# Software Engineering I
## Object-Oriented Design

Software Design Refinement

Using Design Patterns

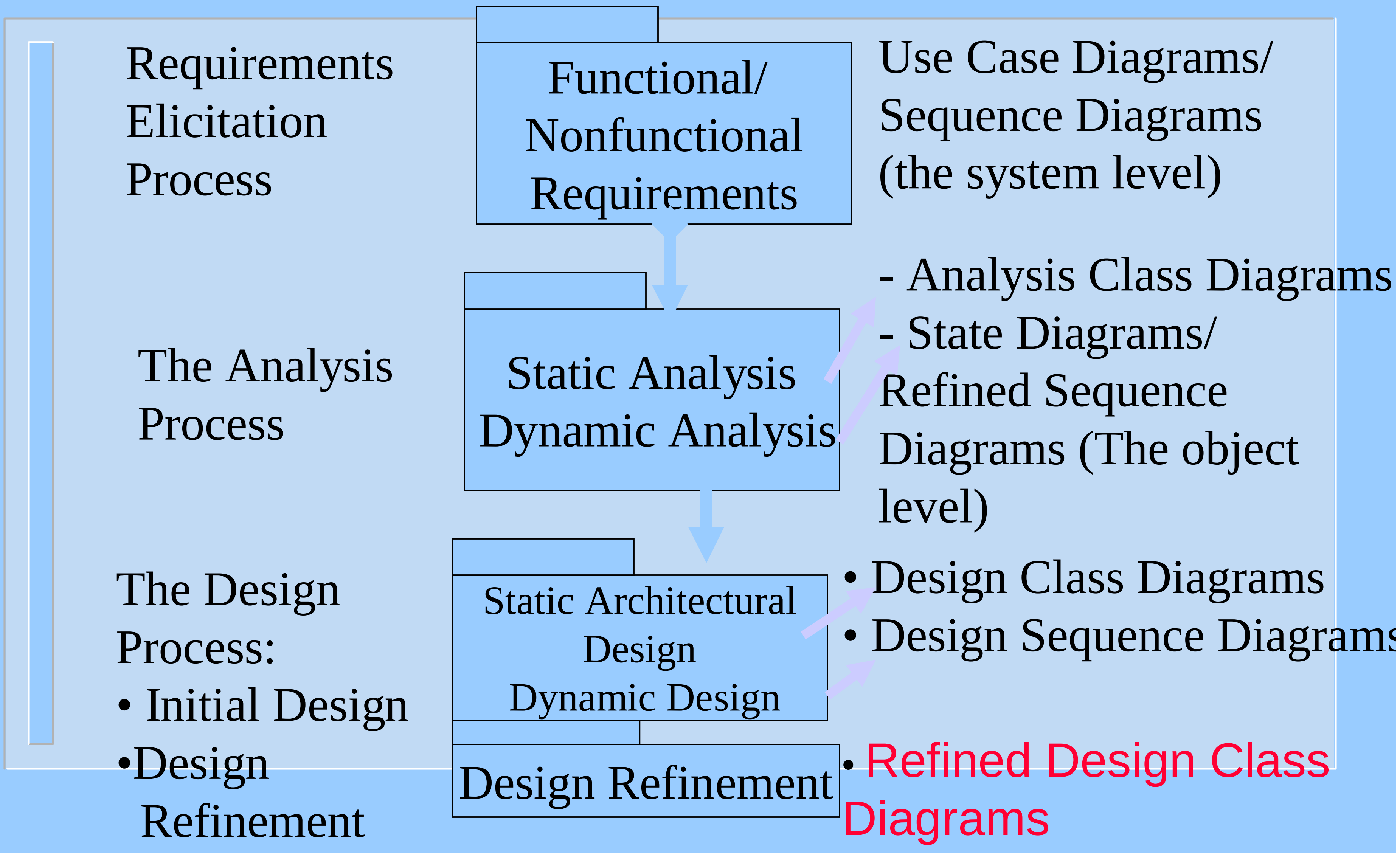
Instructor: Dr. Hany H. Ammar

Dept. of Computer Science and
Electrical Engineering, WVU

# Outline

- The Requirements, Analysis, Design, and Design Refinement Models
- Design refinement
- Class diagram refinement using design patterns
- Design patterns examples
  - The Facade pattern
  - The Strategy Pattern
  - The State Pattern
  - The Command Pattern
  - The Observer Pattern
  - The Proxy Pattern
- Design Patterns Tutorials

# The Requirements, Analysis, Design, and Desgin Refiement Models



Requirements Elicitation Process

Functional/ Nonfunctional Requirements

Use Case Diagrams/ Sequence Diagrams (the system level)

The Analysis Process

Static Analysis
Dynamic Analysis

- Analysis Class Diagrams
- State Diagrams/ Refined Sequence Diagrams (The object level)

The Design Process:

- Initial Design
- Design Refinement

Static Architectural Design
Dynamic Design

Design Refinement

- Design Class Diagrams
- Design Sequence Diagrams

- Refined Design Class Diagrams

# Design Refinement

- It is difficult to obtain a quality design from the initial design
- The initial design is refined to enhance design quality using the software design criteria of modularity, information hiding, complexity, testability, and reusability.
- New components (or new classes) are defined and existing components (or classes) structures are refined to enhance design quality
- The design refinement step is an essential step before implementation and testing.

# Class Diagram Refinement Using Design Patterns

- Design Class Diagrams are further refined to enhance design quality (i.e., reduce coupling, increase cohesion, and reduce component complexity) **using design patterns**
- A design pattern is a documented good design solution of a design problem
- Repositories of design patterns were developed for many application domains (communication software, agent-based systems, web applications)
- Many generic design patterns were defined and can be used to enhance the design of systems in different application domains

# What is a Design Pattern

- What is a Design Pattern?

  A design pattern describes a design problem which repeatedly occurred in previous designs, and then describes the core of the solution to that problem
- Solutions are expressed in terms of classes of objects and interfaces (object-oriented design patterns)
- A design pattern names, abstracts, and identifies the key aspects of a high quality design structure that make it useful for creating reusable object-oriented designs

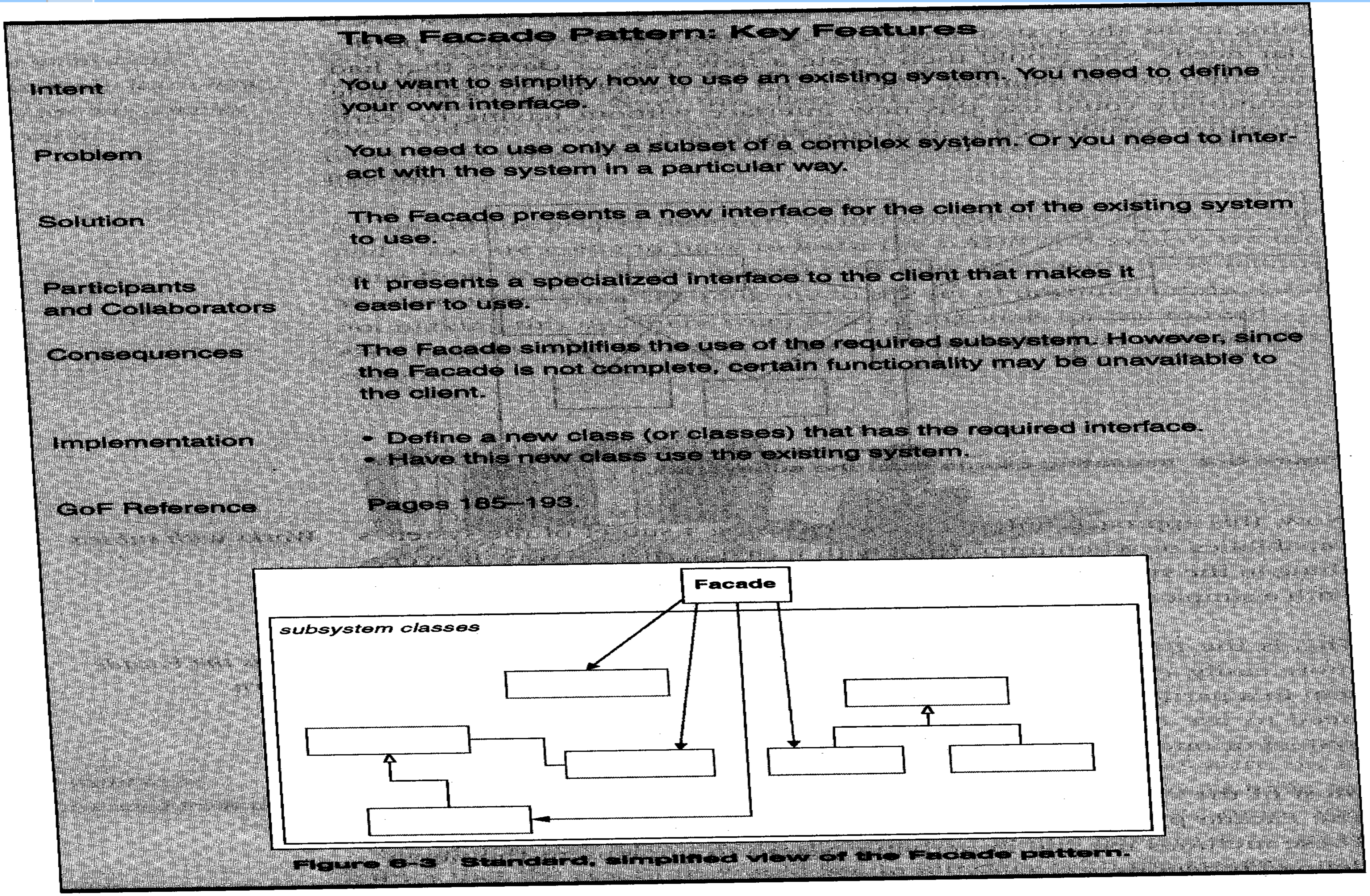
# Defining a Design Pattern

- Design Patterns are documented in the literature by a template consisting of the following

A Design Pattern has 5 basic parts:

1. Name
2. Problem
3. Solution
4. Consequences and trade-of of application
5. Implementation: An architecture using a design class diagram

# Example of Pattern Definition: The  Façade Pattern Provides An Interface To a Subsystem



## The Facade Pattern: Key Features

| | |
|---|---|
| Intent | You want to simplify how to use an existing system. You need to define your own interface. |
| Problem | You need to use only a subset of a complex system. Or you need to interact with the system in a particular way. |
| Solution | The Facade presents a new interface for the client of the existing system to use. |
| Participants and Collaborators | It presents a specialized interface to the client that makes it easier to use. |
| Consequences | The Facade simplifies the use of the required subsystem. However, since the Facade is not complete, certain functionality may be unavailable to the client. |
| Implementation | • Define a new class (or classes) that has the required interface.<br>• Have this new class use the existing system. |
| GoF Reference | Pages 185–193. |

Figure 6-3  Standard, simplified view of the Facade pattern.

# The Facade Pattern

- The class Facade is introduced as an interface to the whole subsystem.
- Any client class needs a service from any of the subsystem classes will be send the request to the facade class.
- All the subsystem interfaces are combined in one class

# Example of Using the Design Pattern

- Design Patterns produce quality designs by reducing coupling
- Example of how a Façade Pattern reduces coupling

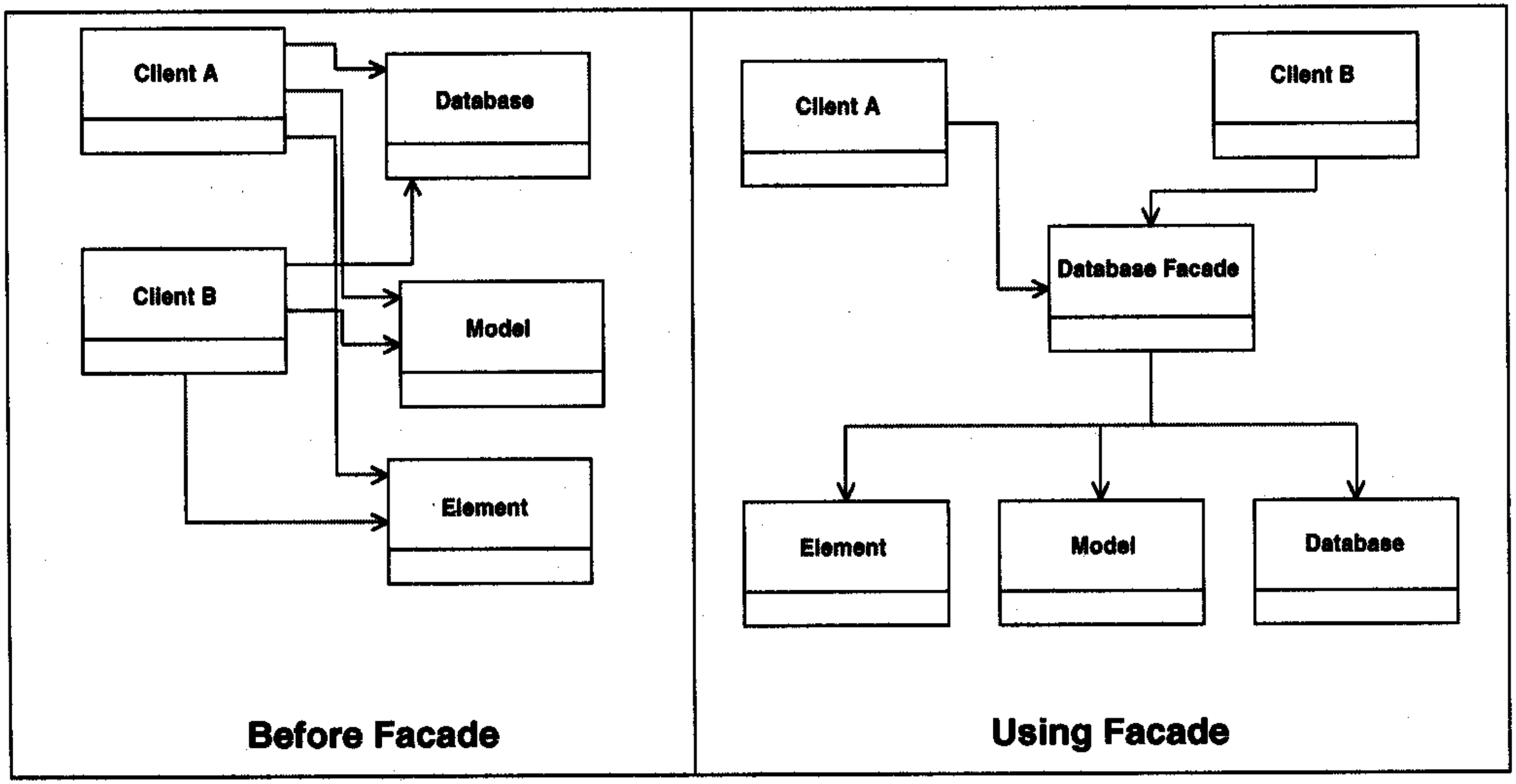
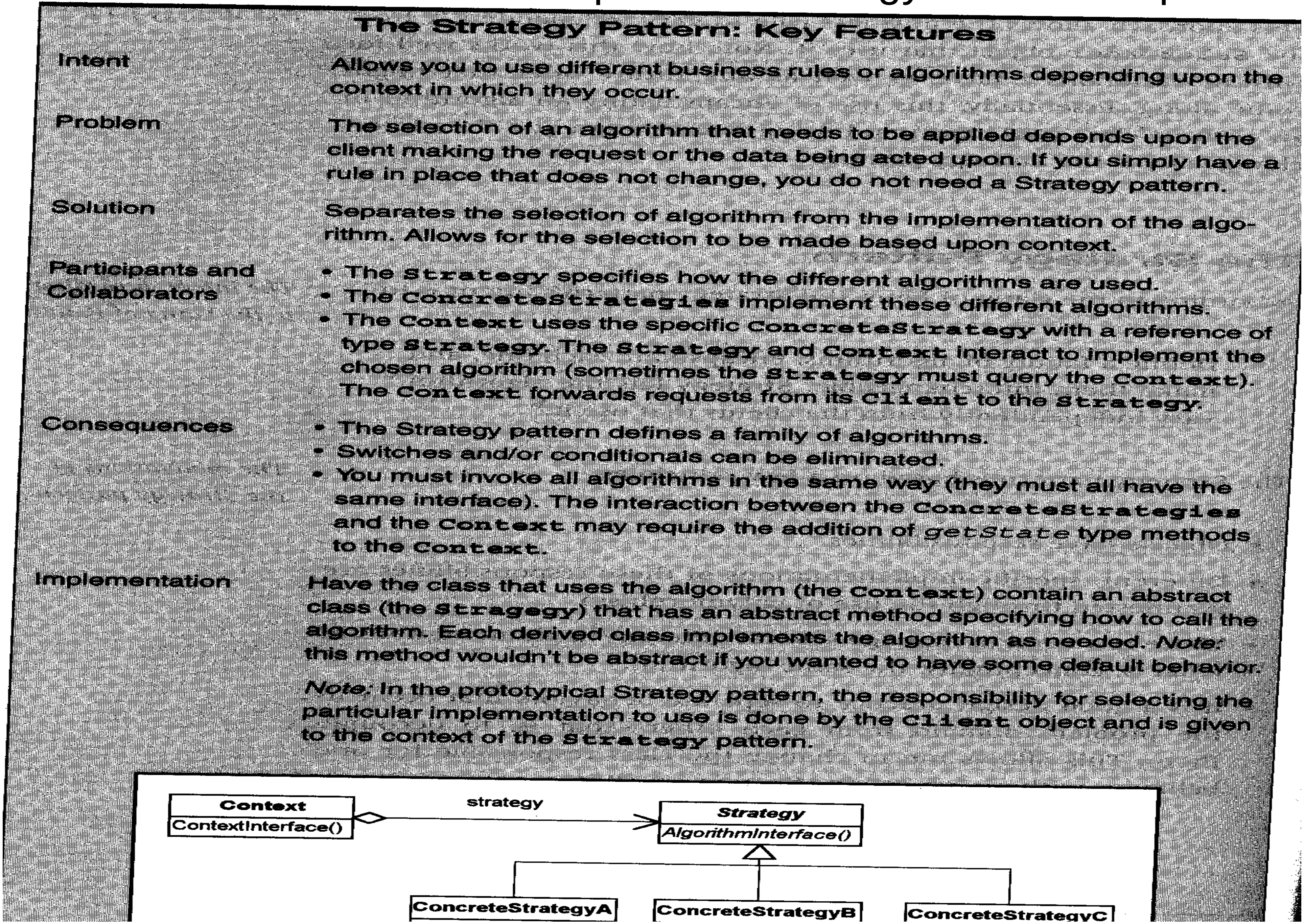


**Figure 6-4 Facade reduces the number of objects for the client.**

# Another Example: The Strategy Pattern template



## The Strategy Pattern: Key Features

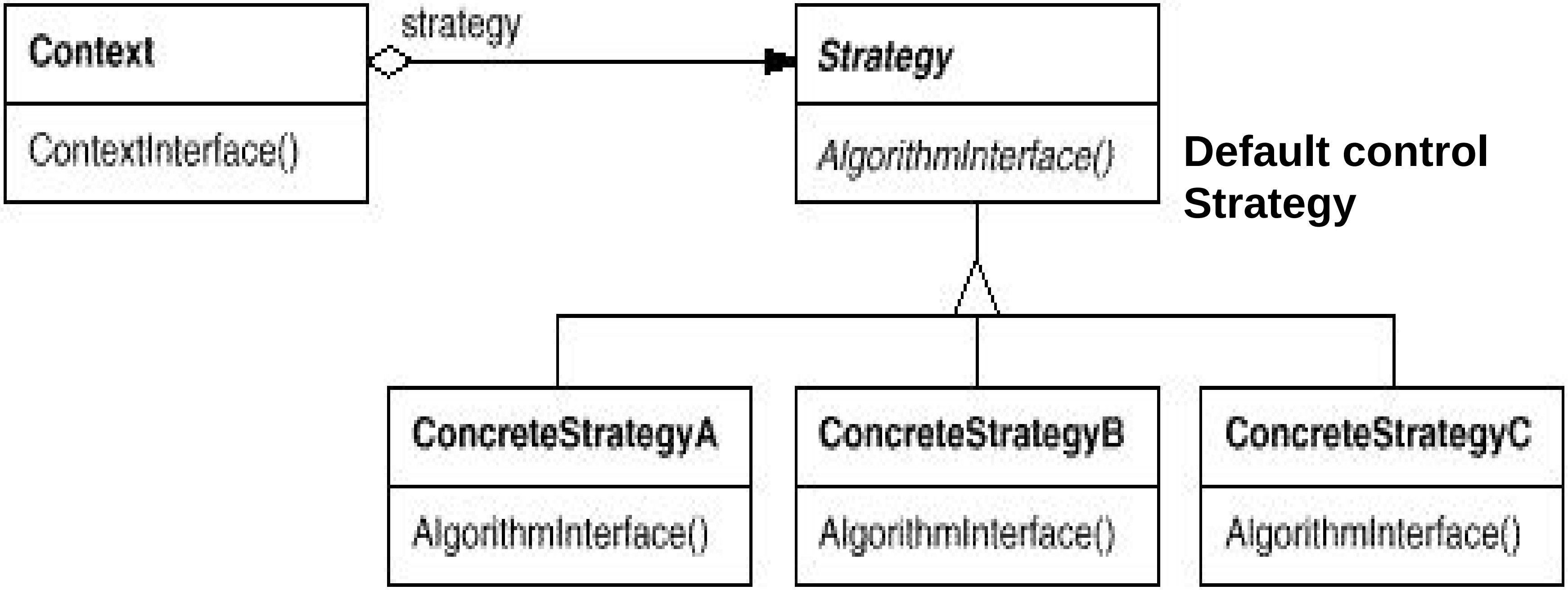
| | |
|---|---|
| **Intent** | Allows you to use different business rules or algorithms depending upon the context in which they occur. |
| **Problem** | The selection of an algorithm that needs to be applied depends upon the client making the request or the data being acted upon. If you simply have a rule in place that does not change, you do not need a Strategy pattern. |
| **Solution** | Separates the selection of algorithm from the implementation of the algorithm. Allows for the selection to be made based upon context. |
| **Participants and Collaborators** | • The `Strategy` specifies how the different algorithms are used.<br>• The `ConcreteStrategies` implement these different algorithms.<br>• The `Context` uses the specific `ConcreteStrategy` with a reference of type `Strategy`. The `Strategy` and `Context` interact to implement the chosen algorithm (sometimes the `Strategy` must query the `Context`). The `Context` forwards requests from its `Client` to the `Strategy`. |
| **Consequences** | • The Strategy pattern defines a family of algorithms.<br>• Switches and/or conditionals can be eliminated.<br>• You must invoke all algorithms in the same way (they must all have the same interface). The interaction between the `ConcreteStrategies` and the `Context` may require the addition of *`getState`* type methods to the `Context`. |
| **Implementation** | Have the class that uses the algorithm (the `Context`) contain an abstract class (the `Stragegy`) that has an abstract method specifying how to call the algorithm. Each derived class implements the algorithm as needed. *Note:* this method wouldn't be abstract if you wanted to have some default behavior.<br>*Note:* In the prototypical Strategy pattern, the responsibility for selecting the particular implementation to use is done by the `Client` object and is given to the context of the `Strategy` pattern. |

# Another Example of Design Patterns

- The Strategy Pattern: lets the algorithm vary independently from clients that use it

Controller Class

Abstract Class



Default control Strategy

Control Strategy A

Control Strategy B

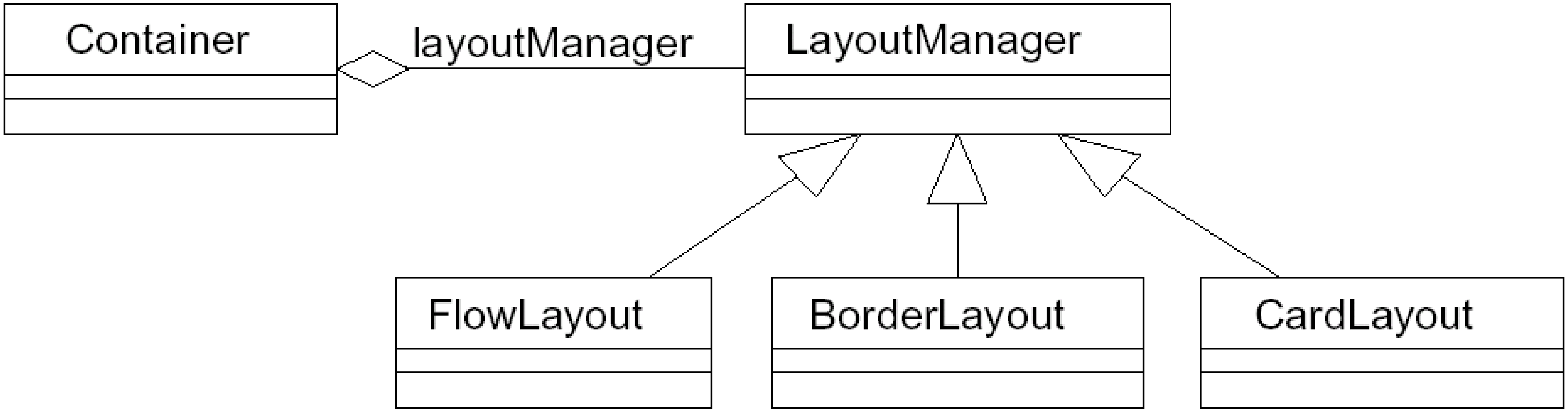
Control Strategy C

# The Strategy Pattern

- The Strategy Pattern Context class has multiple control strategies provided by the concrete strategy classes, or by the abstract strategy (by default)
- The pattern lets us vary the algorithm that implements a certain function during run time depending on the conditions of the system
- The Pattern reduces coupling by having the client class be coupled only to the context class

# Examples of Design Patterns
# The Strategy Pattern

- Example of using the pattern in JAVA AWT GUI components lay out managers



# Examples of Design Patterns
# The Strategy Pattern: another example

- Situation: A GUI text component object wants to decide at runtime what strategy it should use to validate user input. Many different validation strategies are possible: numeric fields, alphanumeric fields, telephone-number fields, etc.

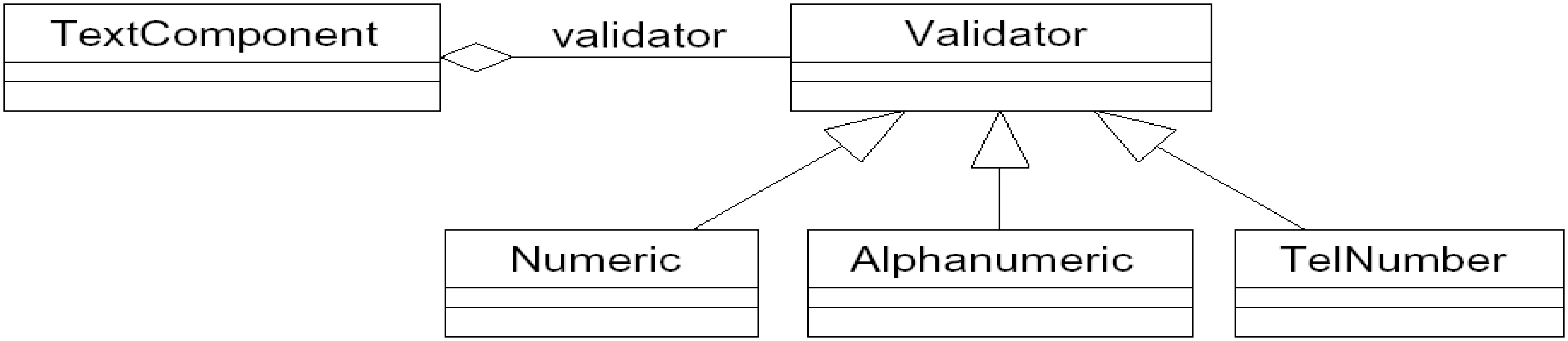
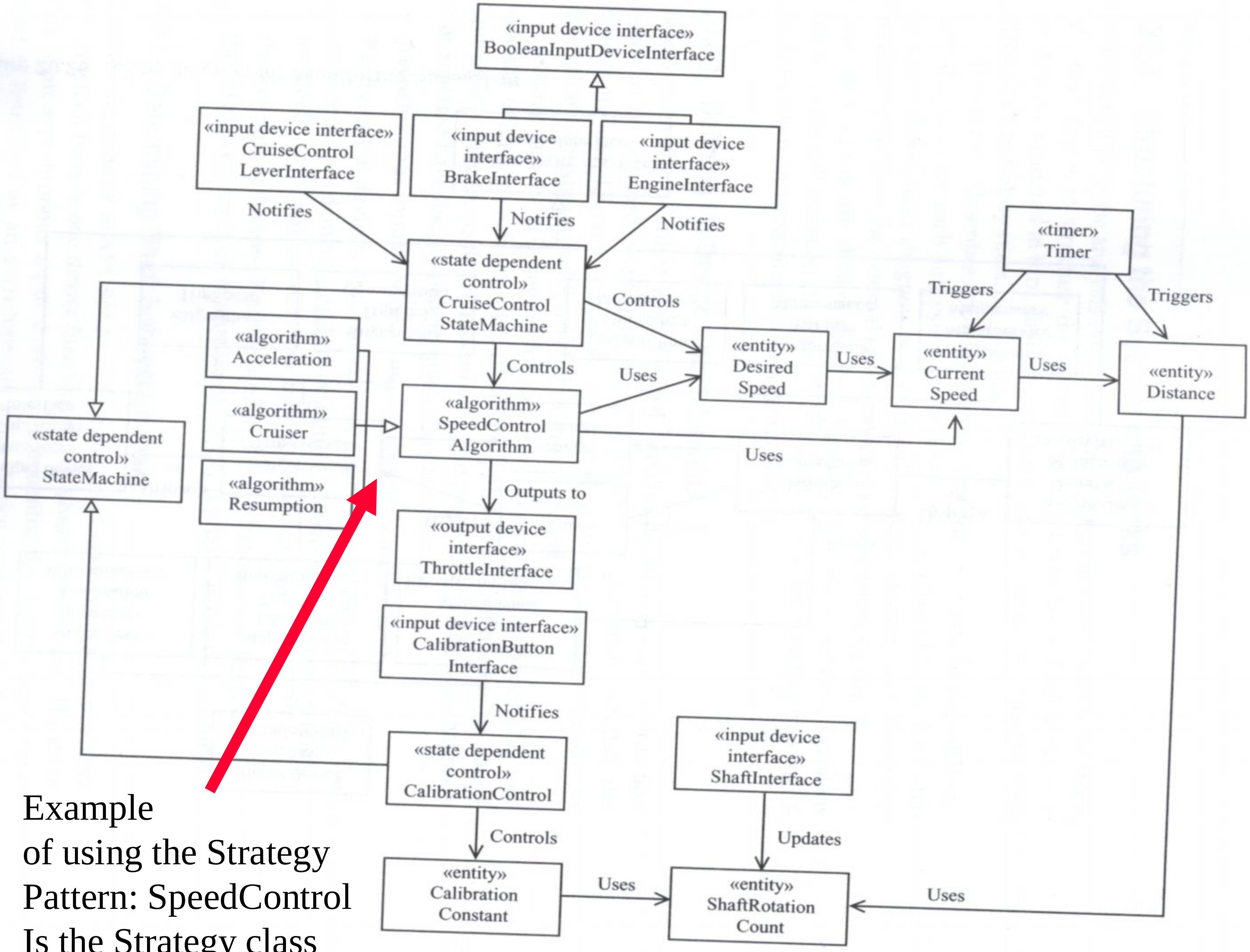


**Figure 20.25** *Class diagram for Cruise Control Subsystem*

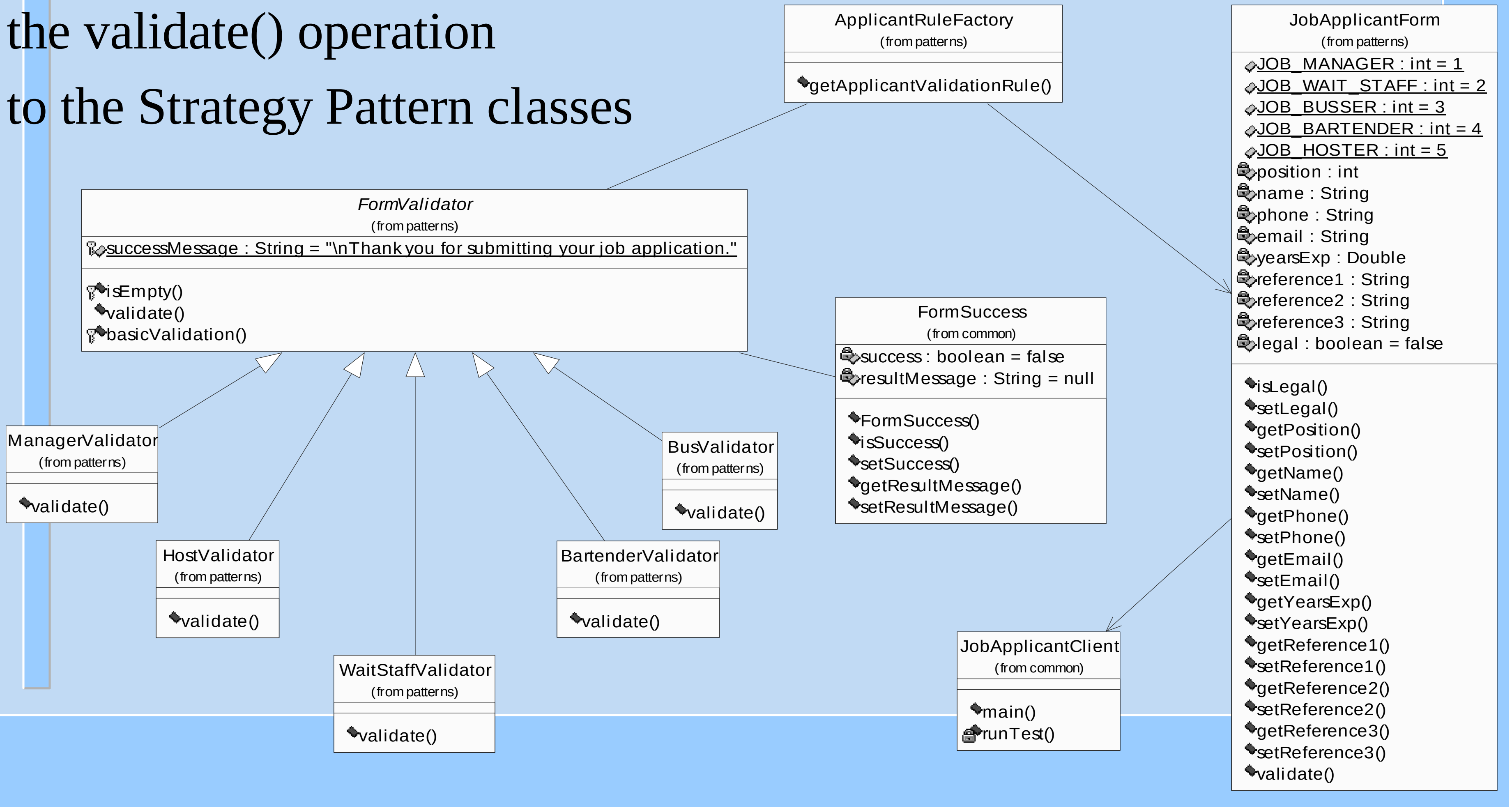

«input device interface» BooleanInputDeviceInterface
«input device interface» CruiseControl LeverInterface
«input device interface» BrakeInterface
«input device interface» EngineInterface
Notifies
Notifies
Notifies
«timer» Timer
«state dependent control» CruiseControl StateMachine
Triggers
Triggers
Controls
«algorithm» Acceleration
Controls
Uses
«entity» Desired Speed
Uses
«entity» Current Speed
Uses
«entity» Distance
«algorithm» Cruiser
«algorithm» SpeedControl Algorithm
«state dependent control» StateMachine
Uses
«algorithm» Resumption
Outputs to
«output device interface» ThrottleInterface
«input device interface» CalibrationButton Interface
Notifies
«input device interface» ShaftInterface
«state dependent control» CalibrationControl
Controls
Updates
«entity» Calibration Constant
Uses
«entity» ShaftRotation Count
Uses

Example
of using the Strategy
Pattern: SpeedControl
Is the Strategy class

# Another example of using the Strategy Pattern: A Job Application System

- The complexity of class JobApplication is reduced by moving the validate() operation

  to the Strategy Pattern classes



ApplicantRuleFactory
(from patterns)
getApplicantValidationRule()

JobApplicantForm
(from patterns)
JOB_MANAGER : int = 1
JOB_WAIT_STAFF : int = 2
JOB_BUSSER : int = 3
JOB_BARTENDER : int = 4
JOB_HOSTER : int = 5
position : int
name : String
phone : String
email : String
yearsExp : Double
reference1 : String
reference2 : String
reference3 : String
legal : boolean = false
isLegal()
setLegal()
getPosition()
setPosition()
getName()
setName()
getPhone()
setPhone()
getEmail()
setEmail()
getYearsExp()
setYearsExp()
getReference1()
setReference1()
getReference2()
setReference2()
getReference3()
setReference3()
validate()

FormValidator
(from patterns)
successMessage : String = "\nThank you for submitting your job application."
isEmpty()
validate()
basicValidation()

FormSuccess
(from common)
success : boolean = false
resultMessage : String = null
FormSuccess()
isSuccess()
setSuccess()
getResultMessage()
setResultMessage()

ManagerValidator
(from patterns)
validate()

BusValidator
(from patterns)
validate()

HostValidator
(from patterns)
validate()

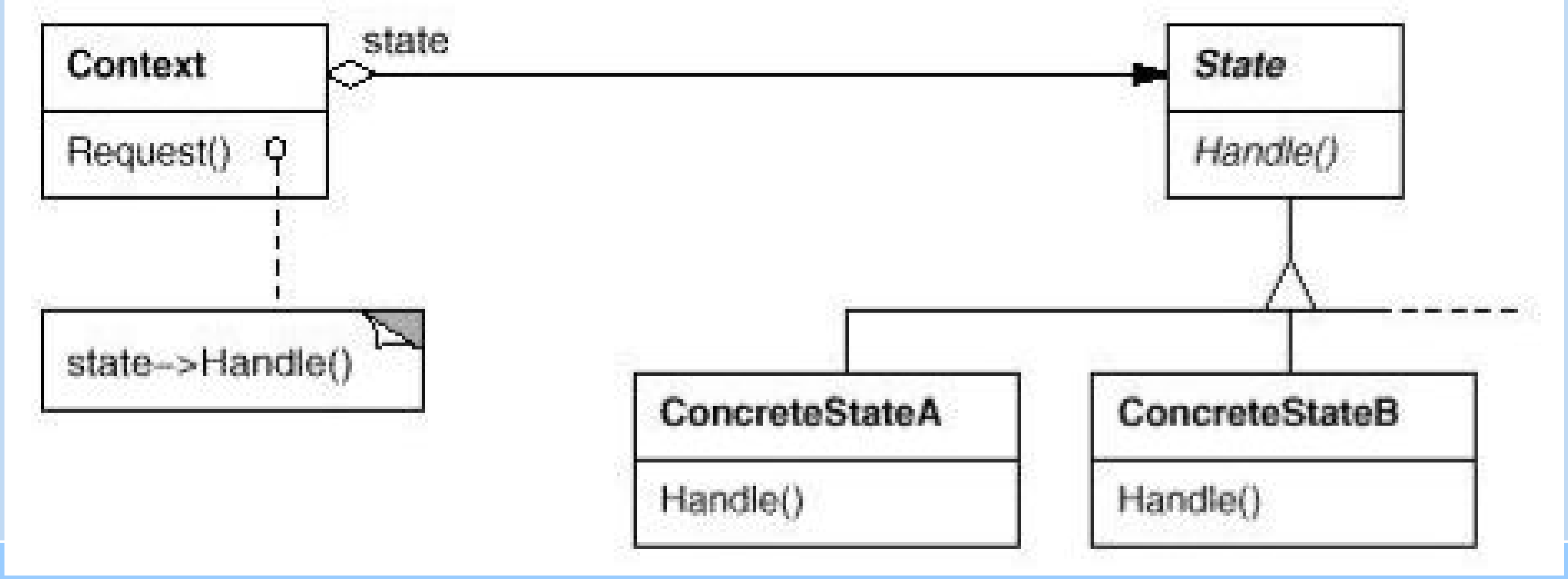
BartenderValidator
(from patterns)
validate()

WaitStaffValidator
(from patterns)
validate()

JobApplicantClient
(from common)
main()
runTest()

# Examples of Design Patterns
# The State Pattern

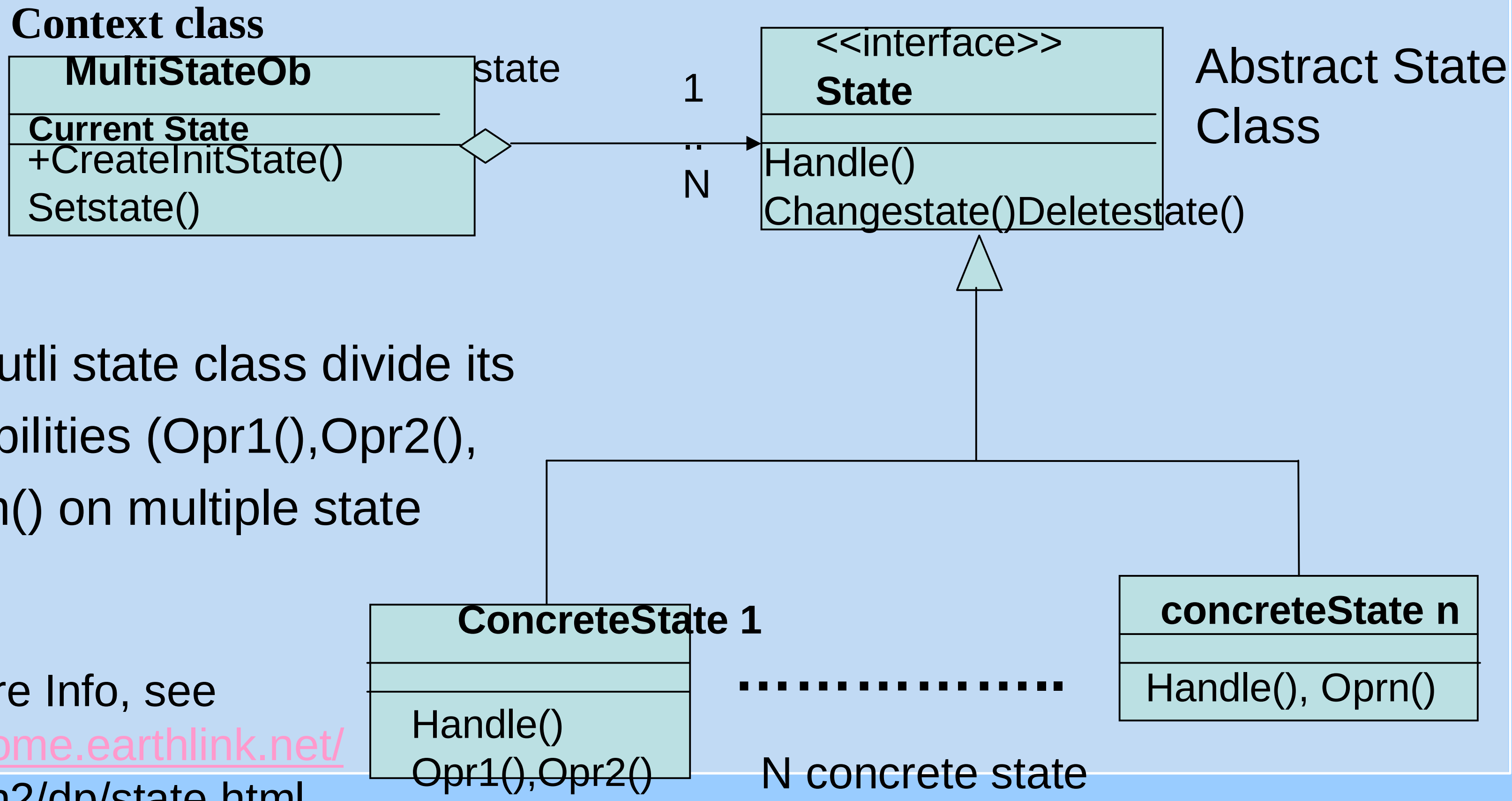
- Similar in structure (static) to the Strategy pattern but differs in dynamics
- Events are handled based on the current state of the object



# Examples of Design Patterns
# The State Pattern

- The State Pattern: is a solution to the problem of how to make the behavior of an object depend on its state.

The Context Class



Abstract State Class

Lets a mutli state class divide its responsibilities (Opr1(),Opr2(), and Oprn() on multiple state classes.

N concrete state classes

For more Info, see http://home.earthlink.net/~huston2/dp/state.html

# Examples of Design Patterns
# The State Pattern

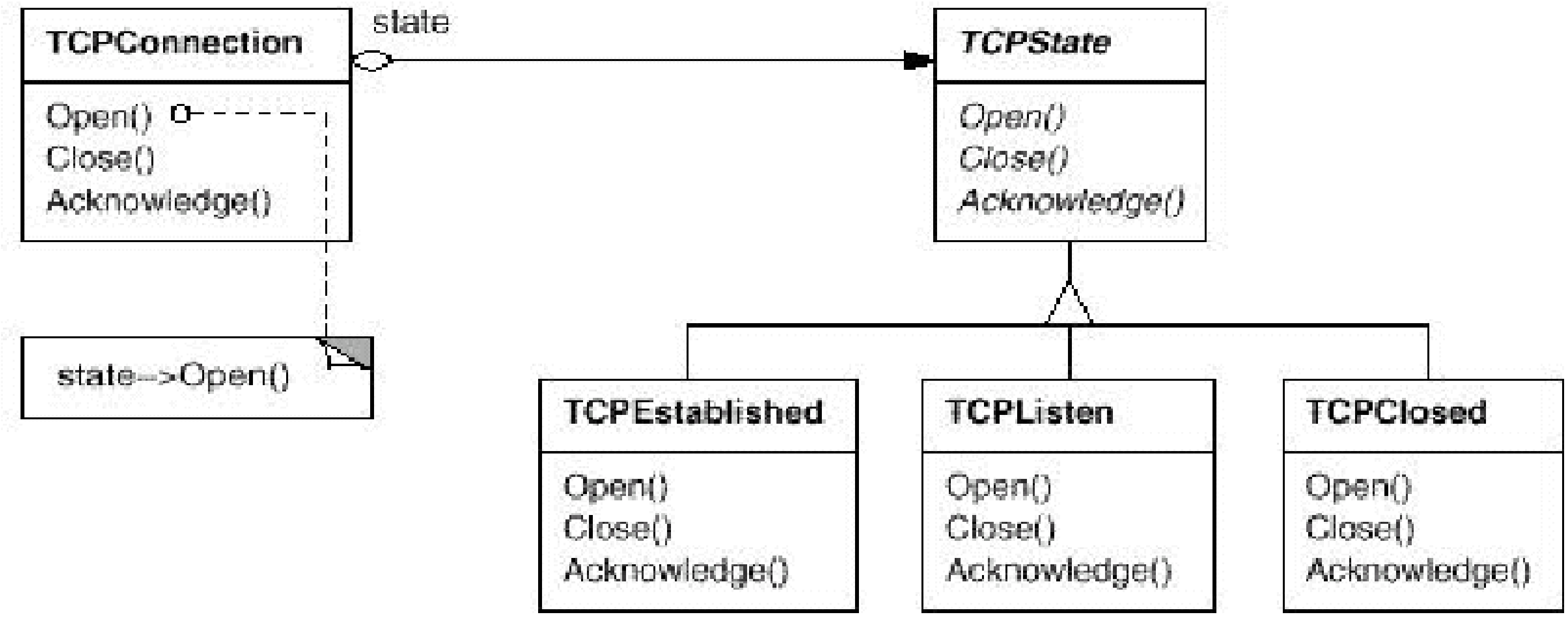
- The State pattern is a similar in structure to the Strategy Pattern but with different behavior or dynamics. the state objects are active one at a time depending on the actual state of the context object.
- The structure is defined as follows:
  - Define a "context" class to present a single interface to the outside world.
  - Define a State abstract base class.
  - Represent the different "states" of the state machine as derived classes of the State base class.
  - Define state-specific behavior in the appropriate State derived classes.
  - Maintain a pointer to the current "state" in the "context" class.
  - To change the state of the state machine, change the current "state" pointer
- State Transitions can be defined for each State class
- To be discussed later at length in slides 10 on

# Examples of Design Patterns

- The context class Multistateob would create the initial state object to provide the services of the initial state (it will set its current state to its initial state)
- The initial state object would sense the condition for state transition to a new state, when this occurs it would then create an object of the new state and destroy itself
- Each state object implements the transition, actions, and activities in the state it represents

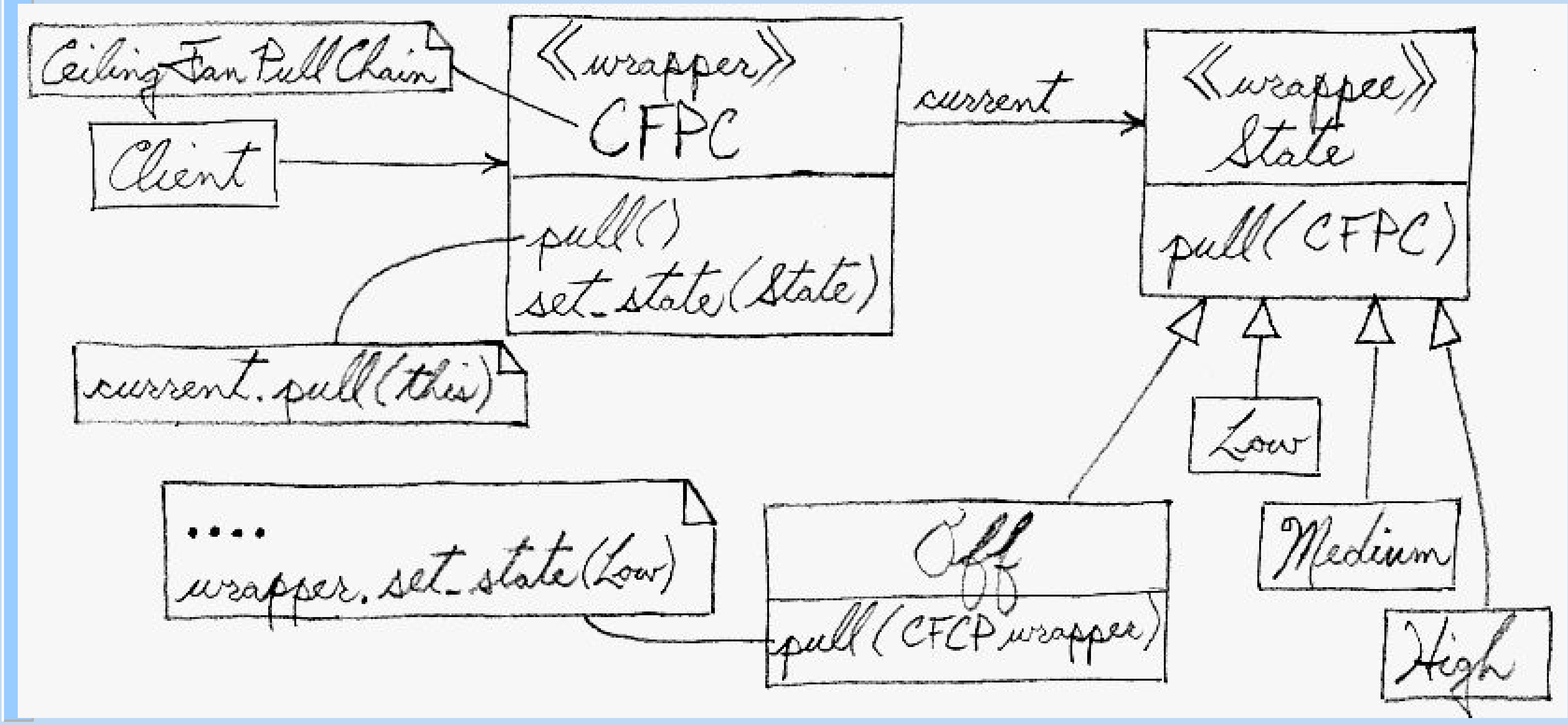
# Examples of Design Patterns
# The State Pattern

- TCP connection example



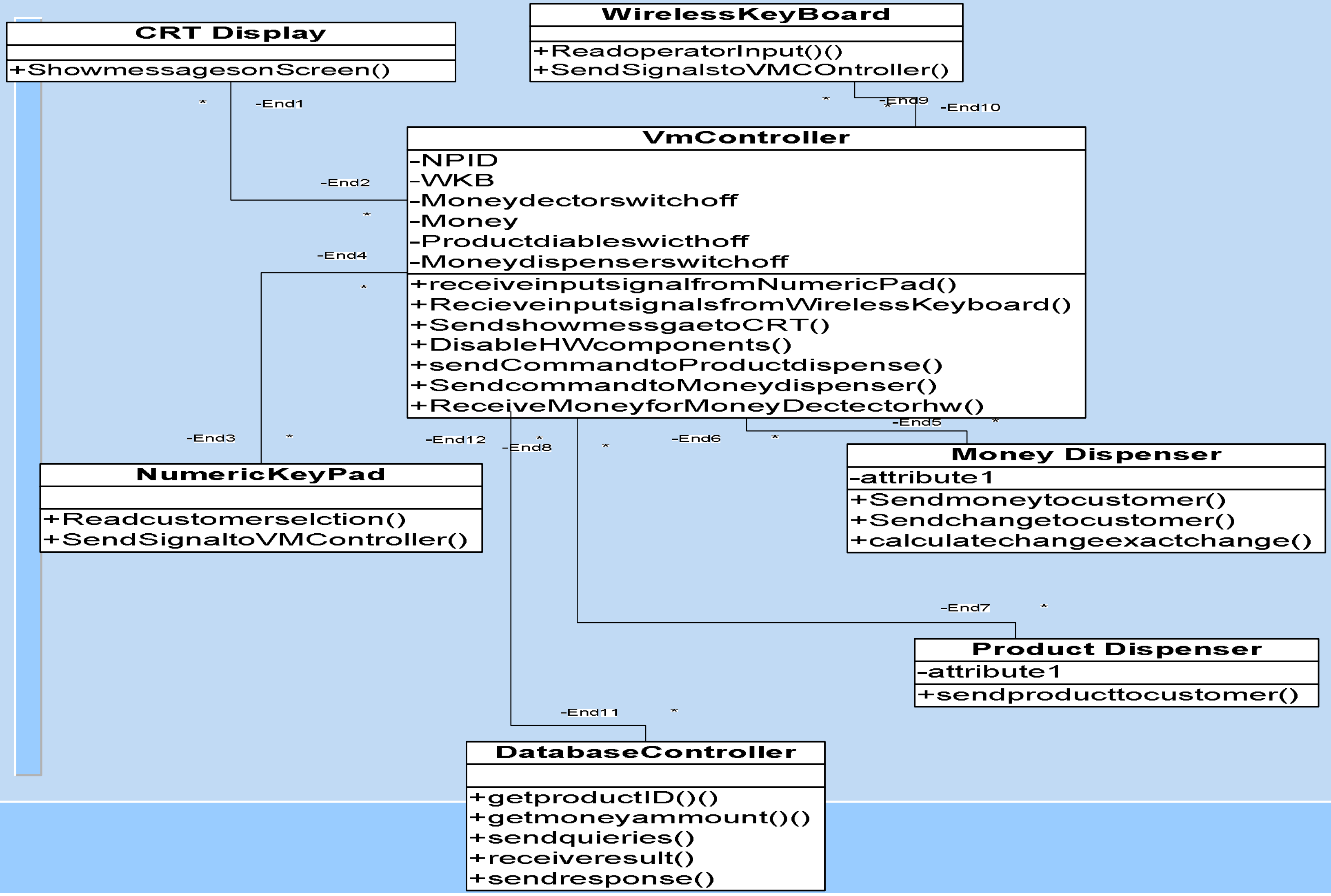
# Examples of Design Patterns
# The State Pattern

A Ceiling Fan Pull Chain Example :



# Design Patterns Application Example: The VM System

- The initial design class diagram of a Vending Machine



**CRT Display**

+ShowmessagesonScreen()

**WirelessKeyBoard**

+ReadoperatorInput()()
+SendSignalstoVMCOntroller()

**VmController**

-NPID
-WKB
-Moneydectorswitchoff
-Money
-Productdiableswicthoff
-Moneydispenserswitchoff

+receiveinputsignalfromNumericPad()
+RecieveinputsignalsfromWirelessKeyboard()
+SendshowmessgaetoCRT()
+DisableHWcomponents()
+sendCommandtoProductdispense()
+SendcommandtoMoneydispenser()
+ReceiveMoneyforMoneyDectectorhw()

**NumericKeyPad**

+Readcustomerselction()
+SendSignaltoVMController()

**Money Dispenser**

-attribute1

+Sendmoneytocustomer()
+Sendchangetocustomer()
+calculatechangeexactchange()

**Product Dispenser**

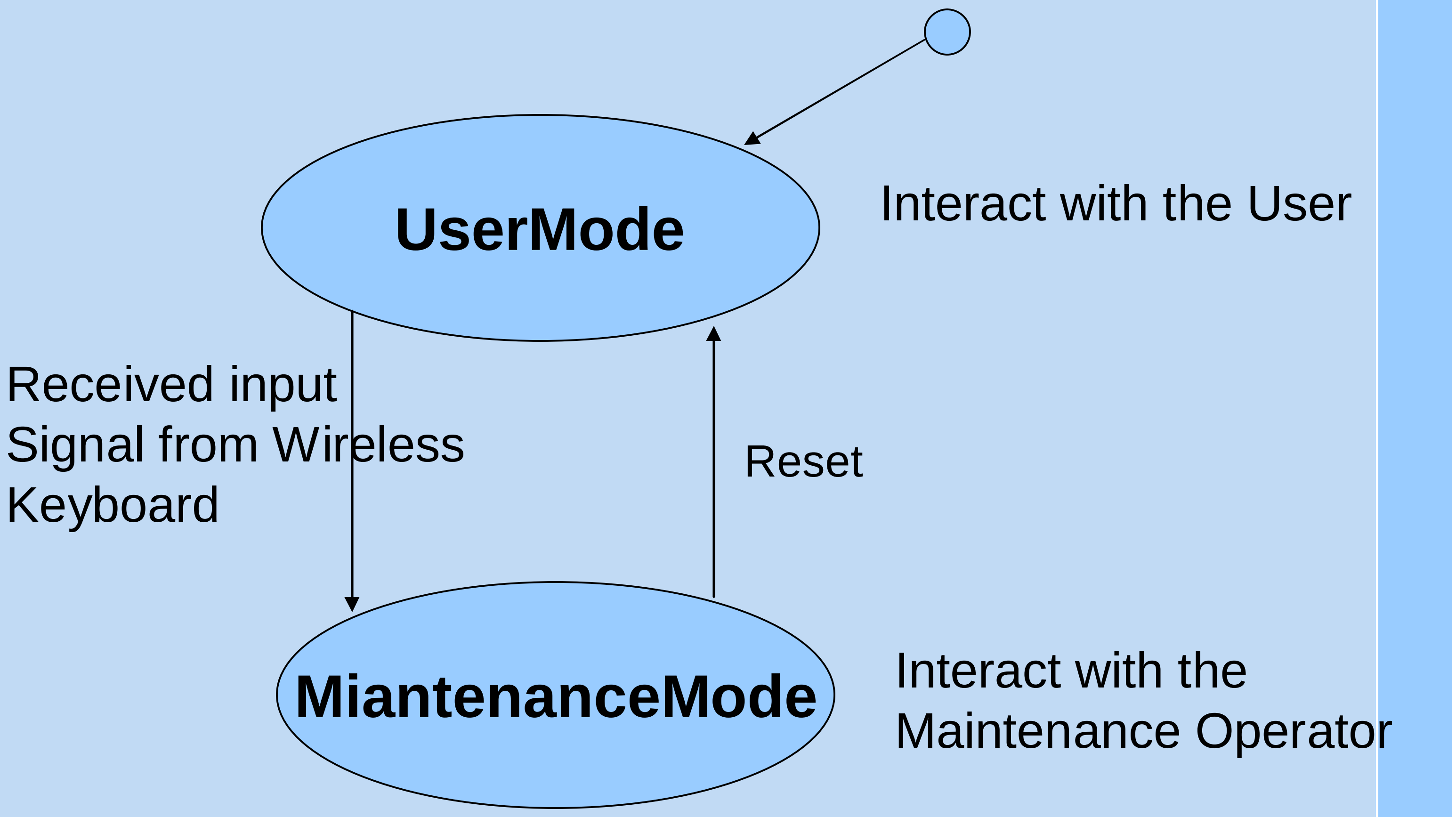
-attribute1

+sendproducttocustomer()

**DatabaseController**

+getproductID()()
+getmoneyammount()()
+sendquieries()
+receiveresult()
+sendresponse()

# The State Diagram of the VM Controller Class



**UserMode**

Interact with the User

Received input
Signal from Wireless
Keyboard

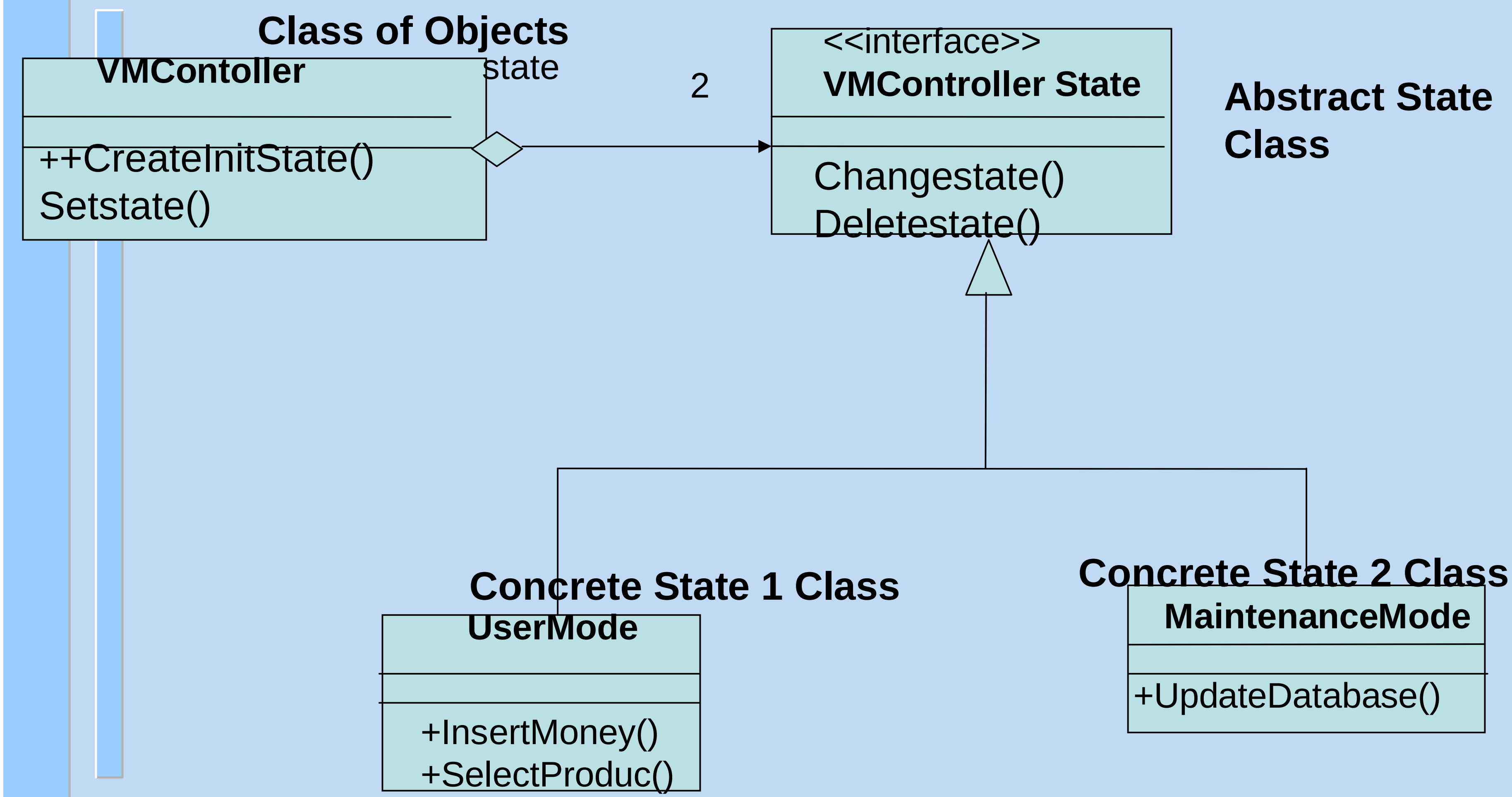
Reset

**MiantenanceMode**

Interact with the
Maintenance Operator

# Design Patterns Application Example: The VM System

- The State Pattern Applied to the 2 state VM example



# Recall The Consolidated Collaboration Diagram of the ATM Client Subsystem

The diagram can be easily used to develop the class diagram of the ATM Client Subsys.

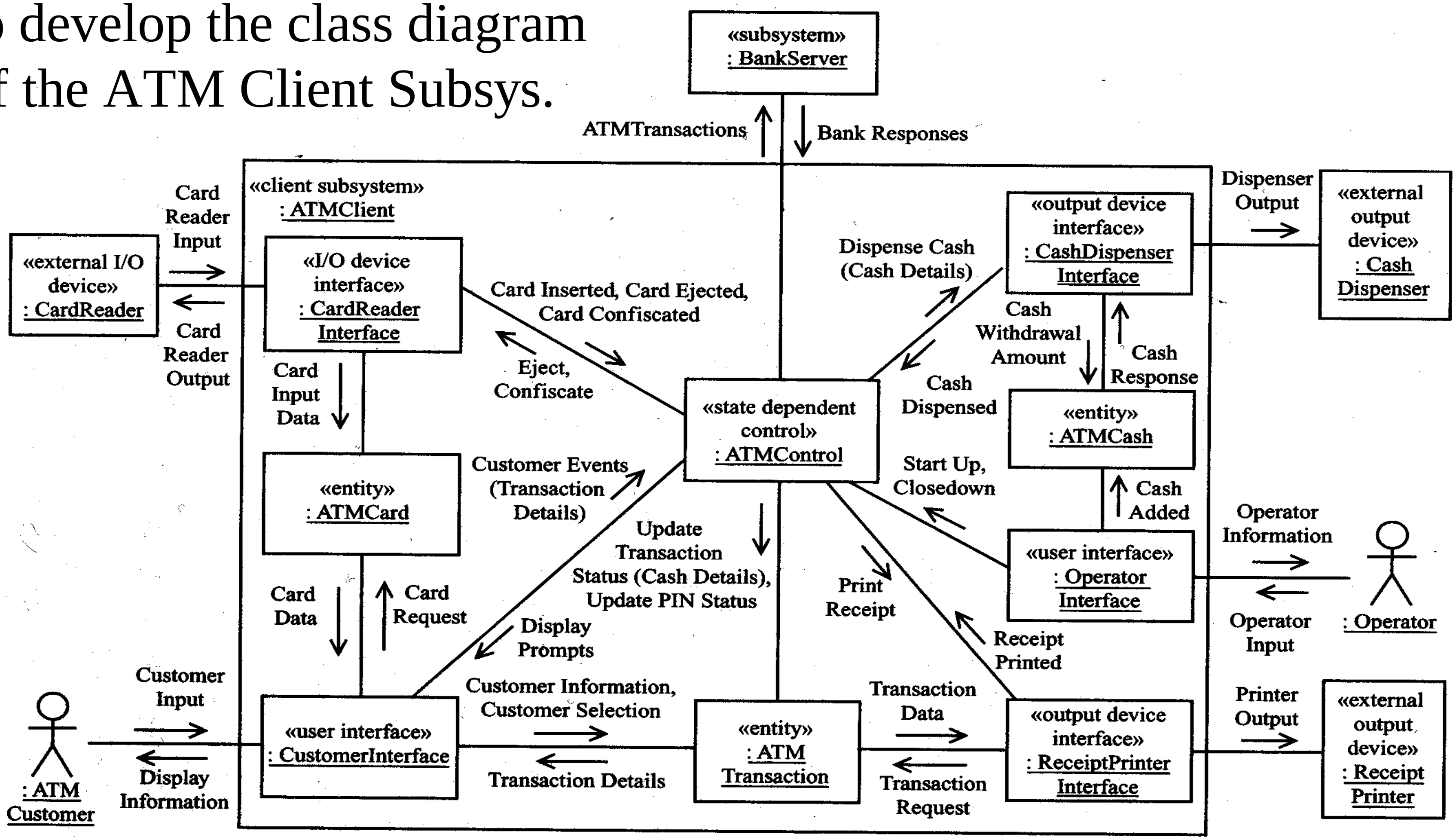


**Figure 12.5** *Example of consolidated collaboration diagram: ATM Client subsystem*

# Example: How can we apply The State Pattern to the ATM system using This ATM controller StateChart ?

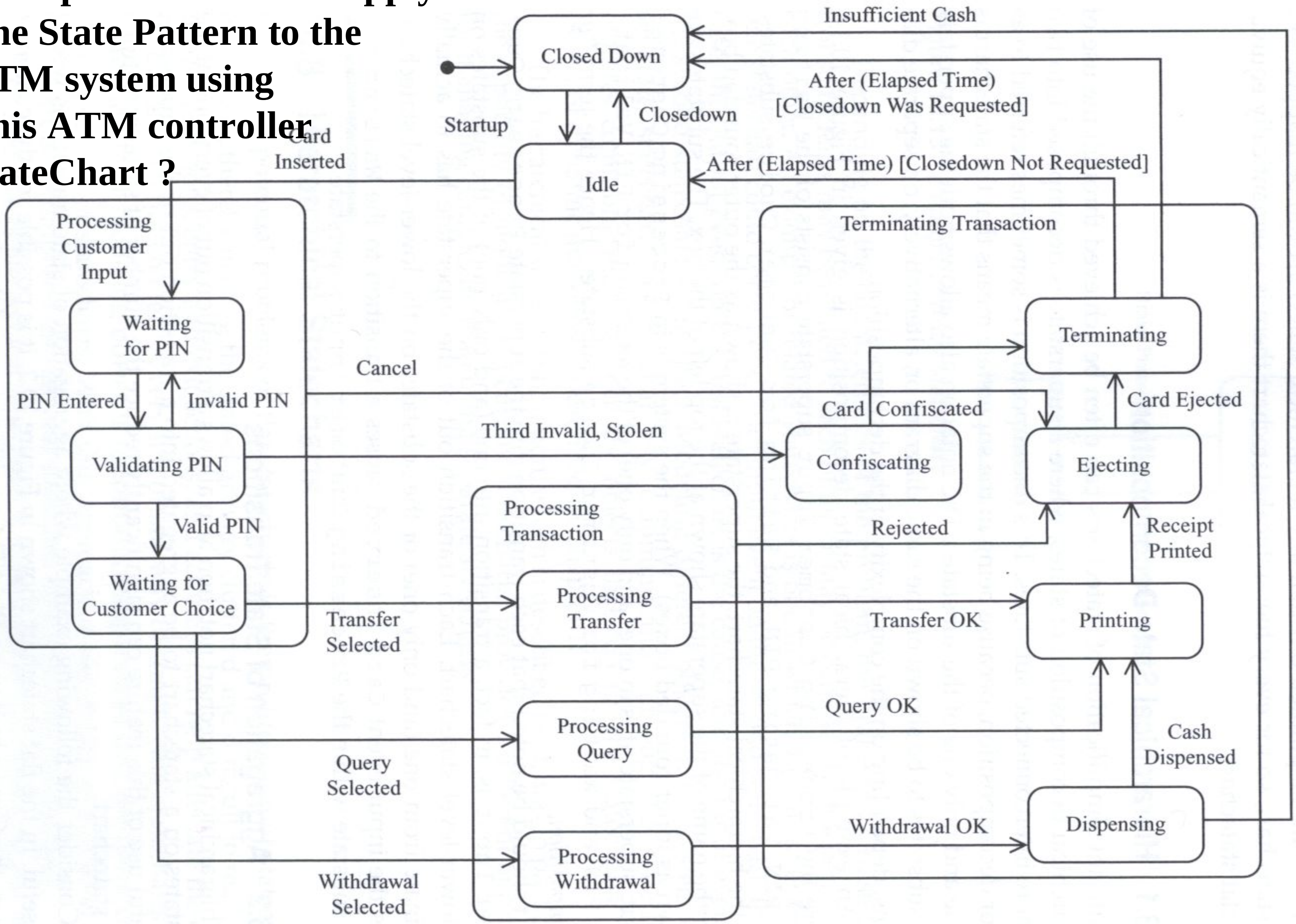
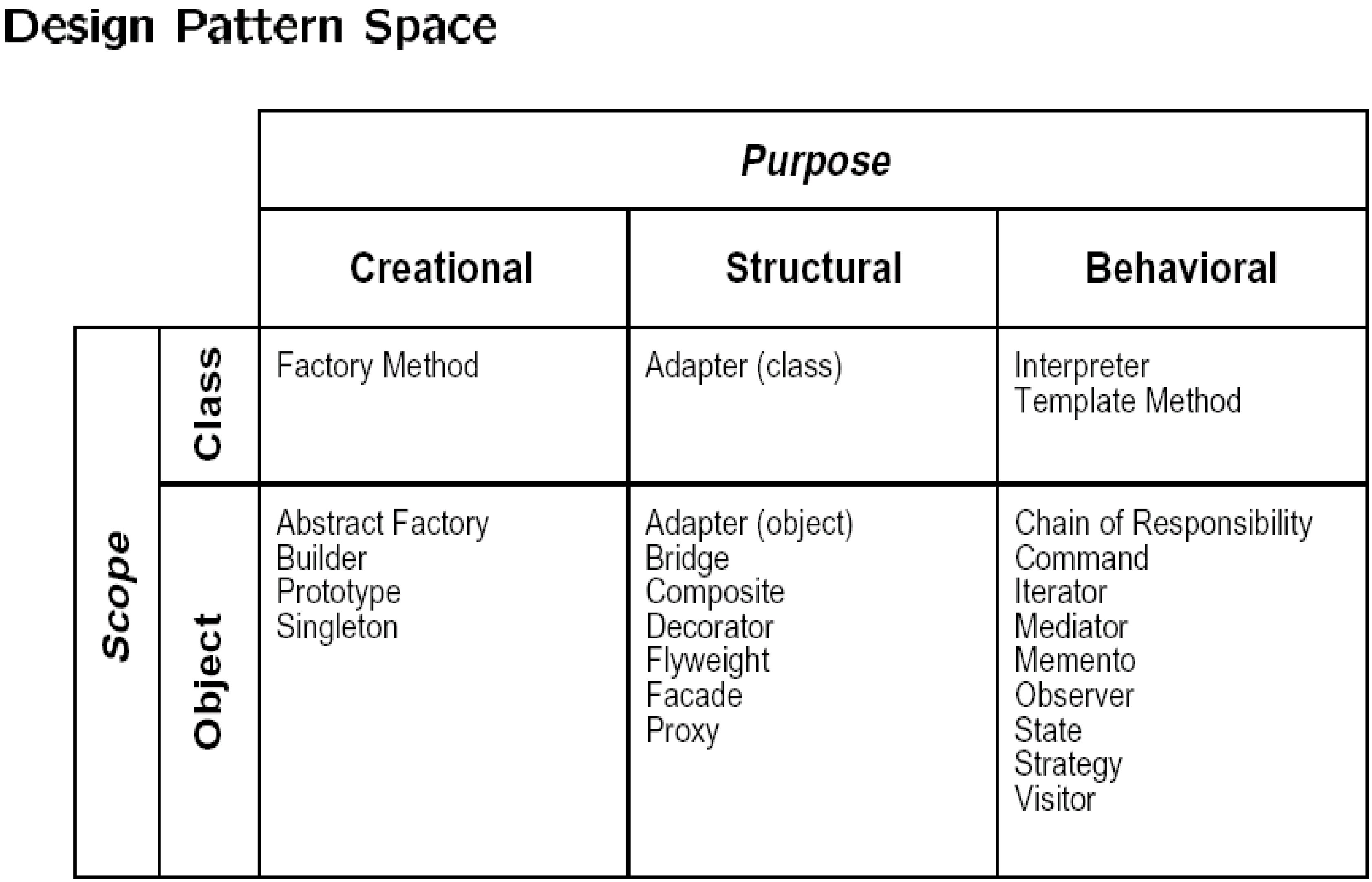


**Figure 10.14** *Example of hierarchical statechart*

# Types of Design Patterns

## The Gang of Four (GoF) Patterns (Gamma et al 1995)

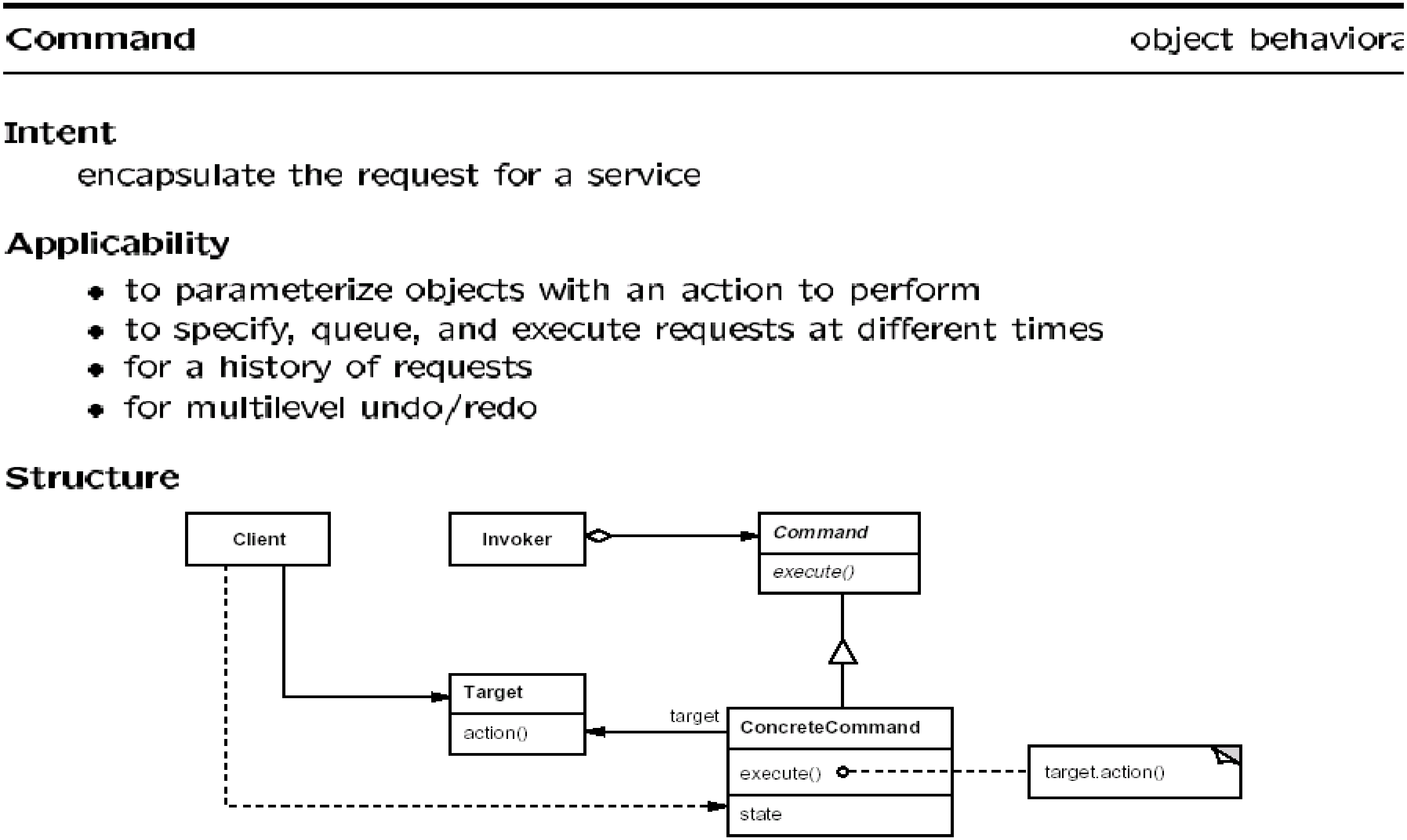


**Design Pattern Space**

| | | *Purpose* | | |
|---|---|---|---|---|
| | | **Creational** | **Structural** | **Behavioral** |
| ***Scope*** | **Class** | Factory Method | Adapter (class) | Interpreter<br>Template Method |
| | **Object** | Abstract Factory<br>Builder<br>Prototype<br>Singleton | Adapter (object)<br>Bridge<br>Composite<br>Decorator<br>Flyweight<br>Facade<br>Proxy | Chain of Responsibility<br>Command<br>Iterator<br>Mediator<br>Memento<br>Observer<br>State<br>Strategy<br>Visitor |

# Examples of Behavioral Design Patterns

**The Command Pattern:** operator commands or user or customer requests are treated as a class of objects



Command object behaviora

**Intent**

encapsulate the request for a service

**Applicability**

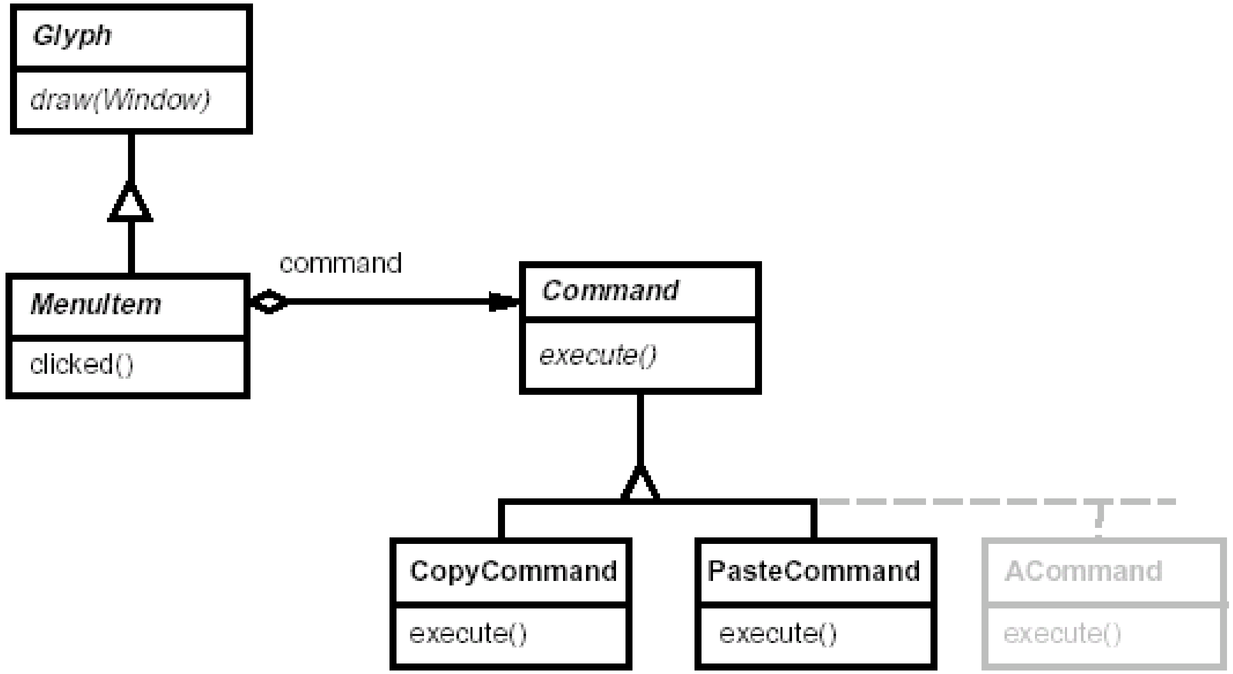
- to parameterize objects with an action to perform
- to specify, queue, and execute requests at different times
- for a history of requests
- for multilevel undo/redo

**Structure**

Client
Invoker
Command
execute()
Target
action()
target
ConcreteCommand
execute()
state
target.action()

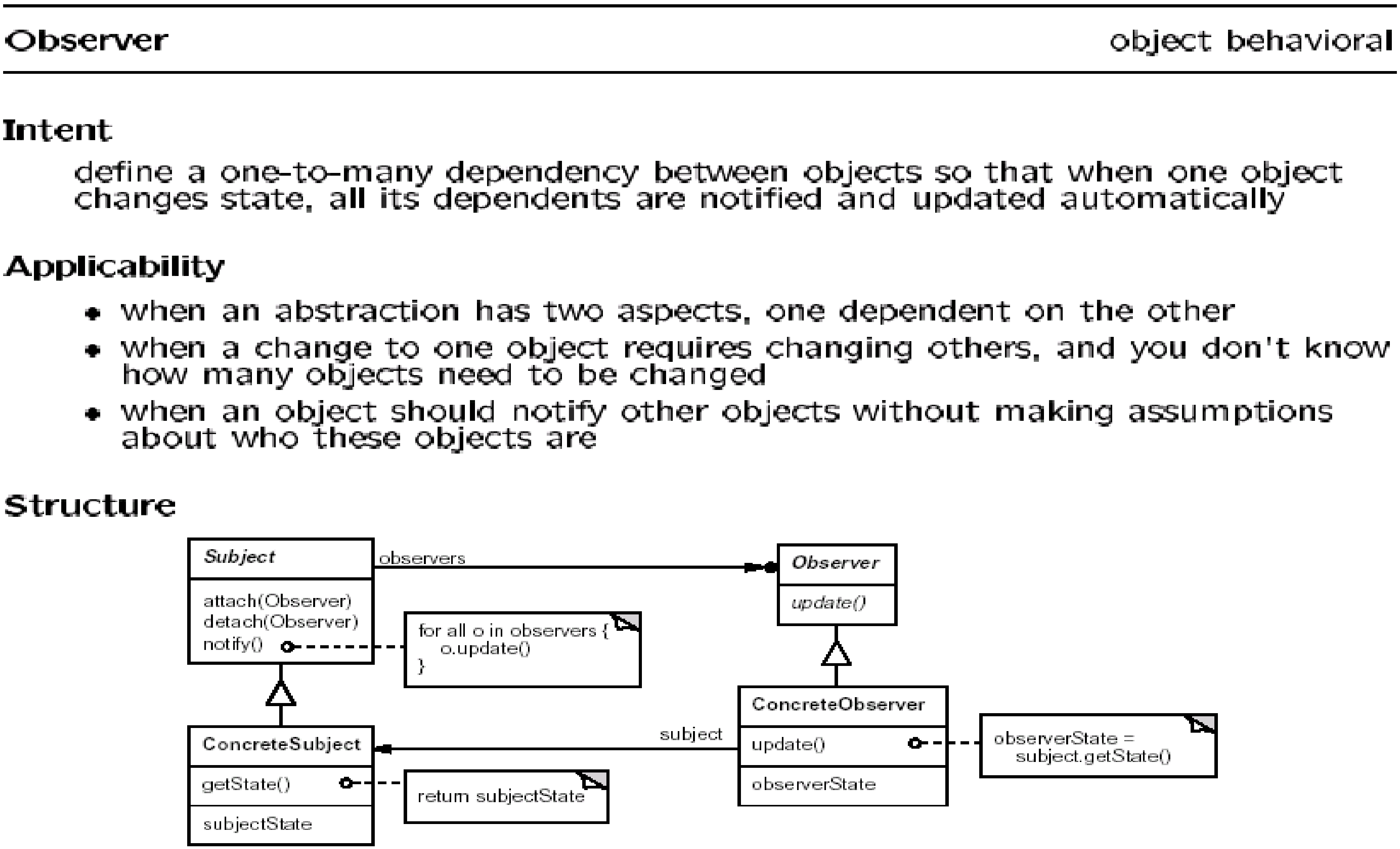
# The Command Pattern

- Example of using the Command Pattern in a Menu driven graphics application



# Examples of Behavioral Design Patterns

**The Observer Pattern:** Multiple observer objects are notified when changes of states subjects occur



**Observer** object behavioral

**Intent**

define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically

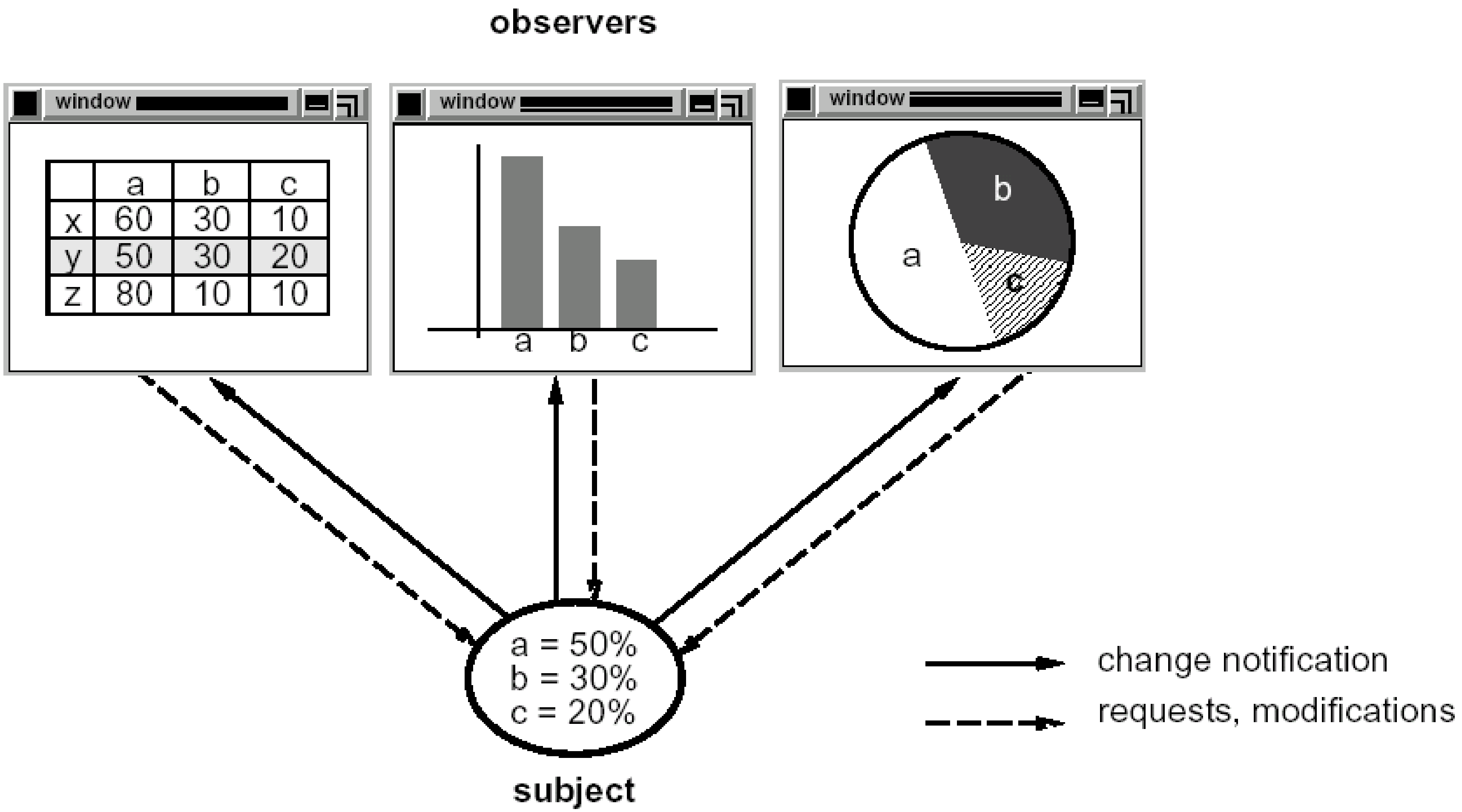
**Applicability**

- when an abstraction has two aspects, one dependent on the other
- when a change to one object requires changing others, and you don't know how many objects need to be changed
- when an object should notify other objects without making assumptions about who these objects are

**Structure**

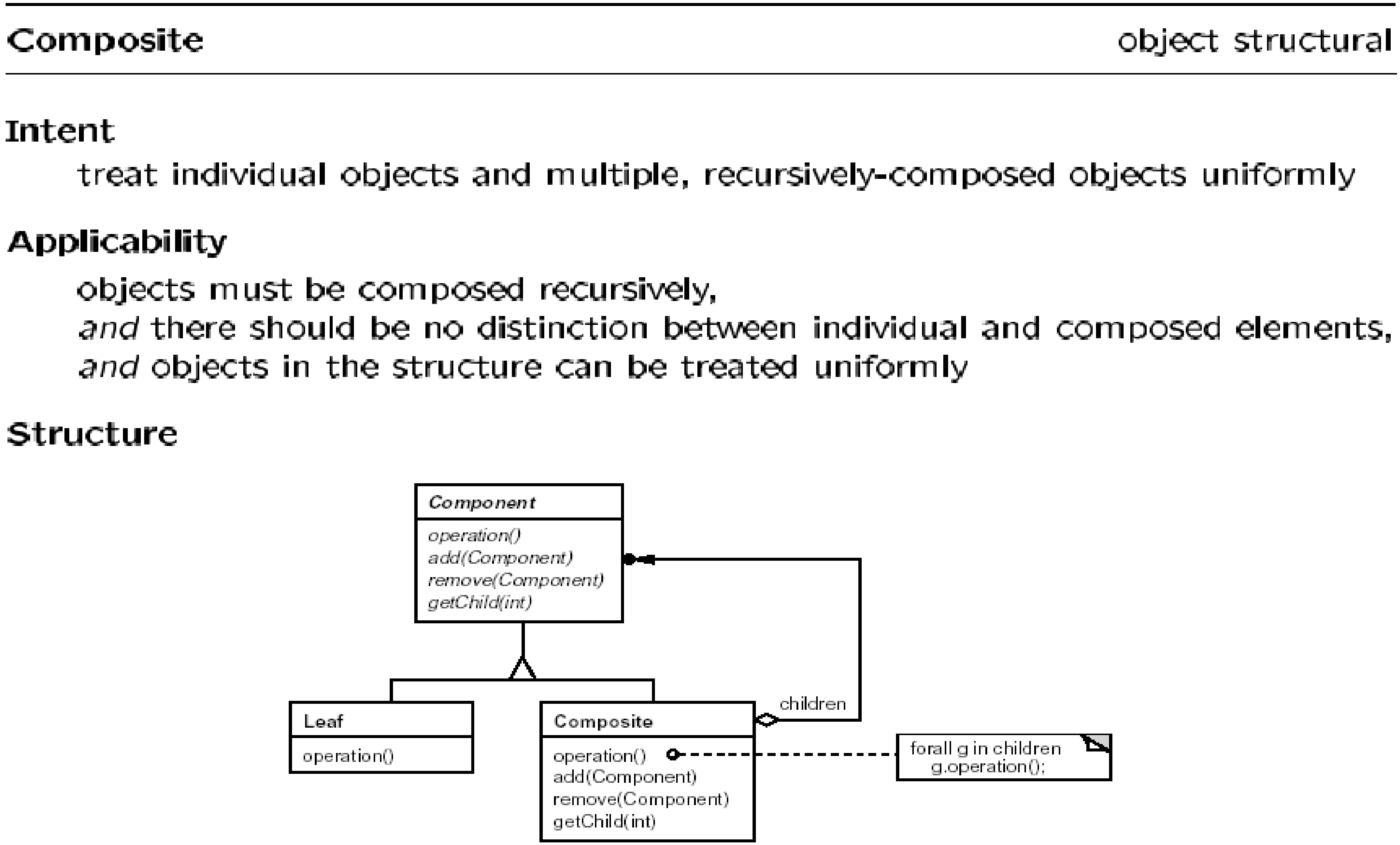
# The Observer Pattern

## Example: Observer



| | a | b | c |
|---|---|---|---|
| x | 60 | 30 | 10 |
| y | 50 | 30 | 20 |
| z | 80 | 10 | 10 |

# Examples of Structural Design Patterns
# The Composite Pattern



**Composite** object structural

**Intent**

treat individual objects and multiple, recursively-composed objects uniformly

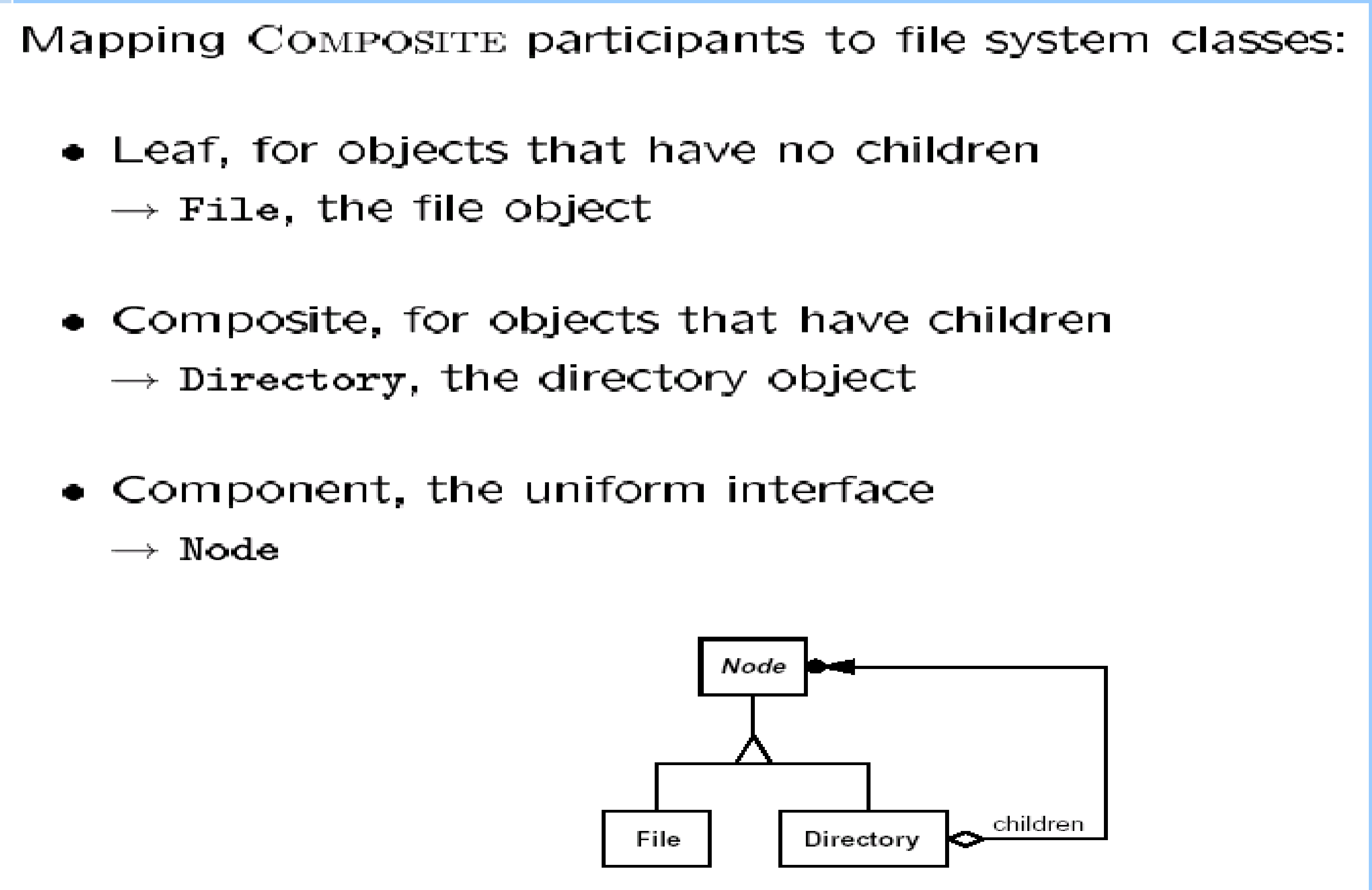
**Applicability**

objects must be composed recursively,
*and* there should be no distinction between individual and composed elements,
*and* objects in the structure can be treated uniformly

**Structure**

Component
operation()
add(Component)
remove(Component)
getChild(int)
Leaf
operation()
Composite
operation()
add(Component)
remove(Component)
getChild(int)
children
forall g in children
g.operation();

# Examples of Design Patterns
# **The Composite Pattern** :File System Structure



Mapping COMPOSITE participants to file system classes:

- Leaf, for objects that have no children

  → `File`, the file object

- Composite, for objects that have children

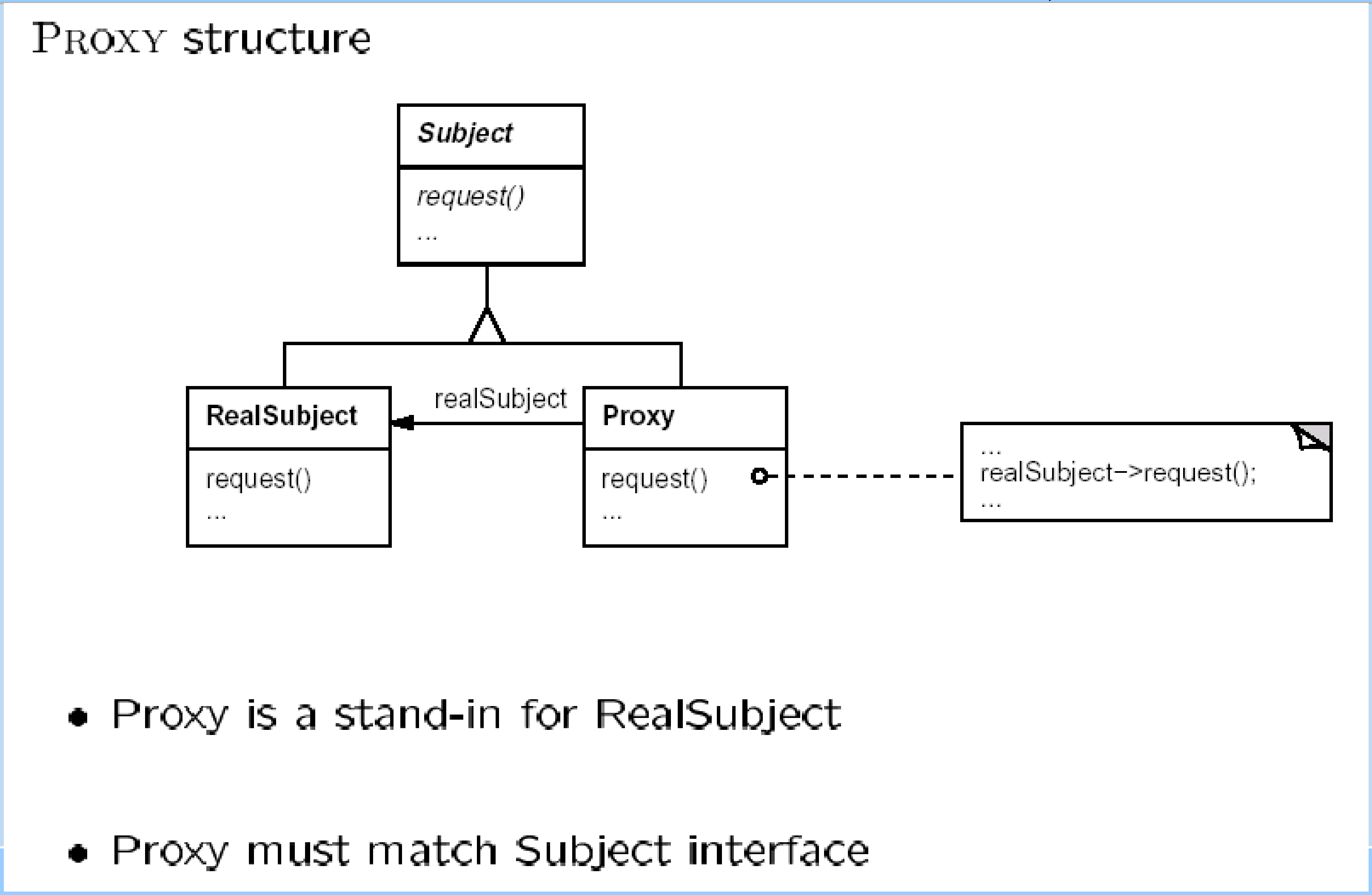
  → `Directory`, the directory object

- Component, the uniform interface

  → `Node`

# Examples of Structural Design Patterns

## The Proxy Pattern (used heavily in communication software, CORBA, SOA )



Proxy structure

- Proxy is a stand-in for RealSubject
- Proxy must match Subject interface

# Design Patterns Examples and Tutorials

- **Design Refinements Examples**
- Two tutorials by John Vlissides
  - **An Introduction to Design Patterns**

    Also on the design patterns CD by Gama et al
  - **Designing with Patterns**
- **Tutorial: More on Design patterns**
- **The VISITOR family of design patterns**
- **TEMPLATE METHOD & STRATEGY Patterns: :Inheritance vs. Delegation**
- **Pattern-Oriented Software Architectures**

  **Patterns & Frameworks for Concurrent & Distributed Systems by** D. C. Schmidt