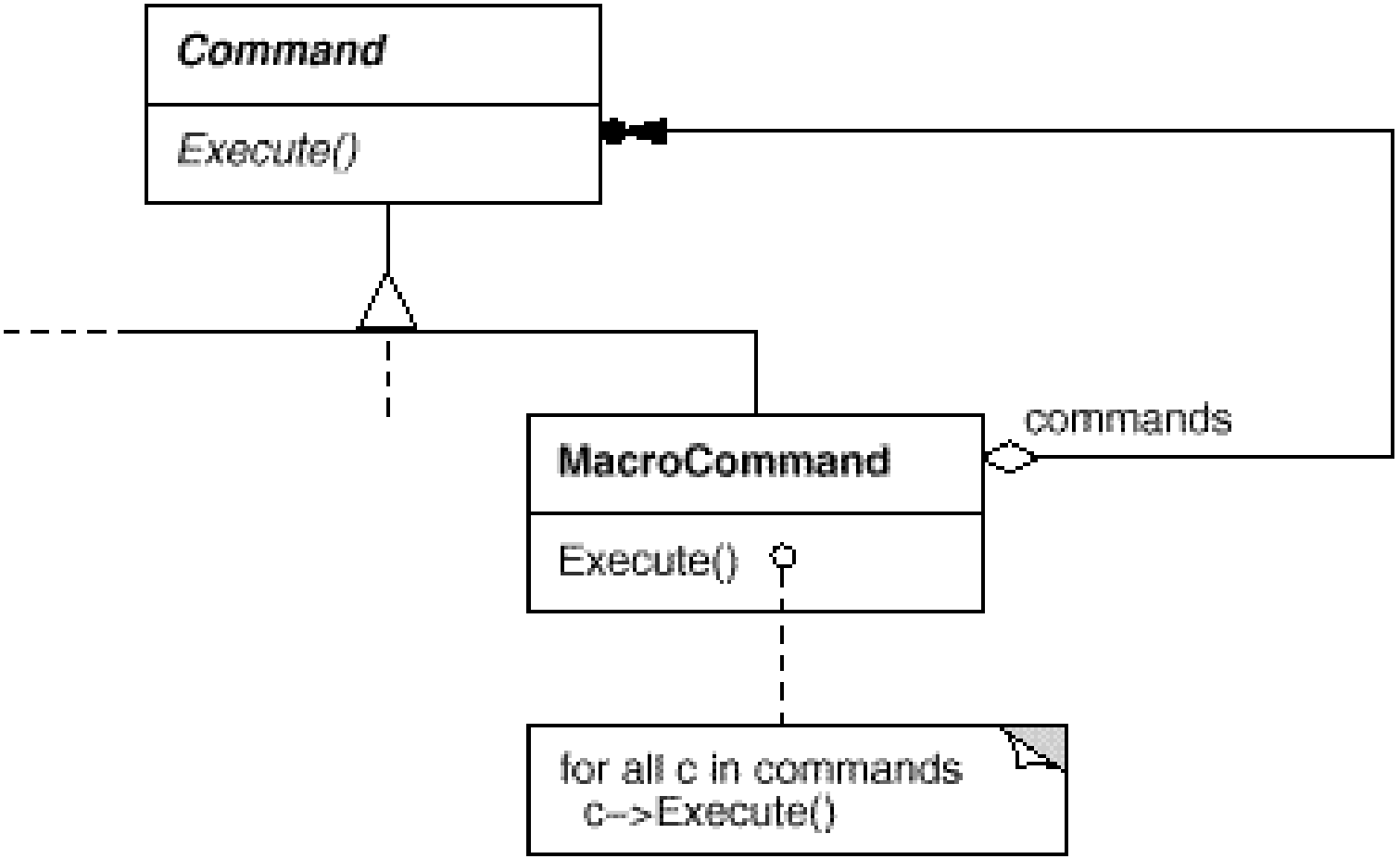
Need to store additional state to reverse execution

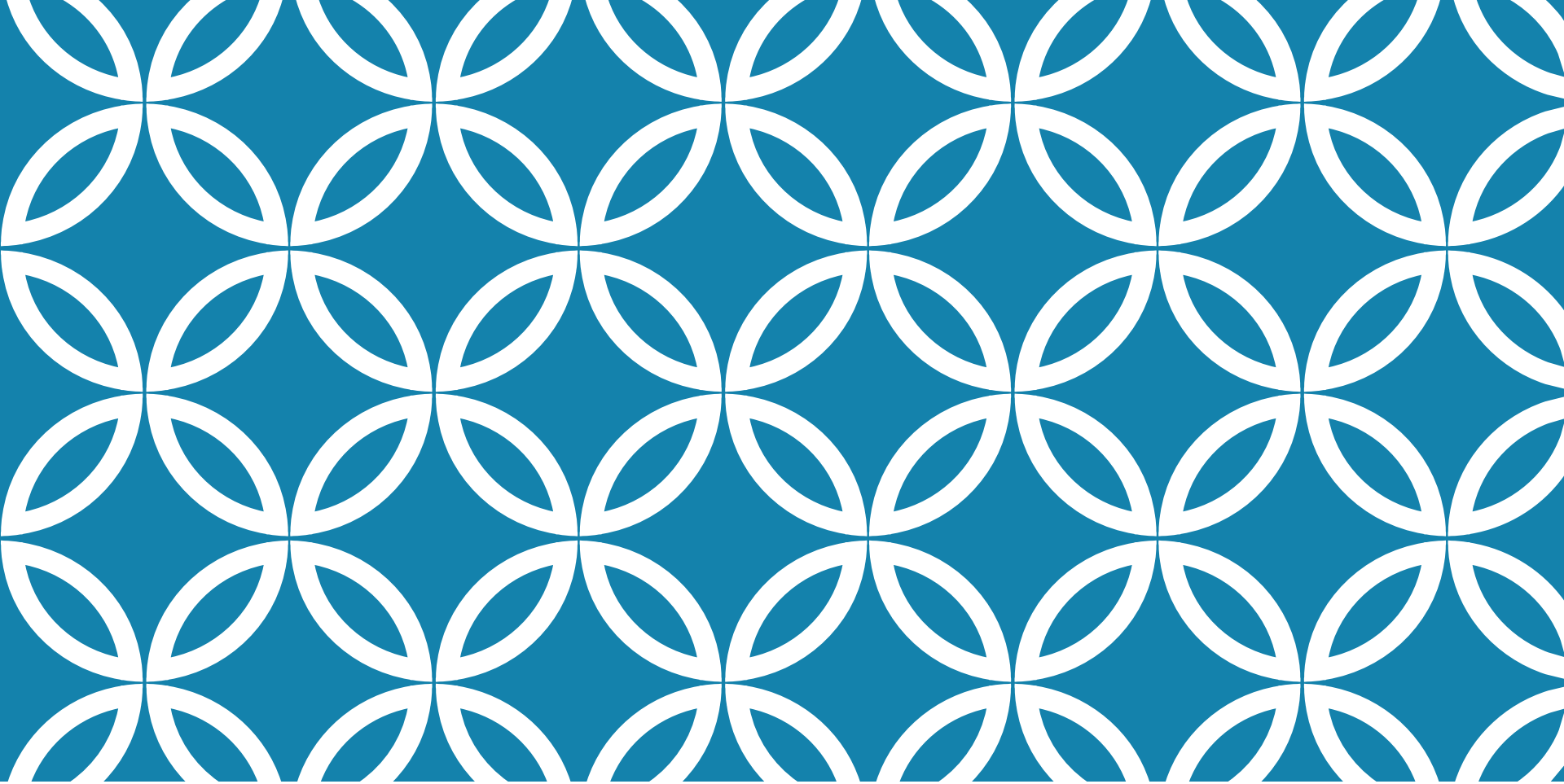
- receiver object
- parameters of the operation performed on receiver
- original values in receiver that may change due to request
  - receiver must provide operations that makes possible for command object to return it to its prior state

History list

- Sequence of commands that have been executed
  - used as LIFO with reverse-execution $\Rightarrow$ undo
  - used as FIFO with execution $\Rightarrow$ redo
- Commands may need to be copied
  - when state of commands change by execution

COMPOSED COMMANDS





CHAIN OF RESPONSIBILITY PATTERN

BASIC ASPECTS

Intent

- Decouple sender of request from its receiver
  - by giving more than one object a chance to handle the request
- Put receivers in a chain and pass the request along the chain
  - until an object handles it

Motivation

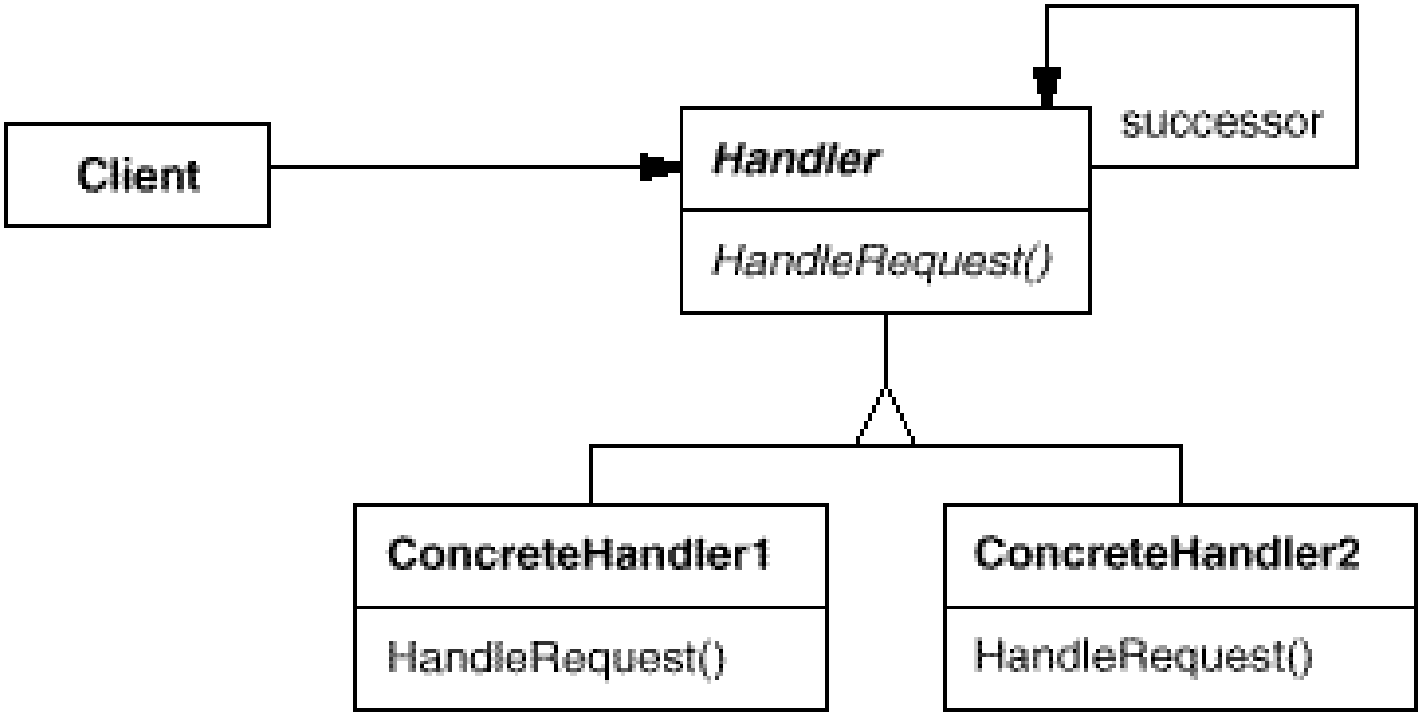
- context-sensitive help
  - a help request is handled by one of several UI objects
- Which one?
  - depends on the context
- The object that initiates the request does not know the object that will eventually provide the help

WHEN TO USE?

Applicability

- more than one object may handle a request
  - and handler isn't known a priori
- set of objects that can handle the request should be dynamically specifiable
- send a request to several objects without specifying the receiver

STRUCTURE


Client
Handler
HandleRequest()
successor
ConcreteHandler1
HandleRequest()
ConcreteHandler2
HandleRequest()


aClient
aHandler
aConcreteHandler
successor
aConcreteHandler
successor

PARTICIPANTS & COLLABORATIONS

Handler

- defines the interface for handling requests
- may implement the successor link

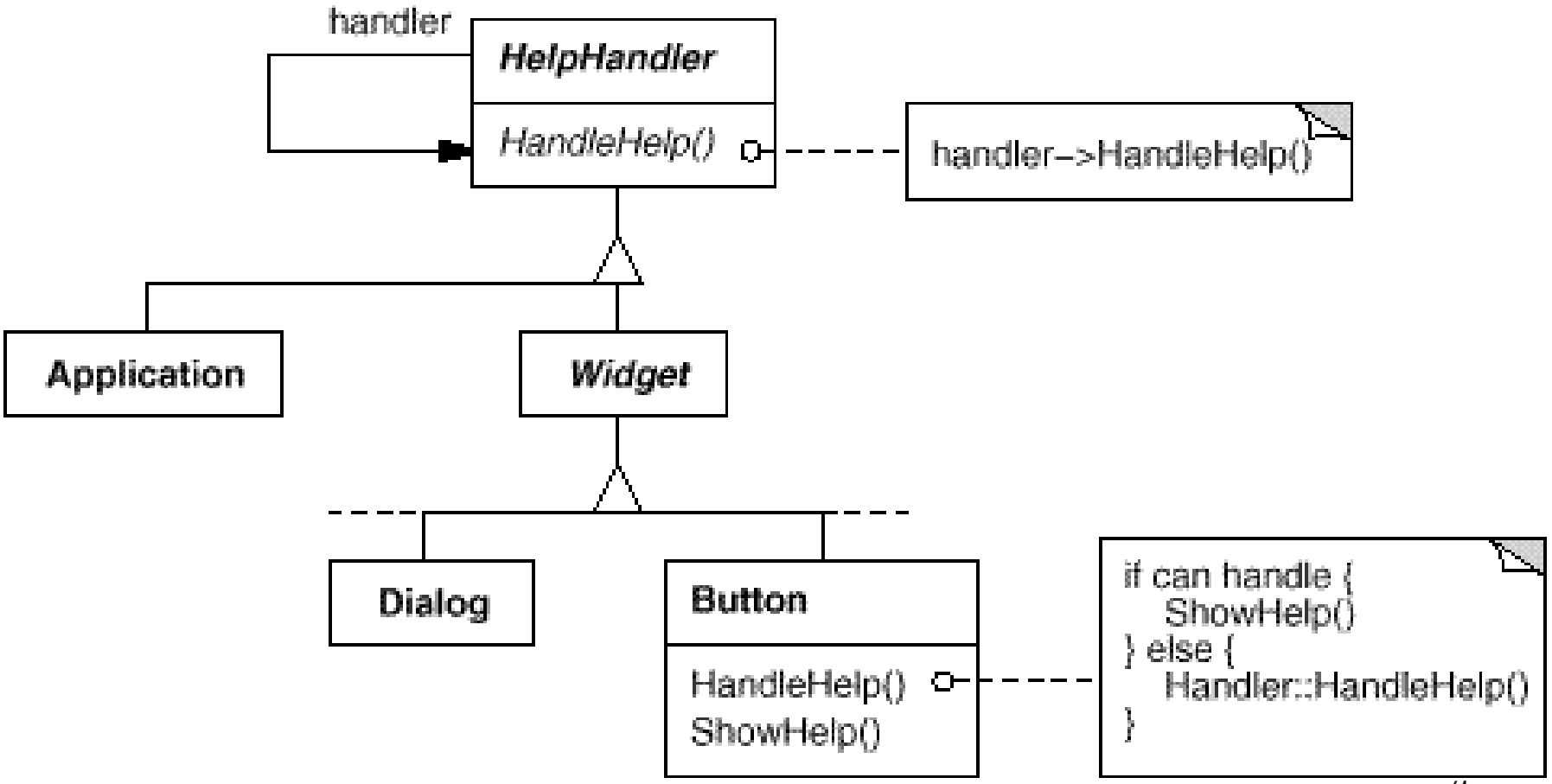
ConcreteHandler

- either handles the request it is responsible for ...
- ... or it forwards the request to its successor

Client

- initiates the request to a ConcreteHandler object in the chain

THE CONTEXT-HELP SYSTEM



CONSEQUENCES

Reduced Coupling

- frees the client (sender) from knowing who will handle its request
- sender and receiver don't know each other
- instead of sender knowing all potential receivers, just keep a single reference to next handler in chain.
  - simplify object interconnections

Flexibility in assigning responsibilities to objects

- responsibilities can be added or changed
- chain can be modified at run-time

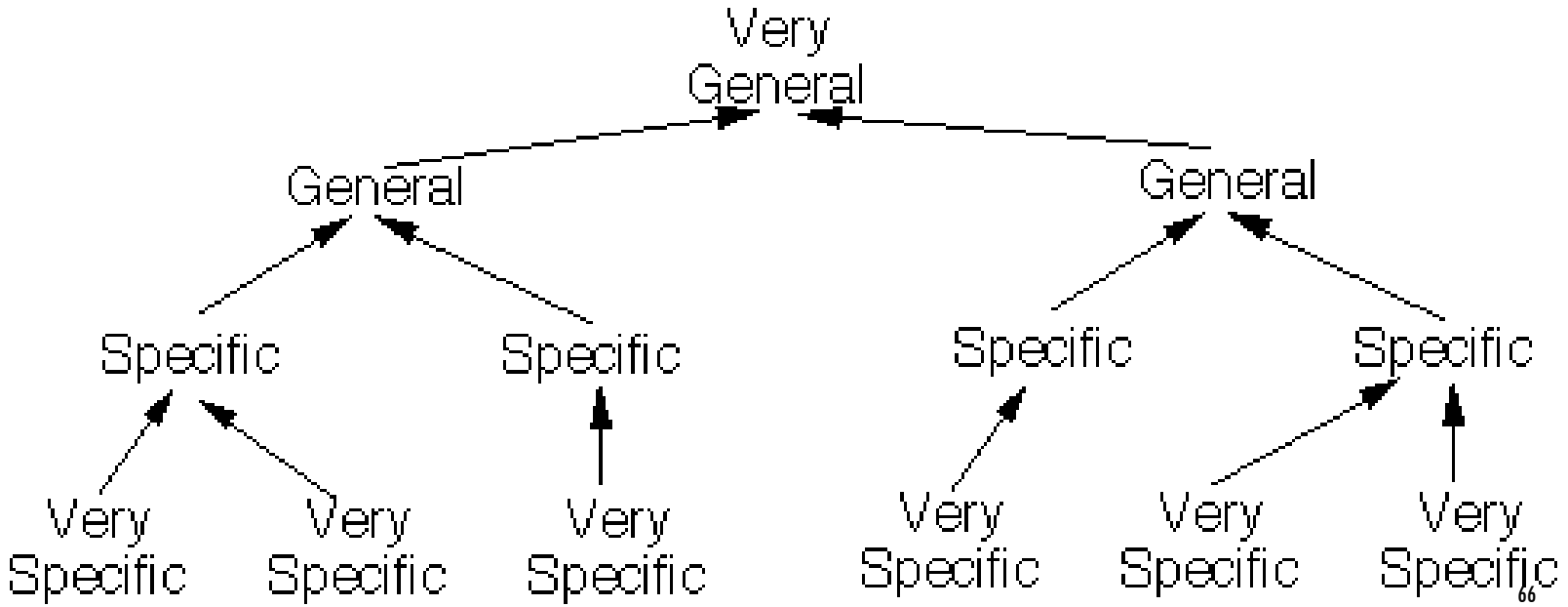
Requests can go unhandled

- chain may be configured improperly!!

HOW TO DESIGN CHAINS OF COMMANDS?

Like the military

- a request is made
- it goes up the chain of command until someone has the authority to answer the request



IMPLEMENTING THE SUCCESSOR CHAIN

Define new link

- Give each handler a link to its successor

Use existing links

- concrete handlers may already have pointers to their successors
- references in a part-whole hierarchy
  - can define a part's successor
- spares work and space ...
- ... but it must reflect the chain of responsibilities that is needed

REPRESENTING MULTIPLE REQUESTS USING ONE CHAIN

Each request is hard-coded

- convenient and safe
- not flexible (limited to the fixed set of requests defined by handler)

```
abstract class HardCodedHandler {
   private HardCodedHandler successor;

   public HardCodedHandler( HardCodedHandler aSuccessor)
      { successor = aSuccessor; }

   public void handleOpen()
      { successor.handleOpen(); }

   public void handleClose()
      { successor.handleClose(); }

   public void handleNew( String fileName)
      { successor.handleNew( fileName ); }
}
```

Unique handler with parameters

- more flexible
- but it requires conditional statements for dispatching request
- less type-safe to pass parameters

```
abstract class SingleHandler {
   private SingleHandler successor;

   public SingleHandler( SingleHandler aSuccessor) {
      successor = aSuccessor;
   }

   public void handle( String request) {
      successor.handle( request );
   }
}

class ConcreteOpenHandler extends SingleHandler {
   public void handle( String request) {
      switch ( request ) {
         case "Open" : // do the right thing;
         case "Close" : // more right things;
         case "New" : // even more right things;
         default: successor.handle( request );
      }
   }
}
```

DECORATOR VS. CHAIN OF RESPONSIBILITY

Chain of Responsibility	Decorator
Comparable to “event-oriented” architecture	Comparable to layered architecture (layers of an onion)
The "filter" objects are of equal rank	A "core" object is assumed, all "layer" objects are optional
User views the chain as a "launch and leave" pipeline	User views the decorated object as an enhanced object
A request is routinely forwarded until a single filter object handles it. many (or all) filter objects *could* contrib. to each request's handling.	A layer object always performs pre or post processing as the request is delegated.
All the handlers are peers (like nodes in a linked list) – "end of list" condition handling is required.	All the layer objects ultimately delegate to a single core object - "end of list" condition handling is not required.

NEXT TIME

Quality attributes

- Representation
- Strategies