SOFTWARE ARCHITECTURE AND DESIGN PATTERNS

UNIT - V

Dr. T. K. Rao

VVIT

Contents

BEHAVIORAL PATTERNS:

- Chain of responsibility (251)
- Command
- Interpreter
- Iterator
- Mediator
- Memento
- Observer
- State
- Strategy
- Template method
- Visitor

BEHAVIORAL PATTERNS

- These are concerned with algorithms & assignment of responsibilities between objects
- These describe the patterns of communication between them
- These patterns characterize complex control flow that's difficult to follow at run-time
- They shift focus away from flow of control to concentrate on the way objects are interconnected
- Behavioural class patterns use inheritance to distribute behaviour between classes
- This chapter includes two such patterns
- Template Method is the simpler and more common of the two
- A template method is an abstract definition of an algorithm.

Chain of responsibility

VVIT
VASIREDDY VENKATADRI INSTITUTE OF TECHNOLOGY

Intent

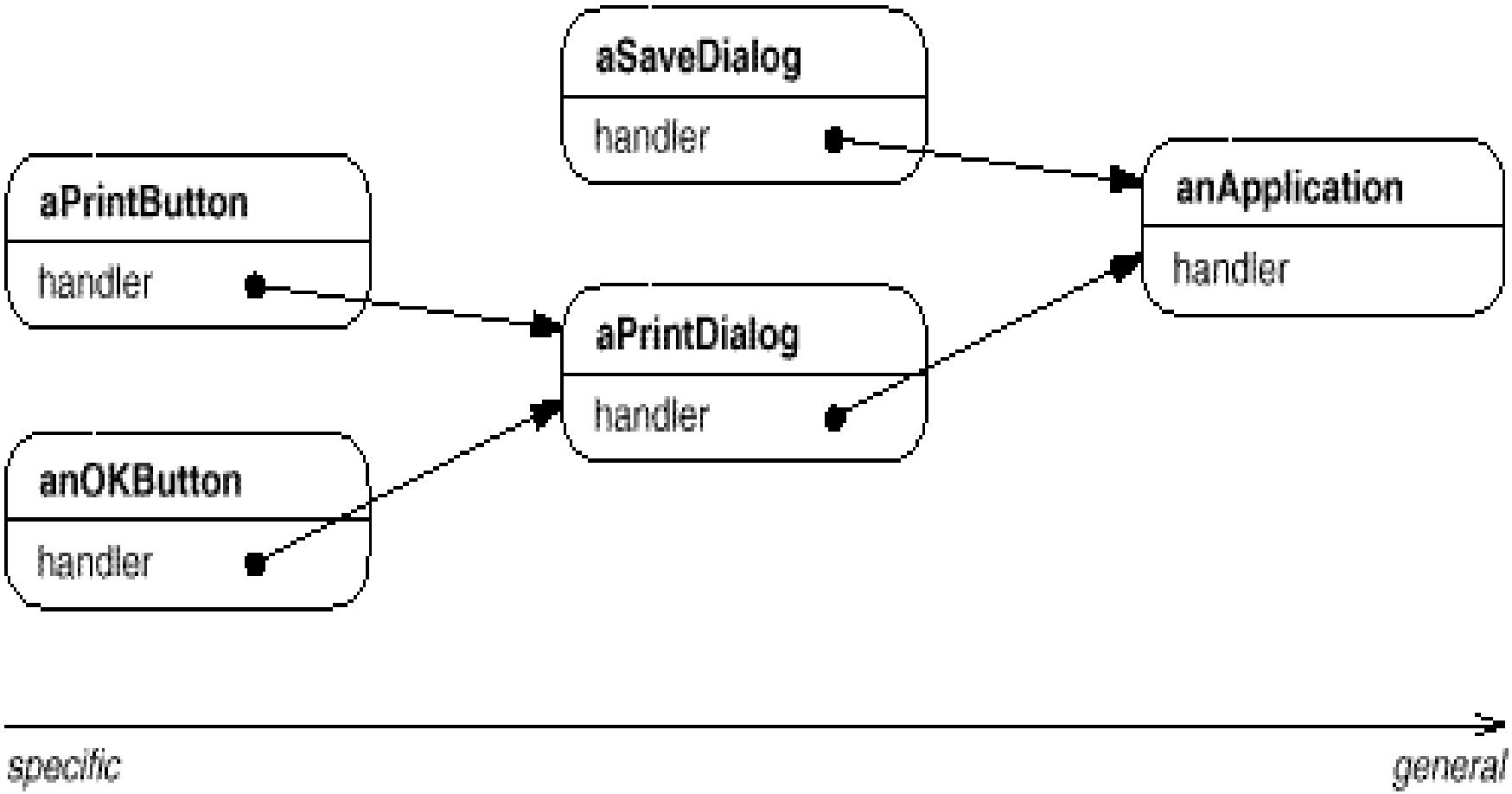
- Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request
- Chain the receiving objects & pass request along the chain until an object handles it

Motivation

- Consider a context-sensitive help facility for a graphical user interface
- User can obtain help information on any part of interface by clicking on it
- E.g., a button widget in a dialog box might have different help information than a similar button in the main window.

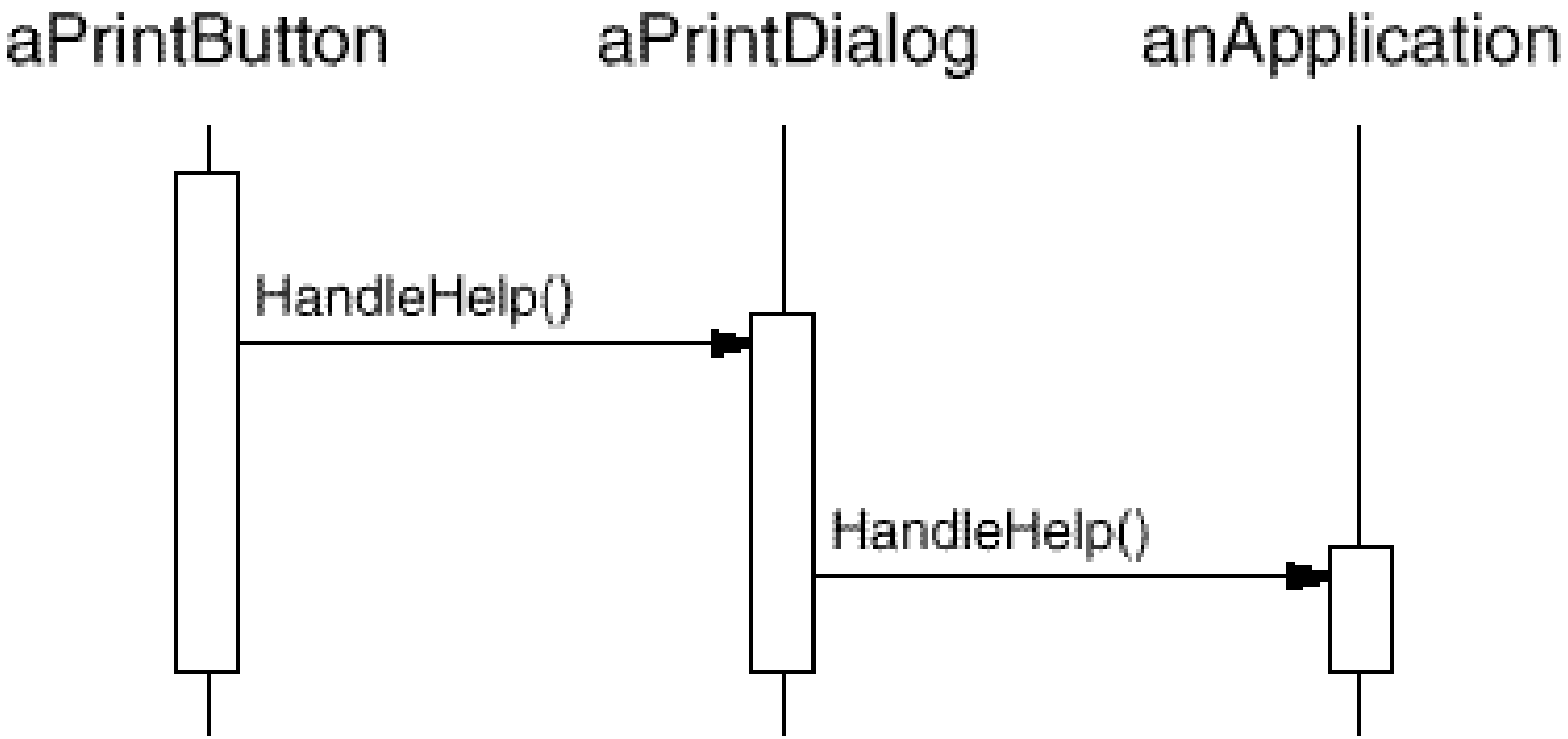
- If no specific help info exists for that part of GUI, help system should display a more general help
- Hence it's natural to organize help info according to its generality—from the most specific to the most general
- Further more, it's clear that a help request is handled by one of several user interface objects
- Problem is the object that ultimately *provides* help isn't know explicitly to the object (e.g., the button) that *initiates* the help request

- A way is needed to decouple button that initiates help request from objects that might provide help information
- Chain of Responsibility pattern defines how that happens
- The idea of this pattern is to decouple senders and receivers by giving multiple objects a chance to handle a request
- Request is passed along a chain of objects until one of them handles it

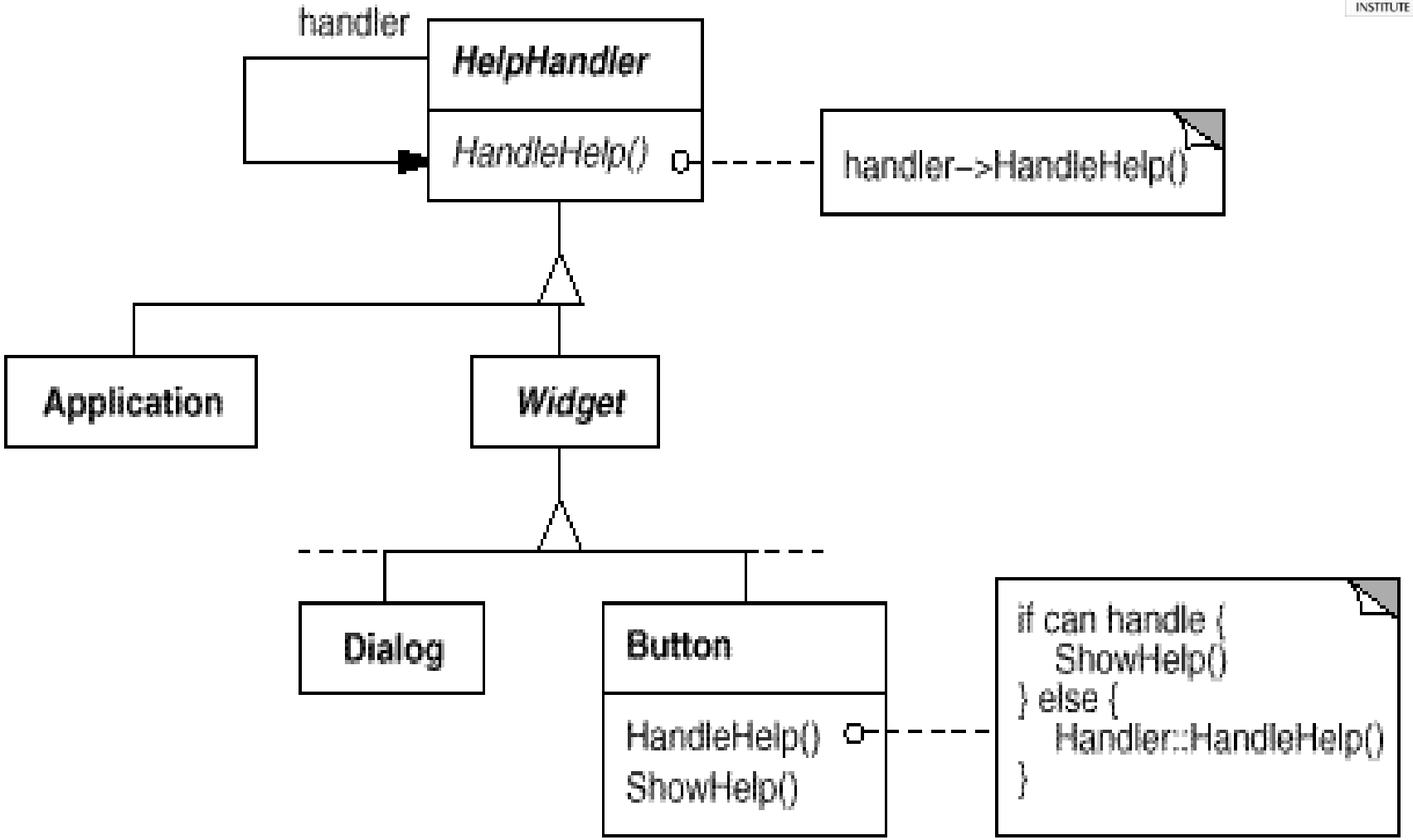


Chain of Responsibility

- First object in the chain receives request and either handles it or forwards it to the next object on the chain, and so on
- Object that made the request has no explicit knowledge of who handles it—we say the request has an **implicit receiver**
- The following interaction diagram illustrates how the help request gets forwarded along the chain:


aPrintButton
aPrintDialog
anApplication
HandleHelp()
HandleHelp()

- Here, neither aPrintButton nor aPrintDialog handles the request;
- Client that sent request has no direct reference to object that fulfills it
- Each object on the chain shares a common interface for handling requests and for accessing its **successor** on the chain
- E.g., help system define a HelpHandler class with a HandleHelp operation
- HelpHandler is parent for candidate object classes, or it is amixin class
- Classes that want to handle help requests can makeHelpHandler a parent:



Chain of Responsibility

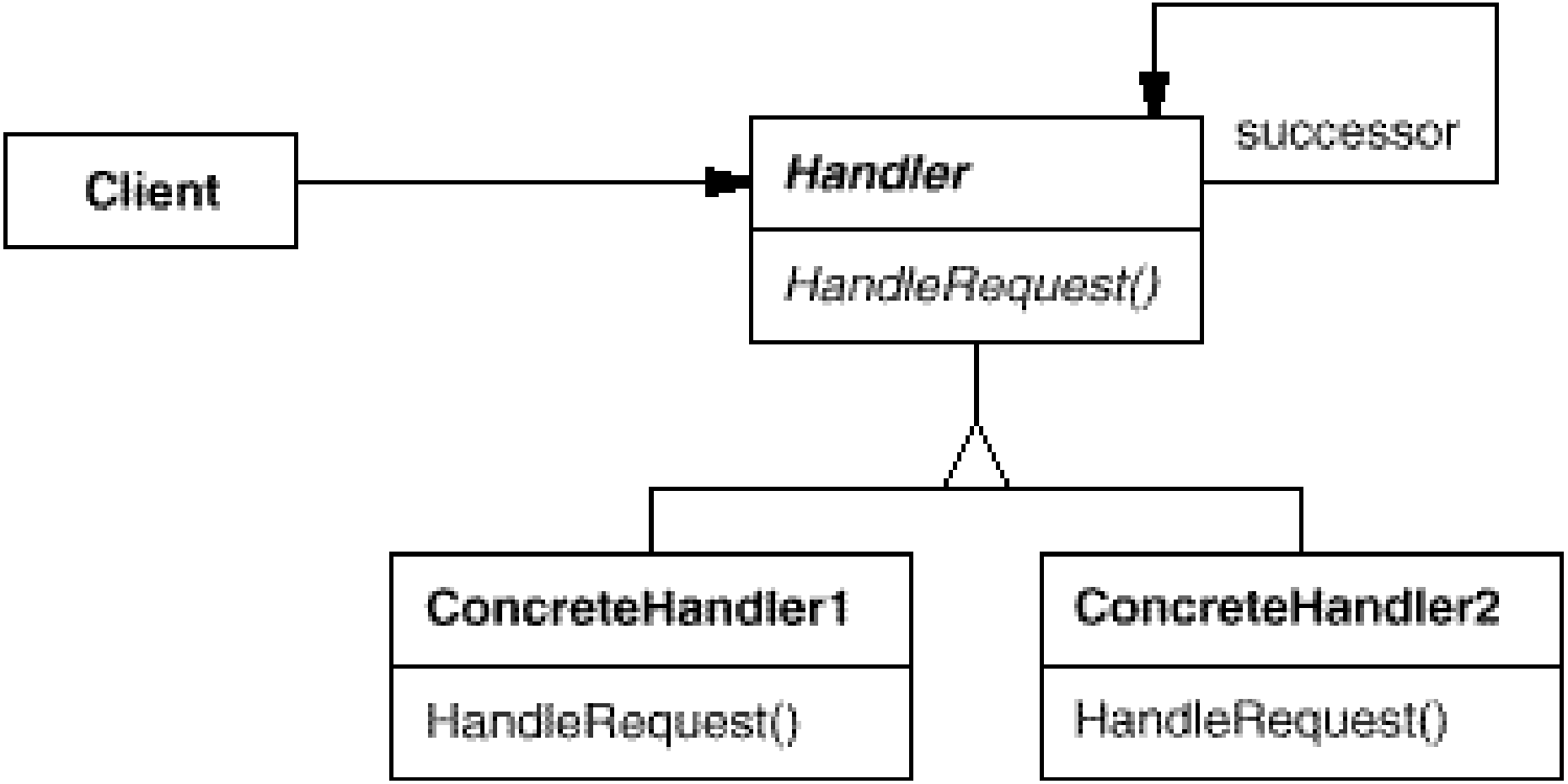
- The Button, Dialog, and Application classes use HelpHandler operations to handle help requests
- HelpHandler's HandleHelp operation forwardsthe request to the successor by default
- Subclasses can override this operation to provide help under the right circumstances;
- Otherwise they can use default implementation to forward the request.

Applicability

Use Chain of Responsibility when

- More than one object may handle a request, and the handler isn't known *apriori*
- Handler should be ensured automatically
- You want to issue a request to one of several objects without specifying the receiver explicitly
- The set of objects that can handle a request should be specified dynamically.

Structure


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

A typical object structure might look like this

Participants

Handler (HelpHandler)

- Defines an interface for handling requests.
- (optional) implements the successor link.

ConcreteHandler (PrintButton, PrintDialog)

- Handles requests it is responsible for.
- Can access its successor.
- If ConcreteHandler can handle request, it does so; else it forwards the request to its successor.

Client

- Initiates the request to a ConcreteHandler object on the chain.

Collaborations

- When a request is issued, request propagates along chain until a ConcreteHandler object handle it.

Consequences

It has the following benefits and liabilities:

1. ***Reduced coupling***:

- Pattern frees an object from knowing which object handles a request
- Both receiver & sender have no explicit knowledge of each other, & object in chain doesn't have to know chain's structure.
- Chain of Responsibility can simplify object interconnections.
- Instead of objects maintaining references to all candidate receivers, they keep a single reference to their successor.

2. *Added flexibility in assigning responsibilities to objects*.

- Chain of Responsibility gives added flexibility in distributing responsibilities among objects.
- add/change responsibilities for handling a request by adding /changing chain at run-time.
- You can combine this with subclassing to specialize handlers statically.

3. *Receipt isn't guaranteed*

- Since a request has no explicit receiver, there's no *guarantee* it'll be handled—the request can fall off end of chain without being handled.
- A request can also go unhandled when the chain is not configured properly

Command

Intent

- Encapsulate a request as object, by letting you parameterize clients with various requests, queue/ log requests, & support undoable operations

Also Known As

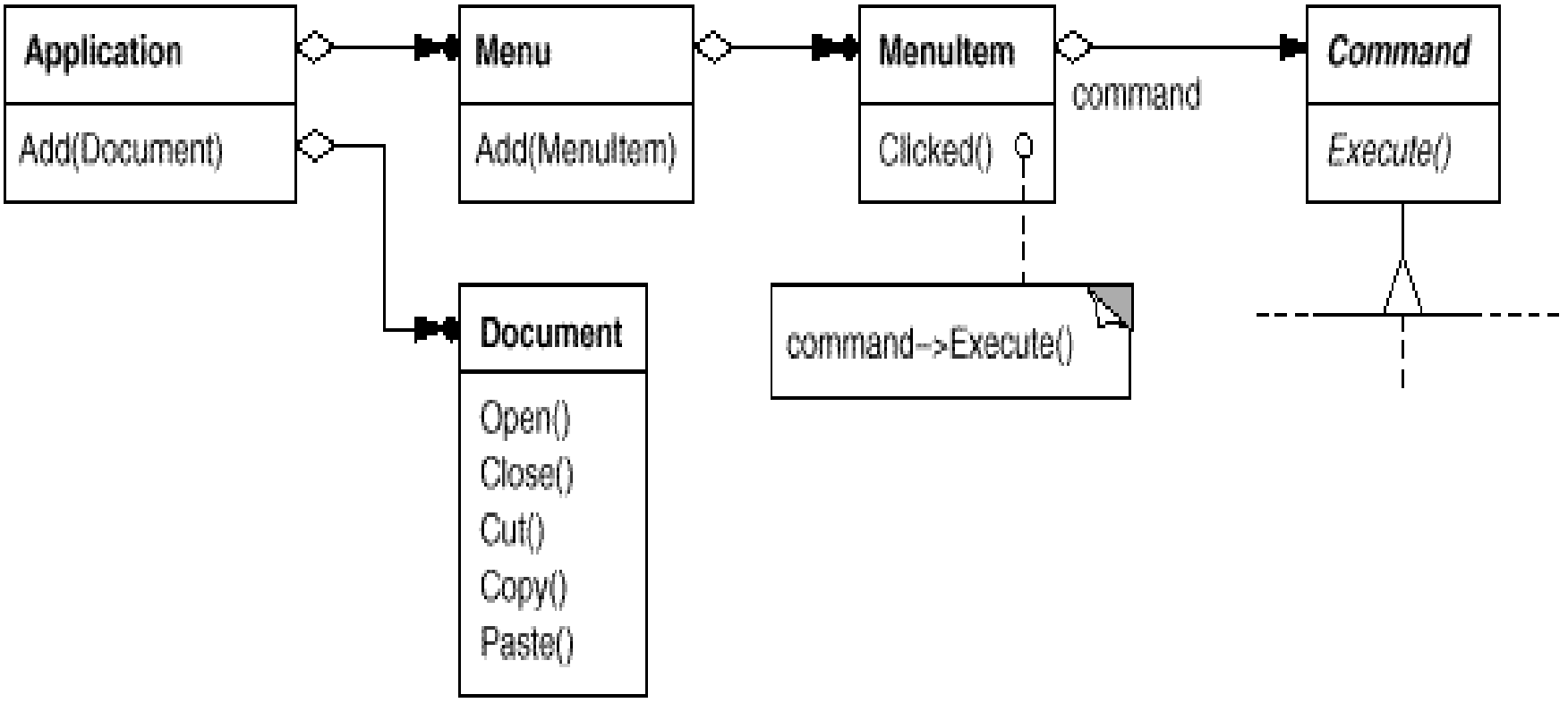
- Action, Transaction

Motivation

- To issue requests to objects without knowing anything about the operation being requested or receiver of request

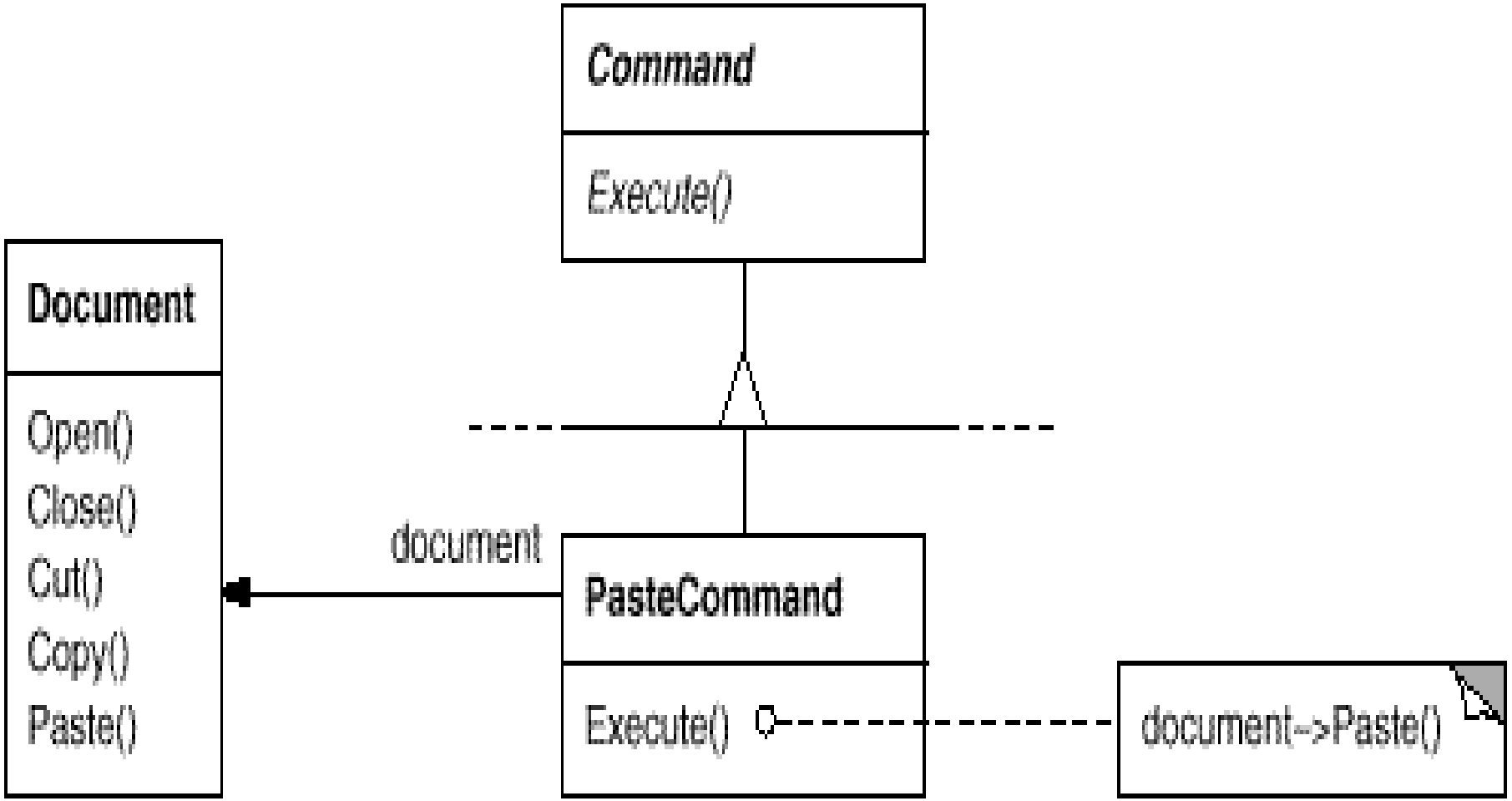
- E.g., user interface toolkits include objects like buttons and menus that carry out a request in response to user input
- Toolkit can't implement the request in the button or menu
- Only applications that use the toolkit know what should be done on which object
- Command pattern lets toolkit objects make requests of unspecified application objects by turning request itself into an object
- This object can be stored and passed around

- The key to this pattern is an abstract Command class, which declares an interface for executing operations.
- Concrete Command Subclasses specify a receiver-action pair by storing the receiver as an Instance variable and by implementing Execute to invoke the request.
- The receiver has the knowledge required to carry out the request.

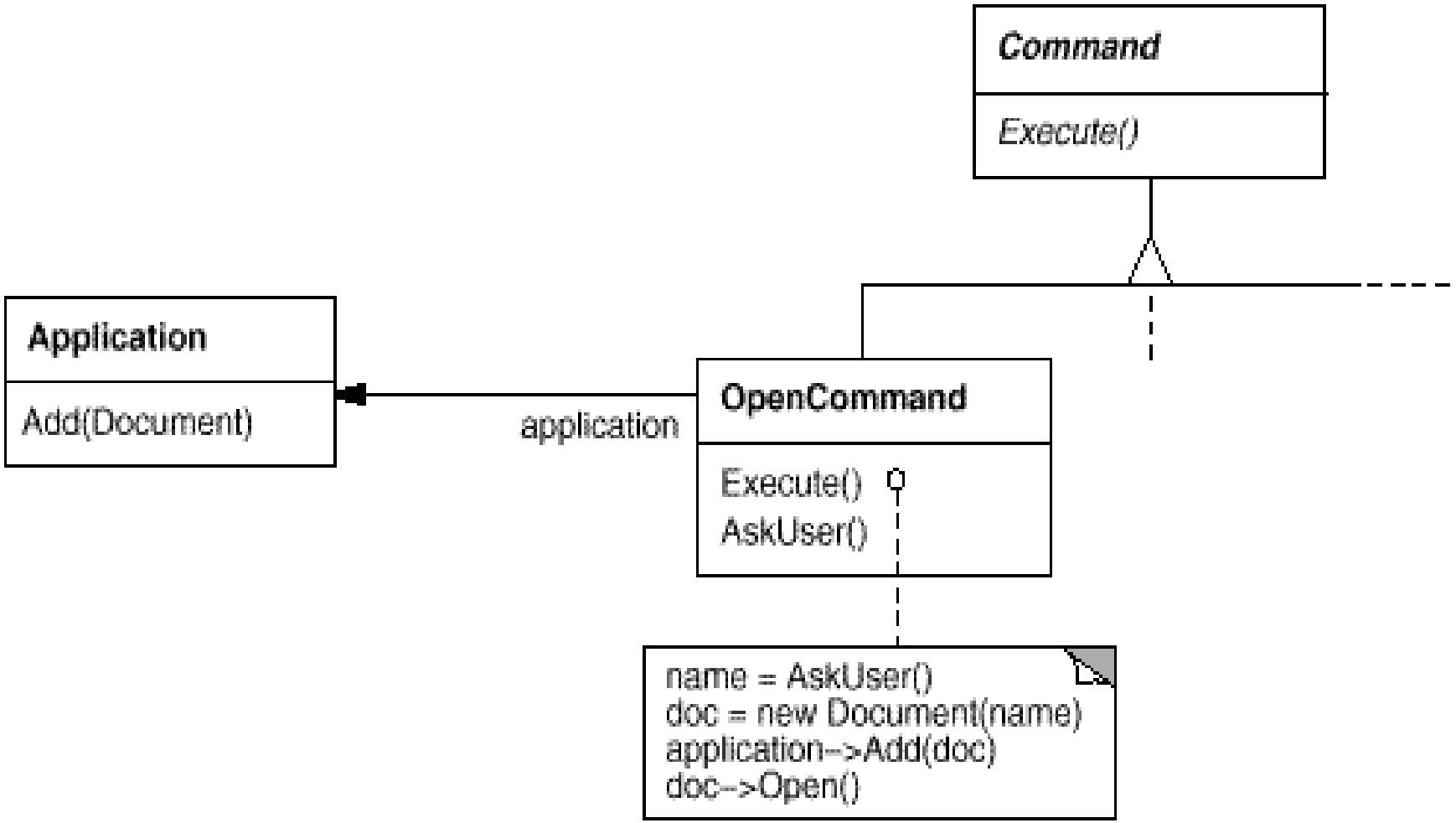

Application
Add(Document)
Menu
Add(MenuItem)
MenuItem
Clicked()
command
Command
Execute()
Document
Open()
Close()
Cut()
Copy()
Paste()
command->Execute()

- Menus can be implemented easily with Command objects
- Each choice in a Menu is an instance of a MenuItem class
- An Application class creates these menus and their menu items along with the rest of the user interface
- Application class also keeps track of Document objects that a user has opened.
- The application configures each MenuItem with an instance of a concrete Command subclass

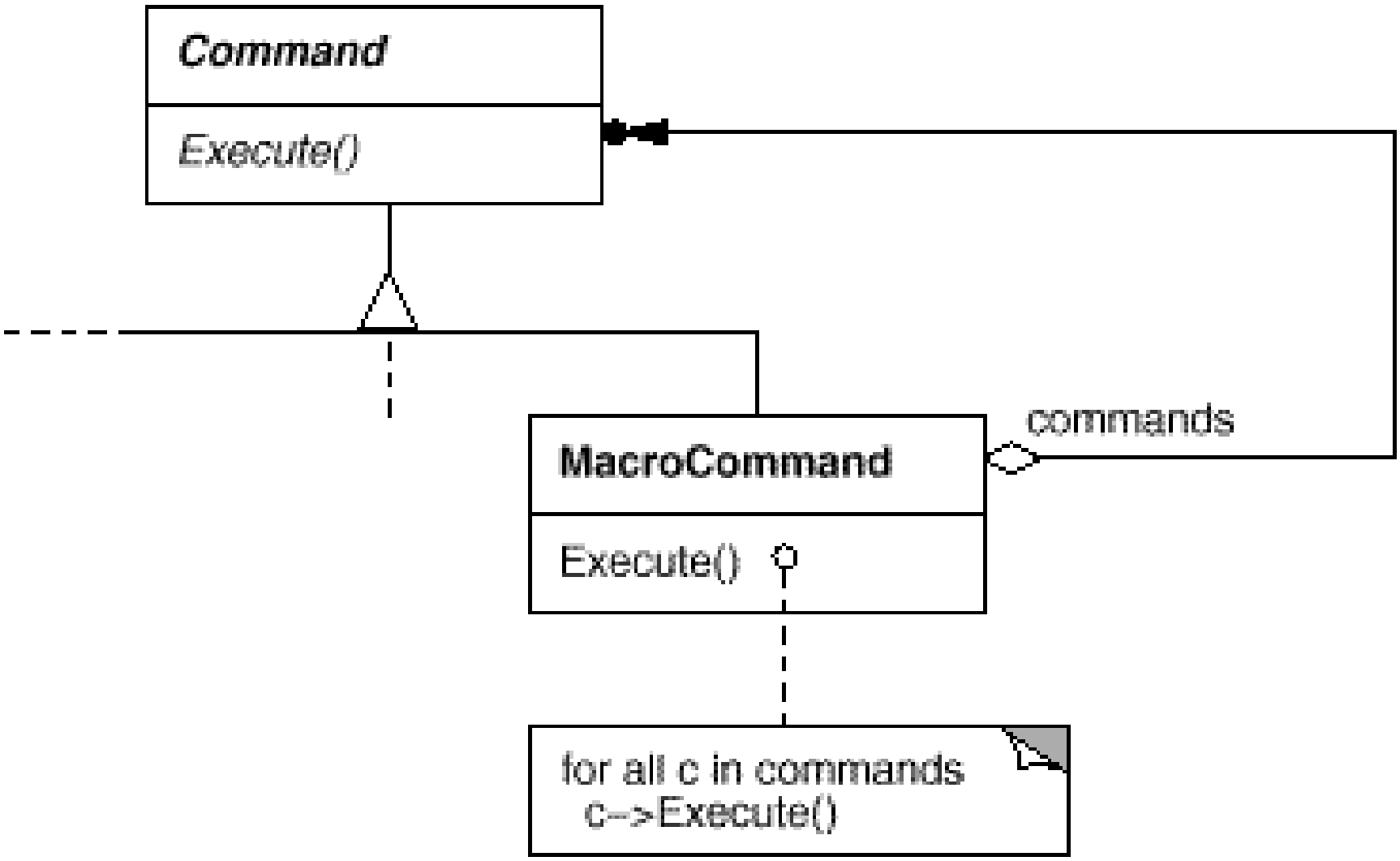
- When user selects a MenuItem, MenuItem calls Execute on its command, and Execute carries out the operation
- MenuItems don't know which subclass of Command they use
- Command subclasses store the receiver of the request and invoke one or more operations on the receiver.
- E.g., PasteCommand supports pasting text from clipboard to a Doc.
- The Execute operation invokes Paste on the receiving Document


Command
Execute()
Document
Open()
Close()
Cut()
Copy()
Paste()
document
PasteCommand
Execute()
document->Paste()

- OpenCommand's Execute operation is different:
- it prompts the user for a document name, creates a corresponding Document object, adds the document to the receiving application, and opens the document.


Command
Execute()
Application
Add(Document)
application
OpenCommand
Execute()
AskUser()
name = AskUser()
doc = new Document(name)
application->Add(doc)
doc->Open()

- Sometimes a MenuItem needs to execute a *sequence* of commands
- E.g, a MenuItem for centering a page at normal size could beconstructed from a CenterDocumentCommand object and a NormalSizeCommand object
- Because it's common to string commandstogether in this way, we can define a MacroCommand class to allow aMenuItem to execute an open-ended number of commands
- MacroCommand isa concrete Command subclass that simply executes a sequence ofCommands
- MacroCommand has no explicit receiver, because the commandsit sequences define their own receiver.


Command
Execute()
commands
MacroCommand
Execute()
for all c in commands
c->Execute()

- In all examples, notice how the Command pattern decouples the object that invokes operation from the one having the knowledge to perform it
- This gives us a lot of flexibility in designing our user interface
- Application provides menu & push button interface to a feature by making menu & push button share an instance of same concrete Command subclass
- We can replace commands dynamically, which would be useful for implementing context-sensitive menus
- All of this is possible because the object that issues a request only needs to knowhow to issue it
- It doesn't need to know how the request will be carried out.

Applicability

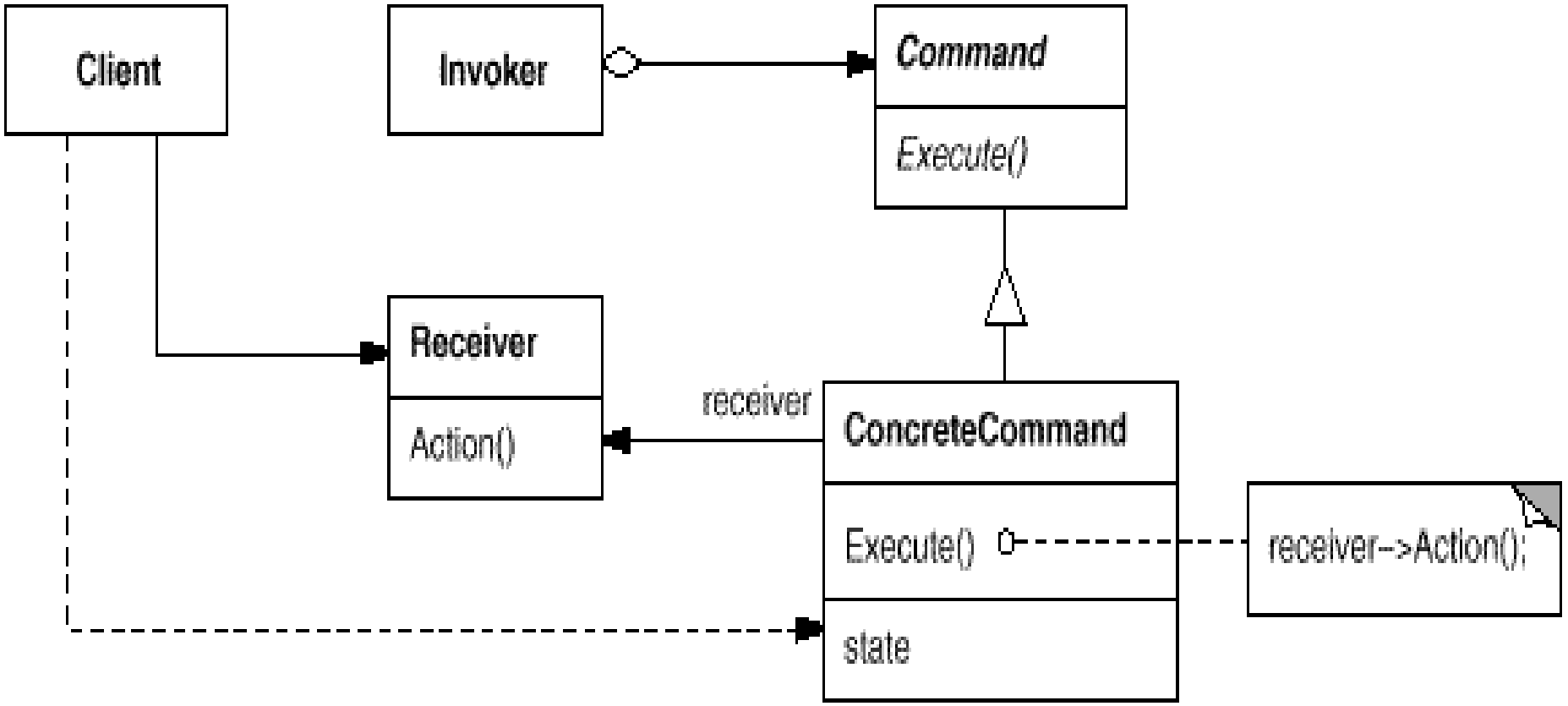
Use the Command pattern when you want to

- Parameterize objects by an action to perform, as MenuItem objects did above.
  - Such parameterization can be expressed in a language with a **callback** function, i.e, a function that's registered somewhere to be called at a later point
  - Commands are an object-oriented replacement for callbacks.

- specify, queue, and execute requests at different times.
  - A Command object can have a lifetime independent of the original request
- support undo
  - The Command's Execute operation can store state for reversing its effects in the command itself
  - Executed commands are stored in a history list

- support logging changes so that they can be reapplied in case of a system crash
  - By augmenting the Command interface with load and store operations, you can keep a persistent log of changes
- Structure a system around high-level operations built on primitives operations
  - Such a structure is common in information systems that support **transactions**.
  - The pattern also makes it easy to extend the system with new transactions.

Structure


Client
Invoker
Command
Execute()
Receiver
Action()
receiver
ConcreteCommand
Execute()
state
receiver->Action();

Participants

- **Command**
  - declares an interface for executing an operation.
- **ConcreteCommand** (PasteCommand, OpenCommand)
  - defines a binding between a Receiver object and an action.
  - implements Execute by invoking the corresponding operation(s) on Receiver.
- **Client** (Application)
  - creates a ConcreteCommand object and sets its receiver.
- **Invoker** (MenuItem)
  - asks the command to carry out the request.
- **Receiver** (Document, Application)
  - knows how to perform the operations associated with carrying out a request. Any class may serve as a Receiver.

Collaborations

- The client creates a ConcreteCommand object and specifies its receiver.
- An Invoker object stores the ConcreteCommand object
- The invoker issues a request by calling Execute on the command.
  - Whencommands are undoable, ConcreteCommand stores state for undoing thecommand prior to invoking Execute.
- The ConcreteCommand object invokes operations on its receiver to carryout the request.

Consequences

- 1. Command decouples the object that invokes the operation from the one that knows how to perform it.
- 2. Commands are first-class objects and they can be manipulated and extended like any other object.
- 3. You can assemble commands into a composite command
  - E.g., is theMacroCommand class described earlier. In general, composite commandsare an instance of the Composite pattern.
- 4. It's easy to add new Commands, because you don't have to changeexisting classes.

Interpreter

Intent

- For a language, define a representation for its grammar along with an interpreter that uses representation to interpret sentences in the language

Motivation

- If a problem occurs often, then it might be worthwhile to express instances of the problem as sentences in a simple language
- An interpreter can be built that solves problem by interpreting the sentences
- E.g., searching for strings that match a pattern is a common problem

- Regular expressions are a standard language for specifying patterns of strings
- Rather than building custom algorithms to match each pattern against strings, search algorithms could interpret a RE that specifies a set of strings to match
- The Interpreter pattern describes how to define a grammar for simple languages, represent sentences in the language, and interpret these sentences
- In this example, the pattern describes how to define a grammar for regular expressions, represent a particular regular expression, and how to interpret that regular expression

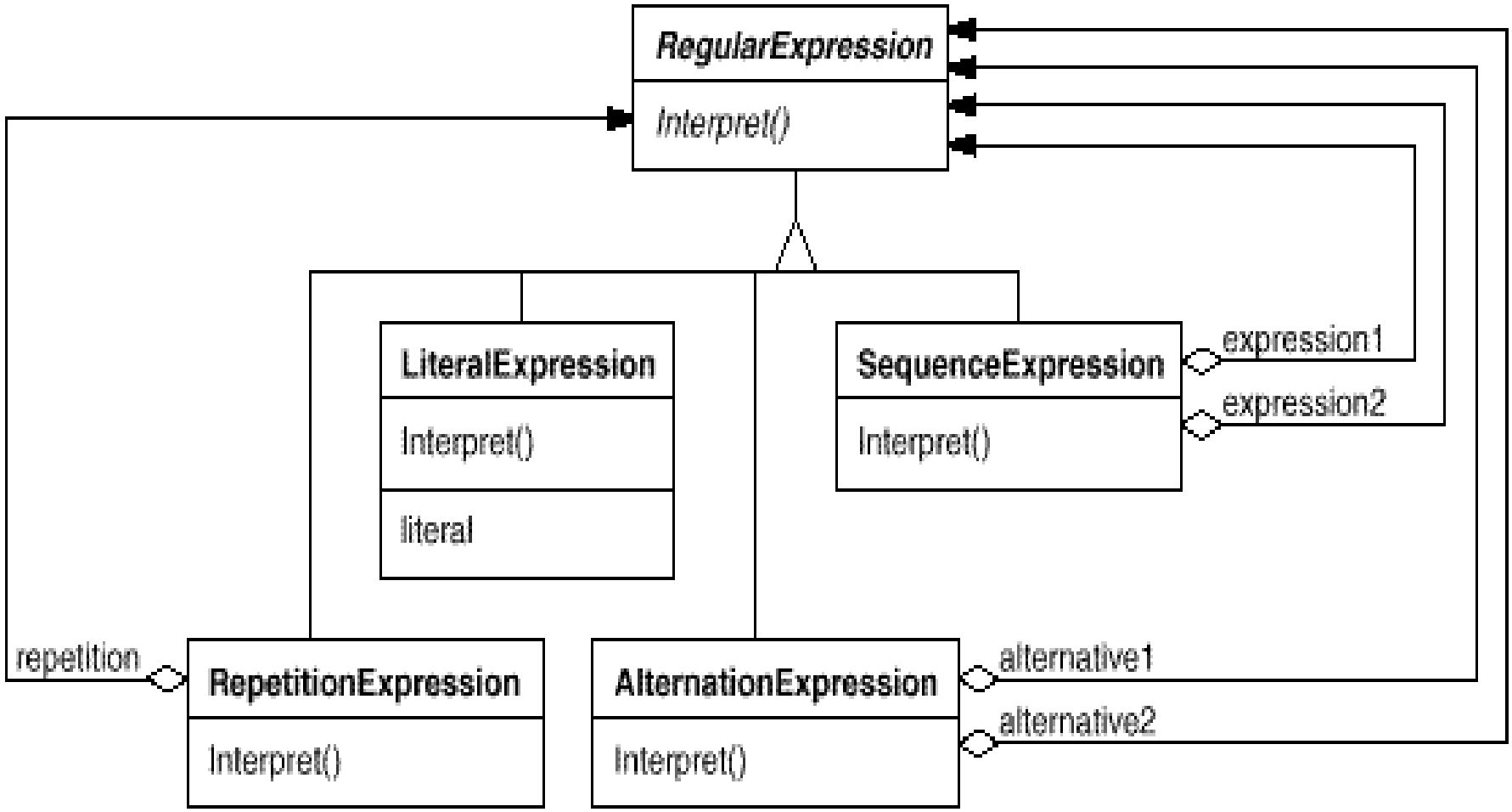
Suppose the following grammar defines the regular expressions:

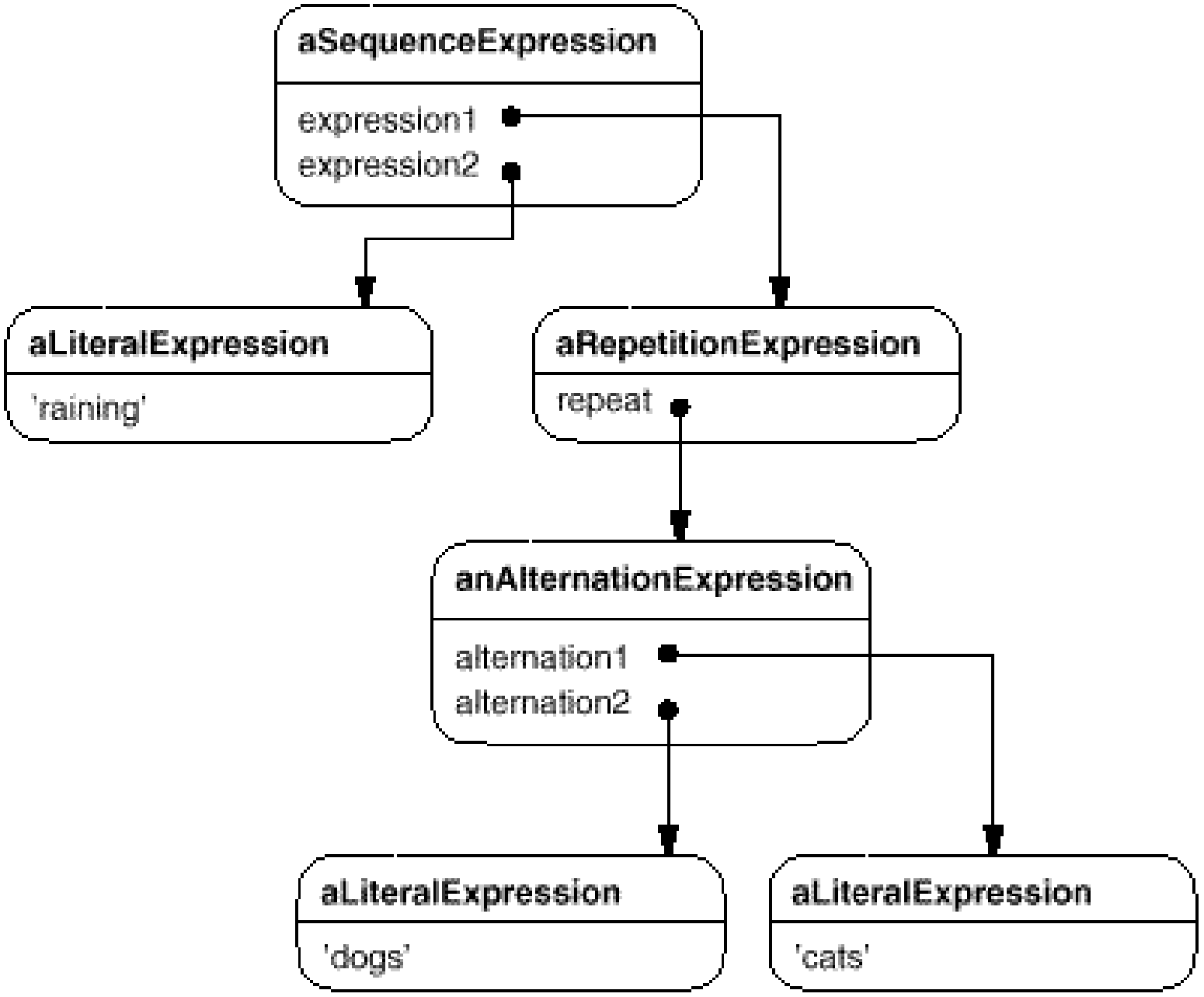
- expression ::= literal | alternation | sequence | repetition | '('expression')'
- alternation ::= expression '|' expression
- sequence ::= expression '&' expression
- repetition ::= expression '*'
- literal ::= 'a' | 'b' | 'c' | ... { 'a' | 'b' | 'c' | ... }*
- The symbol expression is the start symbol, and literal is a terminal symbol defining simple words.
- Interpreter pattern uses a class to represent each grammar rule.

- Symbols on the right-hand side of the rule are instance variables of these classes.
- The grammar above is represented by five classes: an abstract class RE and its four subclasses LiteralExpression, AlternationExpression, SequenceExpression, and RepetitionExpression
- The last three classes define variables that hold sub-expressions
- Every regular expression defined by this grammar is represented by an abstract syntax tree made up of instances of these classes.
- E.g., the abstract syntax tree represents the regular expression raining & (dogs | cats) *

- We can create an interpreter for these RE by defining the Interpret operation on each subclass of RE
- Interpret takes as an argument the context in which to interpret the expression
- The context contains the input string and information on how much of it has been matched so far
- Each subclass of RE implements Interpret to match the next part of the input string based on the current context

- For example,
  - Literal Expression will check if the input matches the literal it defines
  - Alternation Expression will check if the input matches any of its alternatives,
  - Repetition Expression will check if the input has multiple copies of expression it repeats, and so on.


RegularExpression
Interpret()
LiteralExpression
Interpret()
literal
SequenceExpression
Interpret()
expression1
expression2
repetition
RepetitionExpression
Interpret()
AlternationExpression
Interpret()
alternative1
alternative2


aSequenceExpression
expression1
expression2
aLiteralExpression
'raining'
aRepetitionExpression
repeat
anAlternationExpression
alternation1
alternation2
aLiteralExpression
'dogs'
aLiteralExpression
'cats'

Applicability

The Interpreter pattern works best when

- the grammar is simple
  - For complex grammars, the class hierarchy for the grammar becomes large and unmanageable
  - parser generators are a better alternative in such cases
  - They can interpret expressions without building abstract syntax trees, which can save space and possibly time.
- efficiency is not a critical concern
  - Most efficient interpreters are *not* implemented by interpreting parse trees directly in general but by first translating them into another form

Participants

- **AbstractExpression** (RegularExpression)
  - declares an abstract Interpret operation that is common to all nodes in the abstract syntax tree.
- **TerminalExpression** (LiteralExpression)
  - implements an Interpret operation associated with terminal symbols in the grammar.
  - an instance is required for every terminal symbol in a sentence.
- **NonterminalExpression** (AlternationExpression, RepetitionExpression, SequenceExpressions)
  - one such class is required for every rule *R* ::= *R*1 *R*2 ... *R*n in the grammar.

- maintains instance variables of type AbstractExpression for each of the symbols *R*1 through *R*n.
- implements an Interpret operation for nonterminal symbols in the grammar.
- Interpret typically calls itself recursively on the variables representing *R*1 through *R*n.

- **Context**
  - contains information that's global to the interpreter.

Client

- builds an abstract syntax tree representing a particular sentence in the language that the grammar defines
  - The abstract syntax tree is assembled from instances of the NonterminalExpression and TerminalExpression classes.
- invokes the Interpret operation.

Collaborations

- The client builds (or is given) the sentence as an abstract syntaxtree of NonterminalExpression and TerminalExpression instances.
  - Then the client initializes the context and invokes the Interpret operation.
- Each NonterminalExpression node defines Interpret in terms ofInterpret on each sube-xpression
  - The Interpret operation of eachTerminalExpression defines the base case in the recursion.
- The Interpret operations at each node use the context to store and access the state of the interpreter.

Consequences

The Interpreter pattern has the following benefits and liabilities:

- 1. *It's easy to change and extend the grammar, b*ecause the pattern uses classes to represent grammar rules, you can use inheritance to change or extend the grammar
  - Existing expressions can be modified incrementally, and new expressions can be defined as variations on old ones.
- 2. *Implementing the grammar is easy, too*
  - Classes defining nodes in the abstract syntax tree have similar implementations
  - These classes are easy to write, and often their generation can be automated with a compiler or parser generator.

- 3. *Complex grammars are hard to maintain*
  - The Interpreter pattern defines at least one class for every rule in the grammar (grammar rules defined using BNF may require multiple classes)
  - Hence grammars containing many rules can be hard to manage and maintain
  - Other design patterns can be applied to mitigate the problem (see Implementation)
  - But when the grammar is very complex, other techniques such as parser or compiler generators are more appropriate.
- 4. *Adding new ways to interpret expressions*
  - The Interpreter pattern makes it easier to evaluate an expression in anew way
  - E,g., you can support pretty printing or type-checking an expression by defining a new operation on the expression classes
  - If you keep creating new ways of interpreting an expression, then consider using the Visitor pattern to avoid changing the grammar classes

Implementation

The following issues are specific to Interpreter:

1. *Creating the abstract syntax tree.*

- Interpreter pattern doesn't explain how to *create* an abstract syntax tree The abstract syntax tree can be created by a table-driven parser

2. *Defining the Interpret operation.*

- If it's common to create a new interpreter, then it's better to use the Visitor pattern to put Interpret in a separate "visitor" object
- E.g., a grammar for a programming language will have many operations on abstract syntax trees, such as type-checking, optimization, code generation, etc.

3. *Sharing terminal symbols with the Flyweight pattern*

- Grammars whose sentences contain many occurrences of a terminal symbol might benefit from sharing a single copy of that symbol
- Grammars for computer programs are good examples, each program variable will appear in many places throughout the code

Known Uses

- Interpreter pattern is used in compilers implemented with O-O languages, as the Smalltalk compilers are.
- Considered in its most general form, every use of Composite pattern will also contain Interpreter pattern
- But Interpreter pattern should be reserved for those cases in which you want to think of the class hierarchy as defining a language.

Iterator (Cursor)

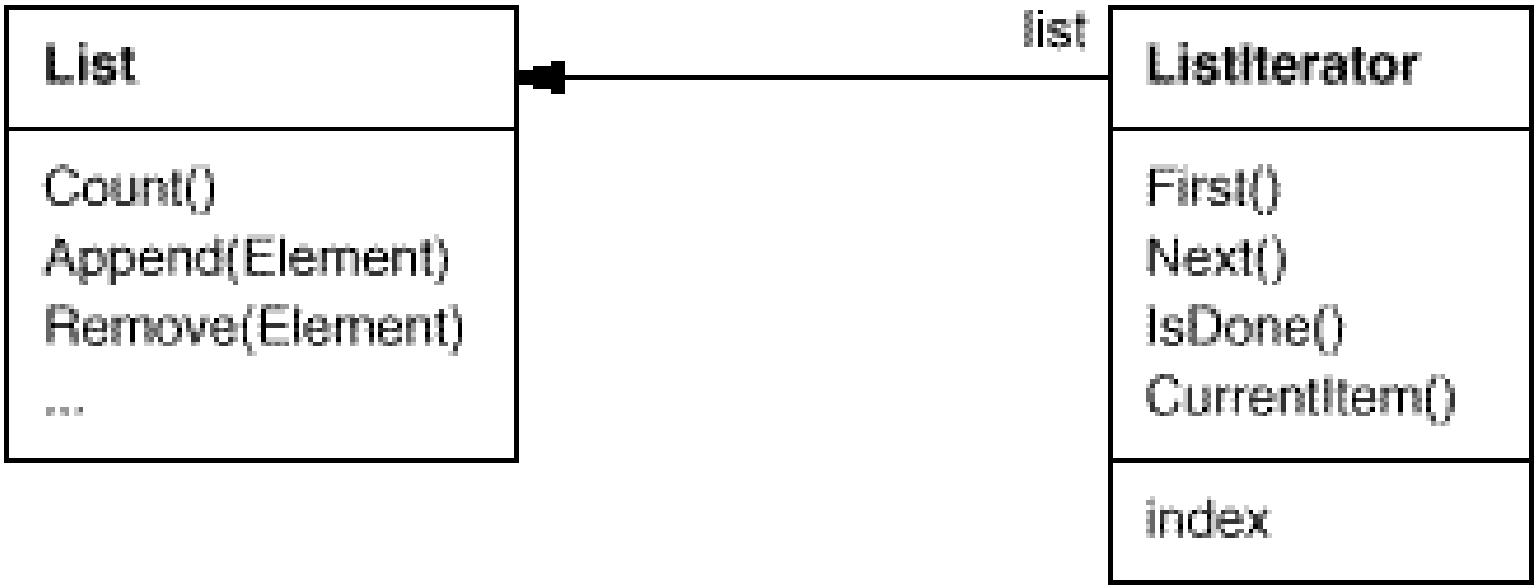
Intent

- Provide a way to access elements of an aggregate object sequentially without exposing its underlying representation

Motivation

- An aggregate object such as a list should give a way to access its elements without exposing its internal structure
- User may want to traverse list in different ways, depending on his needs
- But he don't want to expand 'List' interface with operations for different traversals
- User may also need to have more than one traversal pending on same list

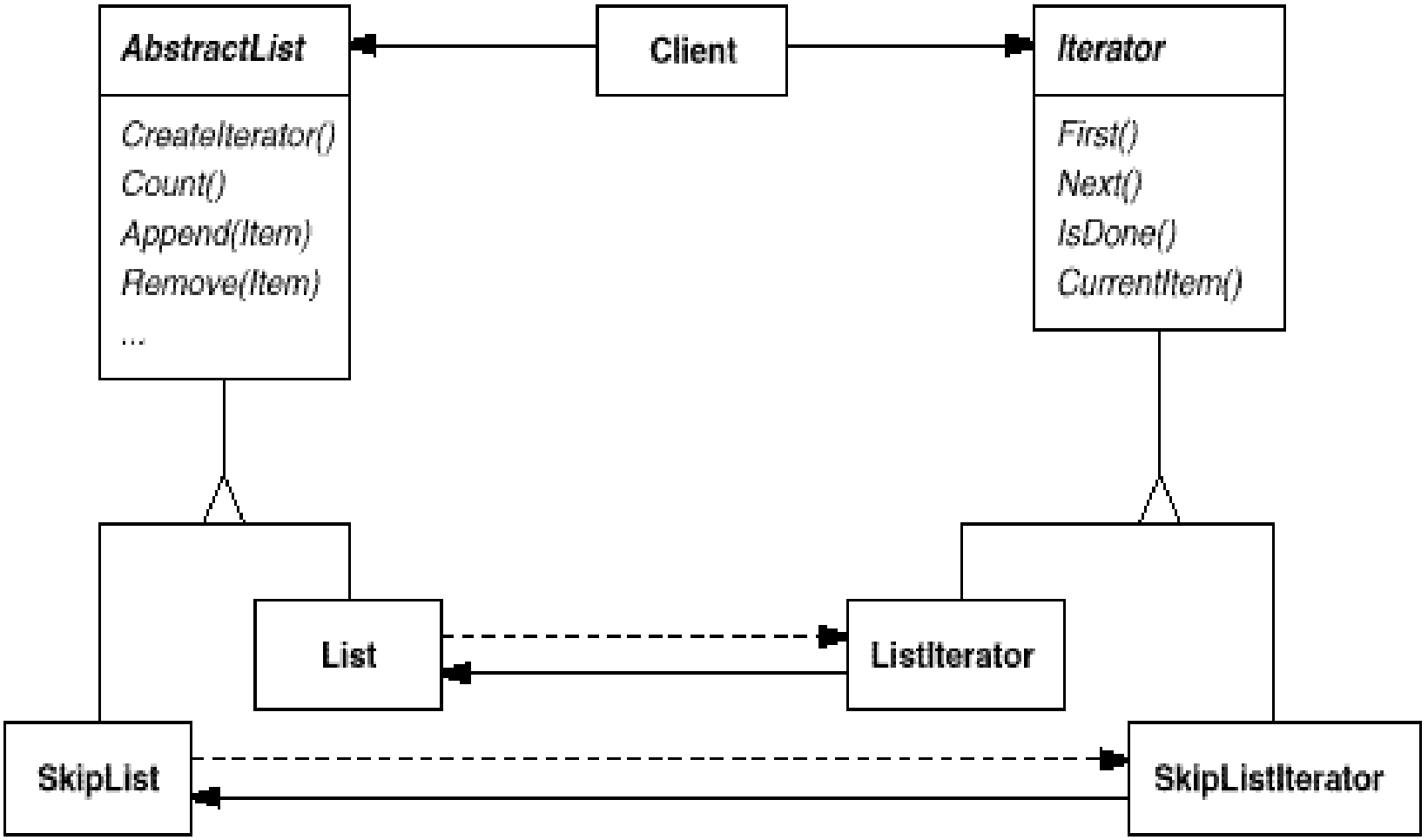
- Iterator pattern lets you do all this
- Key idea is to take responsibility for access & traversal out of list object & put it into an **iterator** object
- Iterator class defines an interface to access list's elements
- An iterator object is responsible for history of current element; i.e., it knows which elements have been traversed already
- E.g., a List class would call for a ListIterator with the following relationship between them:


List
Count()
Append(Element)
Remove(Element)
...
list
ListIterator
First()
Next()
IsDone()
CurrentItem()
index

- Before instantiating ListIterator, it must have a List for traversing
- ‘CurrentItem’ operation returns current element in the list, First initializes the current element to the first element
- ‘Next’ advances current element to the next element
- ‘IsDone’ tests whether we've advanced beyond the last element, i.e., we're finished with the traversal
- Separating the traversal mechanism from the List object lets us define iterators for different traversal policies without enumerating them in the List interface.
- E.g., FilteringListIterator might provide access only to those elements that satisfy specific filtering constraints

- Notice that the iterator and the list are coupled, and the client must know that it is a *list* that's traversed as opposed to some other aggregate structure
- Hence the client commits to a particular aggregate structure
- It would be better if we could change the aggregate class without changing client code
- We can do this by generalizing the iterator concept to support **polymorphic iteration**
- As an example, let's assume that we also have a SkipList implementation of a list.

- A skip list is a probabilistic data structure with characteristics similar to balanced trees
- Write code that works for both List and SkipList objects
- Define an AbstractList class that provides a common interface for manipulating lists
- An abstract Iterator class that defines a common iteration interface is needed
- Define concrete Iterator subclasses for different list implementations
- Hence, iteration mechanism becomes independent of concrete aggregate classes

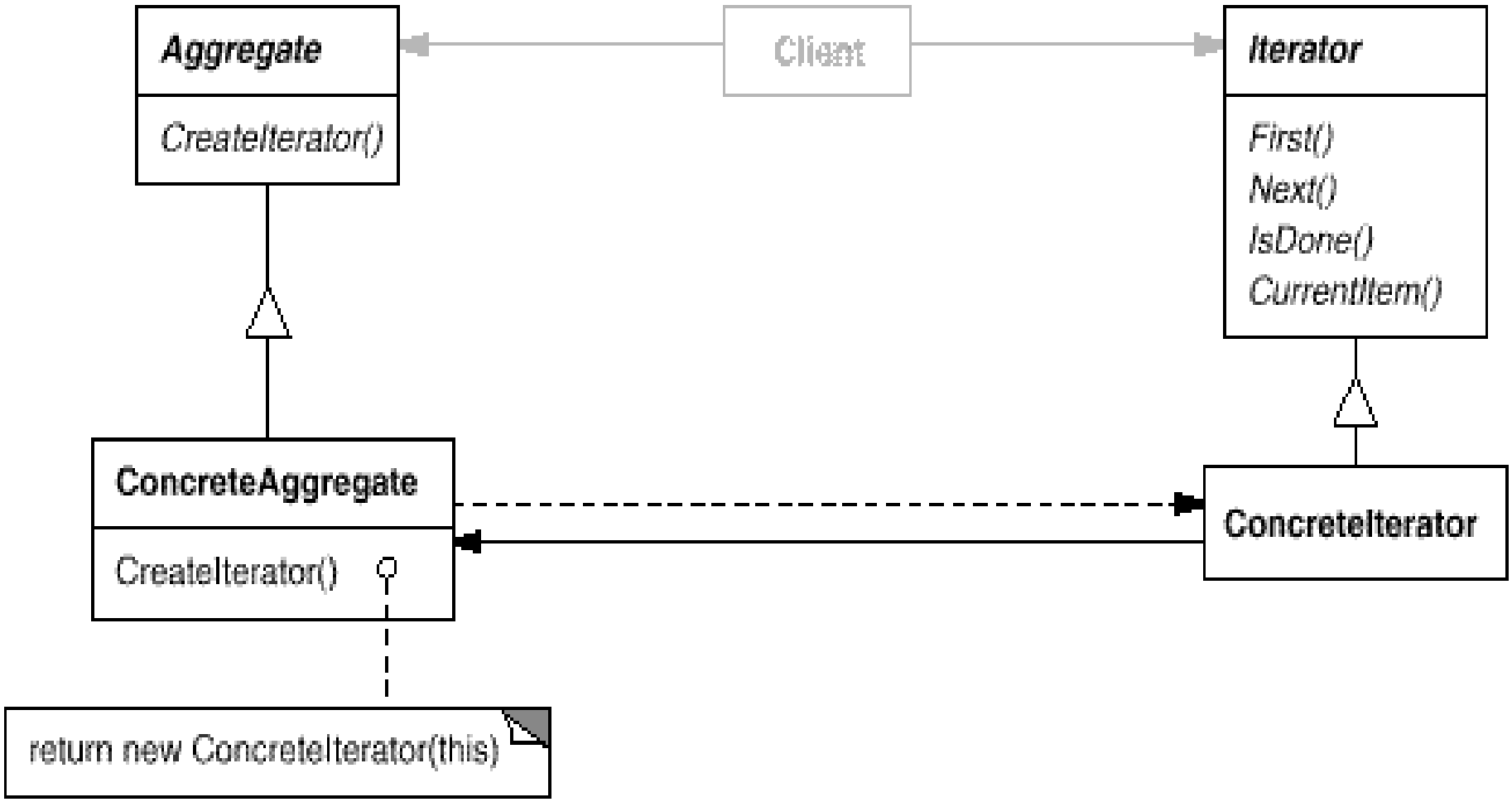

AbstractList
CreateIterator()
Count()
Append(Item)
Remove(Item)
...
Client
Iterator
First()
Next()
IsDone()
CurrentItem()
List
ListIterator
SkipList
SkipListIterator

- The remaining problem is how to create the iterator
- Since we want to write code that's independent of the concrete List subclasses, we cannot simply instantiate a specific class
- Instead, we make the list objects responsible for creating their corresponding iterator
- This requires an operation like CreateIterator through which clients request an iterator object
- CreateIterator is an example of a factory method
- We use it here to let a client ask a list object for the appropriate iterator
- The Factory Method approach give rise to two class hierarchies, one for lists and another for iterators
- CreateIterator factory method "connects" the two hierarchies.

Applicability

Use the Iterator pattern

- to access an aggregate object's contents without exposing its internal representation
- to support multiple traversals of aggregate objects
- to provide a uniform interface for traversing different aggregate structures


Aggregate
CreateIterator()
Client
Iterator
First()
Next()
IsDone()
CurrentItem()
ConcreteAggregate
CreateIterator()
ConcreteIterator
return new ConcreteIterator(this)

Structure

Participants

- **Iterator**
  - defines an interface for accessing and traversing elements.
- **ConcreteIterator**
  - implements the Iterator interface.
  - keeps track of the current position in the traversal of the aggregate.
- **Aggregate**
  - defines an interface for creating an Iterator object.
- **ConcreteAggregate**
  - implements the Iterator creation interface to return an instance of the proper ConcreteIterator

Collaborations

- ConcreteIterator keeps track of current object in the aggregate and can compute the succeeding object in the traversal.

Consequences

The Iterator pattern has three important consequences:

- 1. *It supports variations in the traversal of an aggregate.*
  - Complex aggregates may be traversed in many ways
  - E.g., code generation & semantic checking involve traversing parse trees
  - Code generation may traverse the parse tree inorder or preorder
  - Iterators make it easy to change the traversal algorithm: Just replace the iterator instance with a different one
  - You can also define Iterator subclasses to support new traversals.

- 2. *Iterators simplify the Aggregate interface*
  - Iterator's traversal interface obviates the need for a similar interface in Aggregate, thereby simplifying the aggregate's interface.
- 3. *More than one traversal can be pending on an aggregate*
  - An iterator keeps track of its own traversal state
  - Therefore you can have more than one traversal in progress at once

Mediator

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
- Moderator lets you vary their interaction independently.

Motivation

- O-O design encourages the distribution of behaviour among objects
- Such distribution can result in an object structure with many connections between objects;
- In the worst case, every object ends up knowing about every other.

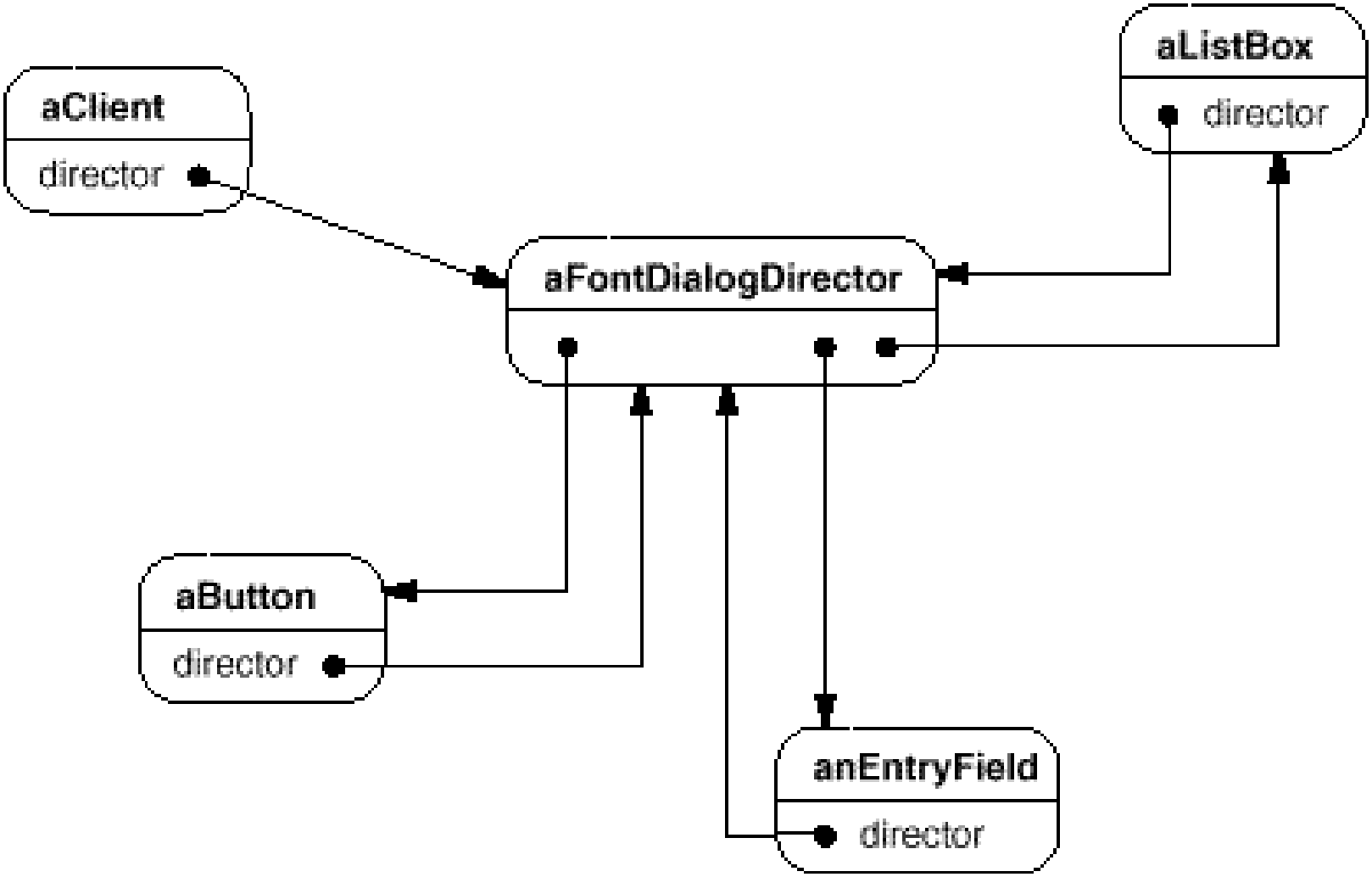
- Though partitioning a system into many objects enhances reusability, expanding interconnections tend to reduce it again
- Lots of interconnections make it less likely that an object can work without the support of others—the system acts as though it were monolithic
- It can be difficult to change the system's behaviour in any significant way, since behaviour is distributed among many objects
- Hence, you may be forced to define many subclasses to customize the system's behaviour.
- As an e.g., consider the implementation of dialog boxes in a GUI


Font Chooser
The quick brown fox...
Family
new century schoolbook
avant garde
chicago
courier
helvetica
new century schoolbook
palatino
times roman
zapf dingbats
Weight
medium
bold
demibold
Slant
roman
italic
oblique
Size
34 pt
condensed
Cancel
OK

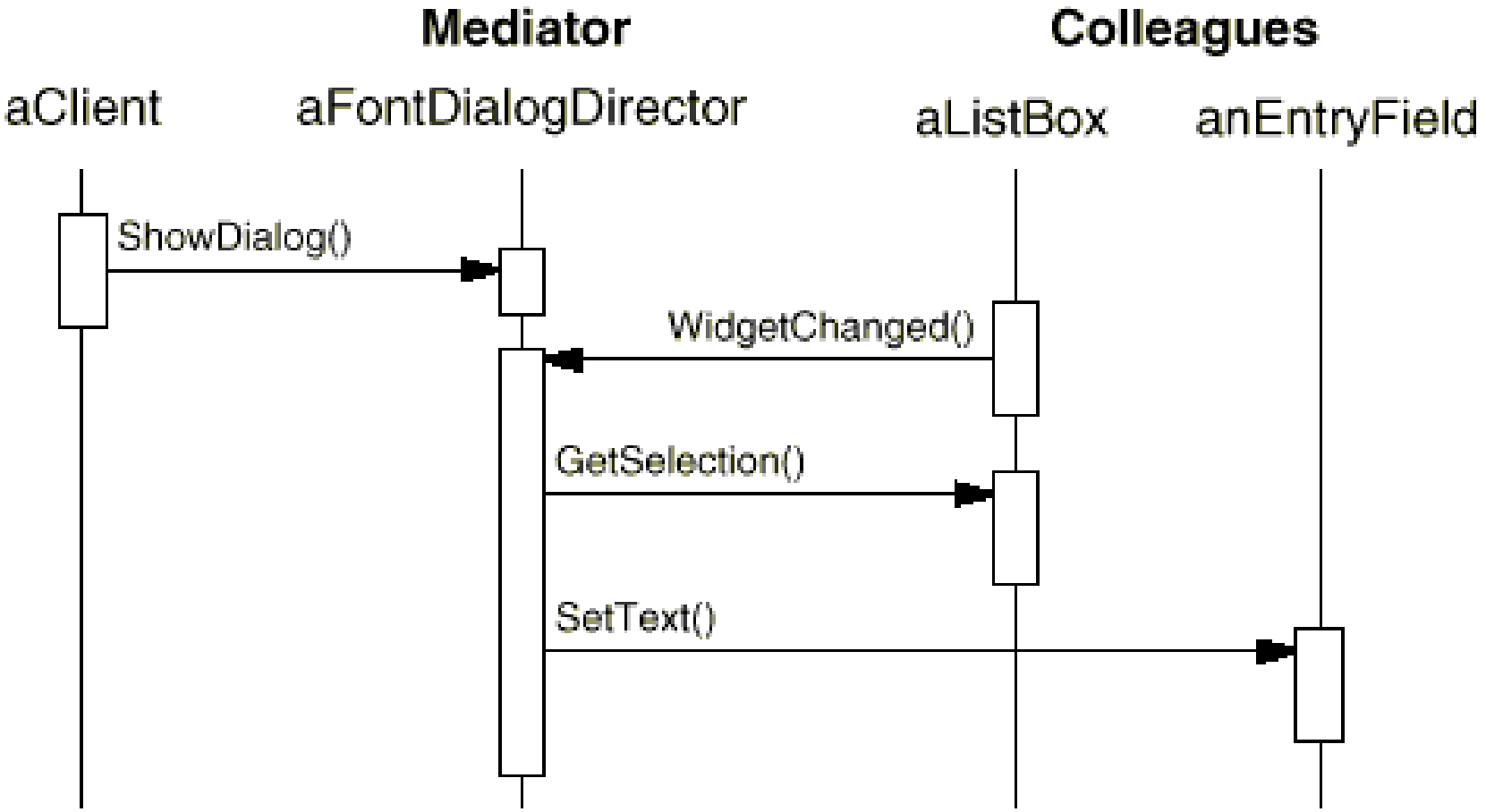
- A dialog box uses a window to present a collection of widgets such as buttons, menus, entry fields, as in fig.
- Often there are dependencies between the widgets in the dialog.
- E.g., a button gets disabled when a certain entry field is empty
- Selecting an entry in a list of choices called a **list box** might change the contents of an entry field.
- Conversely, typing text into the entry field might automatically select one or more corresponding entries in the list box
- Once text appears in the entry field, other buttons may become enabled that let the user do something with the text, such as changing or deleting the thing to which it refers

- Different dialog boxes will have different dependencies between widgets
- So even though dialogs display the same kinds of widgets, they can't simply reuse stock widget classes; they have to be customized to reflect dialog-specific dependencies.
- Customizing them individually by subclassing will be tedious, since many classes are involved.
- You can avoid these problems by encapsulating collective behaviour in a separate **mediator** object

- A mediator is responsible for controlling and coordinating the interactions of a group of objects
- Mediator serves as an intermediary that keeps objects in the group from referring to each other explicitly
- Objects only know mediator, by reducing the no.of interconnections
- E.g., **FontDialogDirector** is the mediator bet. widgets in a dialog box
- A FontDialogDirector object knows the widgets in a dialog and coordinates their interaction
- It acts as a hub of communication for widgets:


aClient
director
aListBox
director
aFontDialogDirector
aButton
director
anEntryField
director

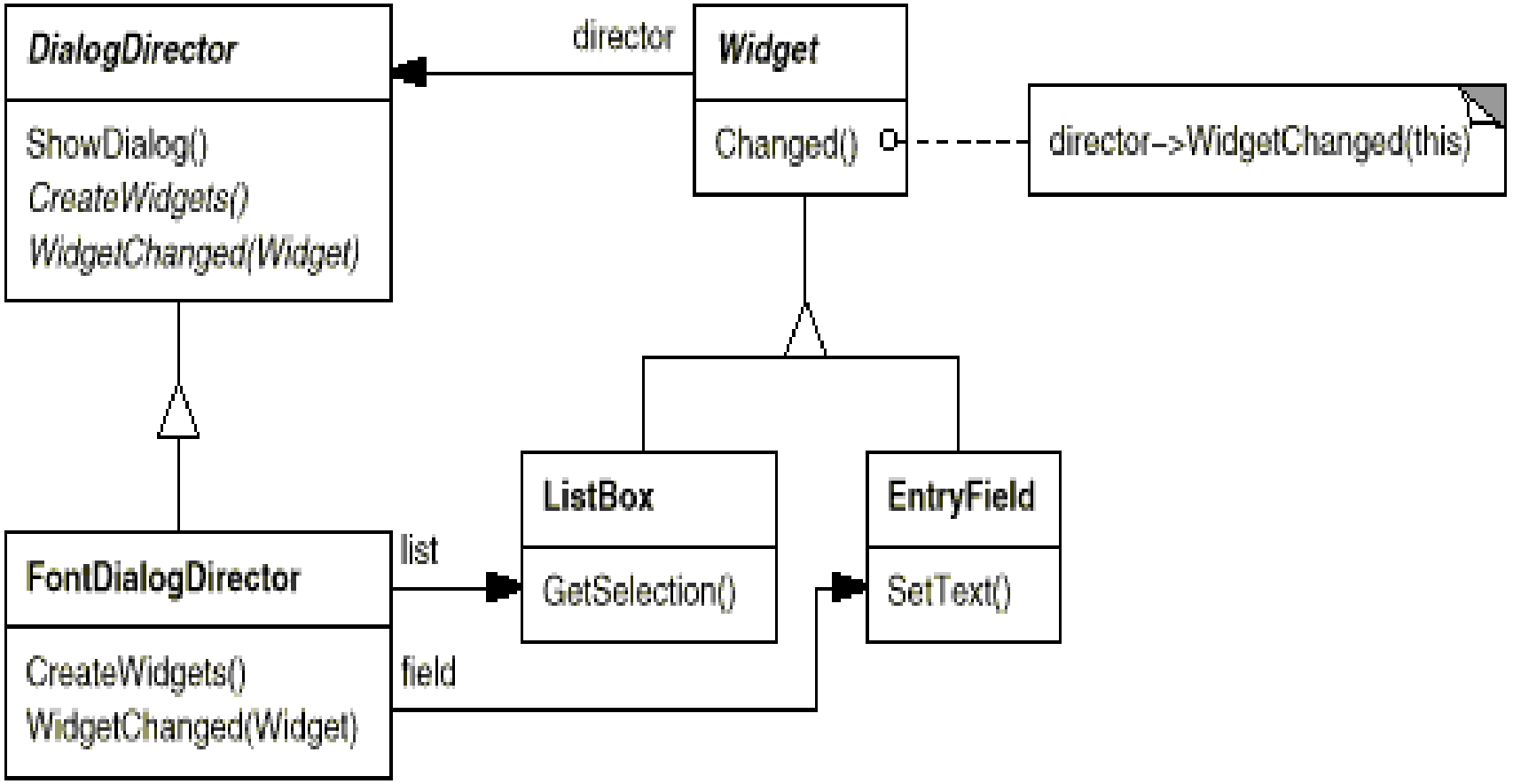
Following interaction diagram illustrates how the objects cooperate to handle a change in a list box's selection:



Here's the succession of events by which a list box's selection passes to an entry field:

- 1. list box tells its director that it's changed.
- 2. director gets the selection from the list box.
- 3. 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
  - e.g., "demibold," "oblique"

- How director mediates between list box & entry field
- Widgets communicate with each other only indirectly, through the director
- They don't have to know each other; all they know is director
- Since the behavior is localized in one class, it can be changed or replaced by extending or replacing that class
- Here's how the FontDialogDirector abstraction can be integrated into aclass library:


DialogDirector
ShowDialog()
CreateWidgets()
WidgetChanged(Widget)
director
Widget
Changed()
director->WidgetChanged(this)
ListBox
GetSelection()
EntryField
SetText()
FontDialogDirector
CreateWidgets()
WidgetChanged(Widget)
list
field

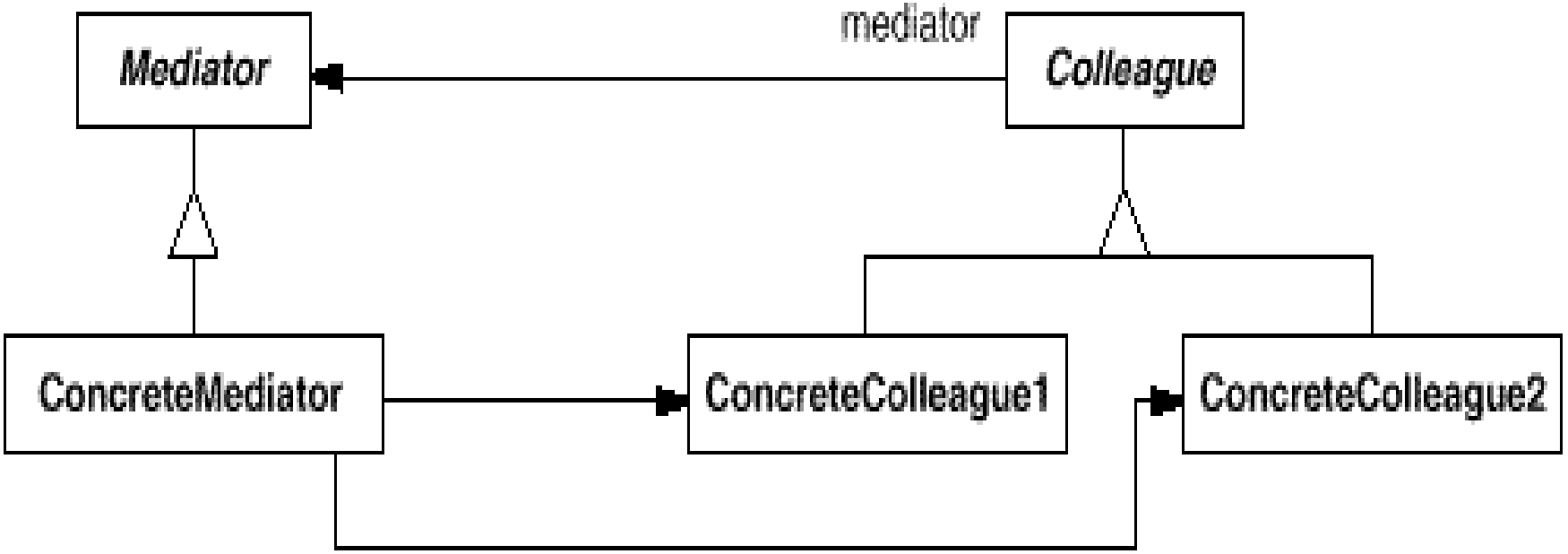
- DialogDirector is an abstract class that defines the overall behavior of a dialog.
- Clients call ShowDialog operation to display dialog on screen
- CreateWidgets is an abstract operation for creating the widgets of a dialog.
- WidgetChanged is another abstract operation; widgets call it to inform their director that they have changed
- DialogDirector subclasses override CreateWidgets to create proper widgets, & they override WidgetChanged to handle changes

Applicability

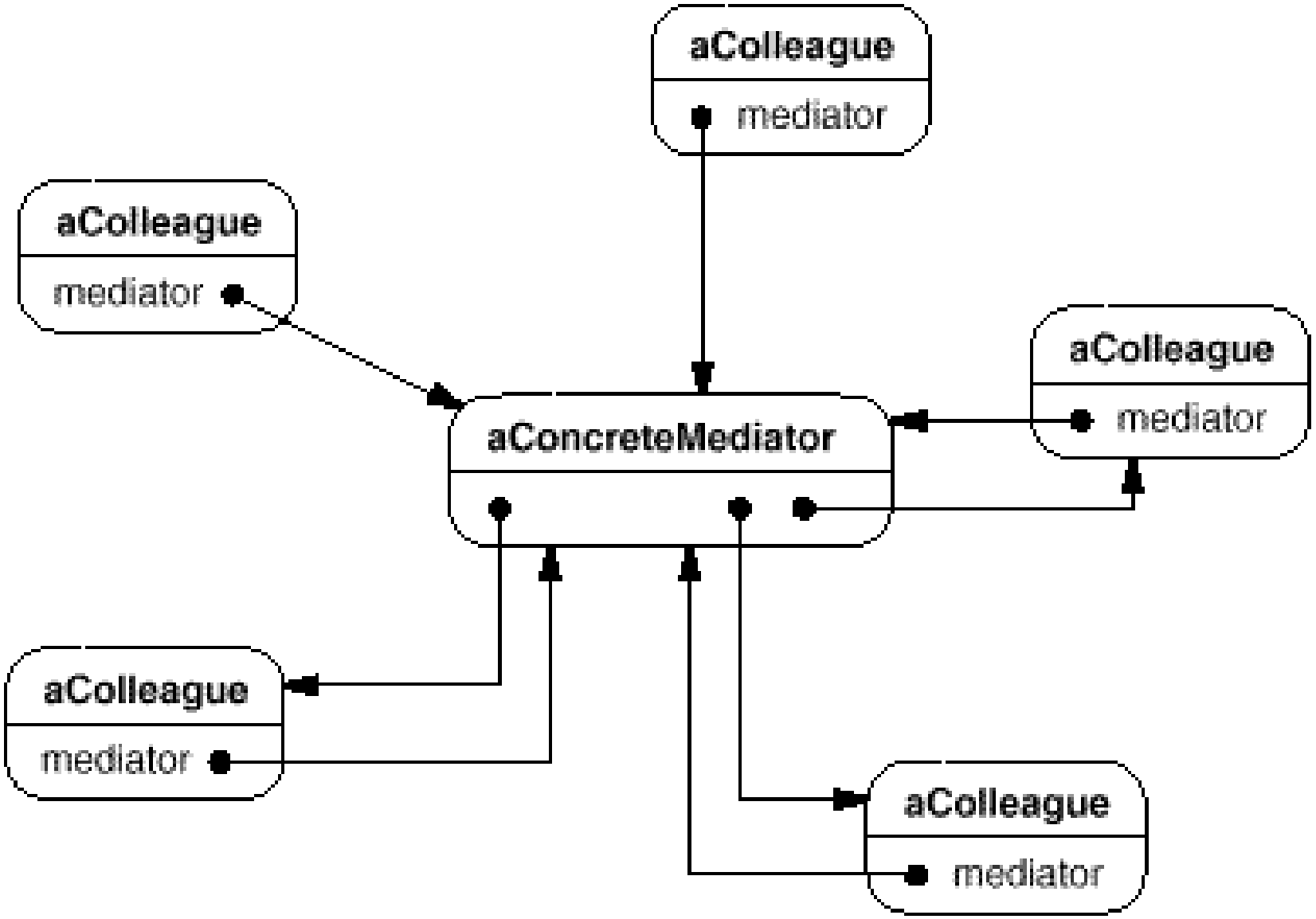
Use the Mediator pattern when

- a set of objects communicate in well-defined but complex ways
  - The resulting interdependencies are unstructured and difficult to understand.
- reusing an object is difficult 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

Structure


Mediator
mediator
Colleague
ConcreteMediator
ConcreteColleague1
ConcreteColleague2

A typical object structure might look like this



Participants

- **Mediator** (DialogDirector)
  - defines an interface for communicating with Colleague objects.
- **ConcreteMediator** (FontDialogDirector)
  - implements cooperative behavior by coordinating Colleague objects
  - knows and maintains its colleagues.
- **Colleague classes** (ListBox, EntryField)
  - each Colleague class knows its Mediator object.
  - each colleague communicates with its mediator whenever it would have otherwise communicated with another colleague

Collaborations

- Colleagues send and receive requests from a Mediator object. Themediator implements the cooperative behavior by routing requestsbetween the appropriate colleague(s).

Consequences

- The Mediator pattern has the following benefits and drawbacks:
  - 1. *It limits subclassing*
    - A mediator localizes behavior that otherwise would be distributed amongseveral objects.
    - Changing this behavior requires subclassing Mediatoronly;
    - Colleague classes can be reused as is.

- 2. *It decouples colleagues*
  - A mediator promotes loose coupling between colleagues
  - You can vary and reuse Colleague and Mediator classes independently.
- 3. *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

- *4. It 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.
- 5. *It centralizes control.* The Mediator pattern 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.

Implementation

Following implementation issues are relevant to the Mediator pattern:

- 1. *Omitting the abstract Mediator class*
  - There's no need to define an abstract Mediator class when colleagues work with only one mediator
  - The abstract coupling that the Mediator class provides lets colleagues work with different Mediator subclasses, and vice versa.
- 2. *Colleague-Mediator communication*
  - Colleagues have to communicate with their mediator when an event of interest occurs
  - One approach is to implement the Mediator as an Observer using the Observer pattern

- Colleague classes act as Subjects, sending notifications to the mediator whenever they change state
- Mediator responds by propagating the effects of the change to other colleagues
- Another approach defines a specialized notification interface in Mediator that lets colleagues be more direct in their communication
- Smalltalk/V for Windows uses a form of delegation: When communicating with the mediator, a colleague passes itself as an argument, allowing the mediator to identify the sender
- The Sample Code uses this approach, and the Smalltalk/V implementation is discussed further in the Known Uses.

Memento (Token)

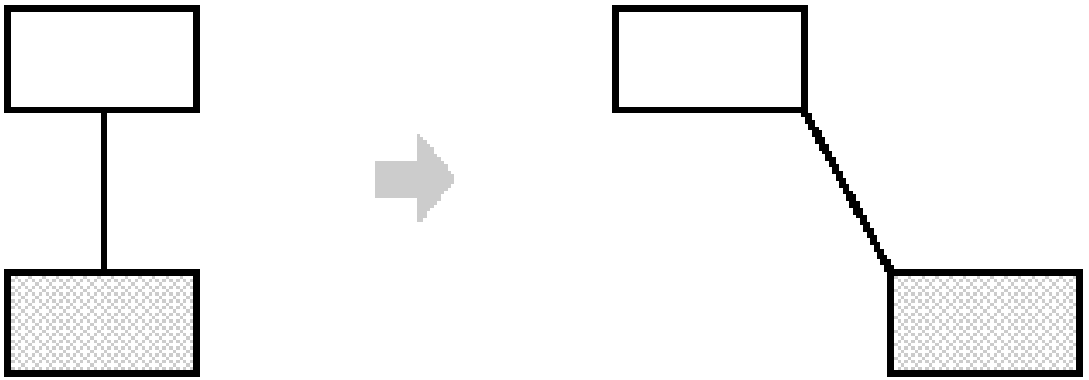
Intent

- Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later.

Motivation

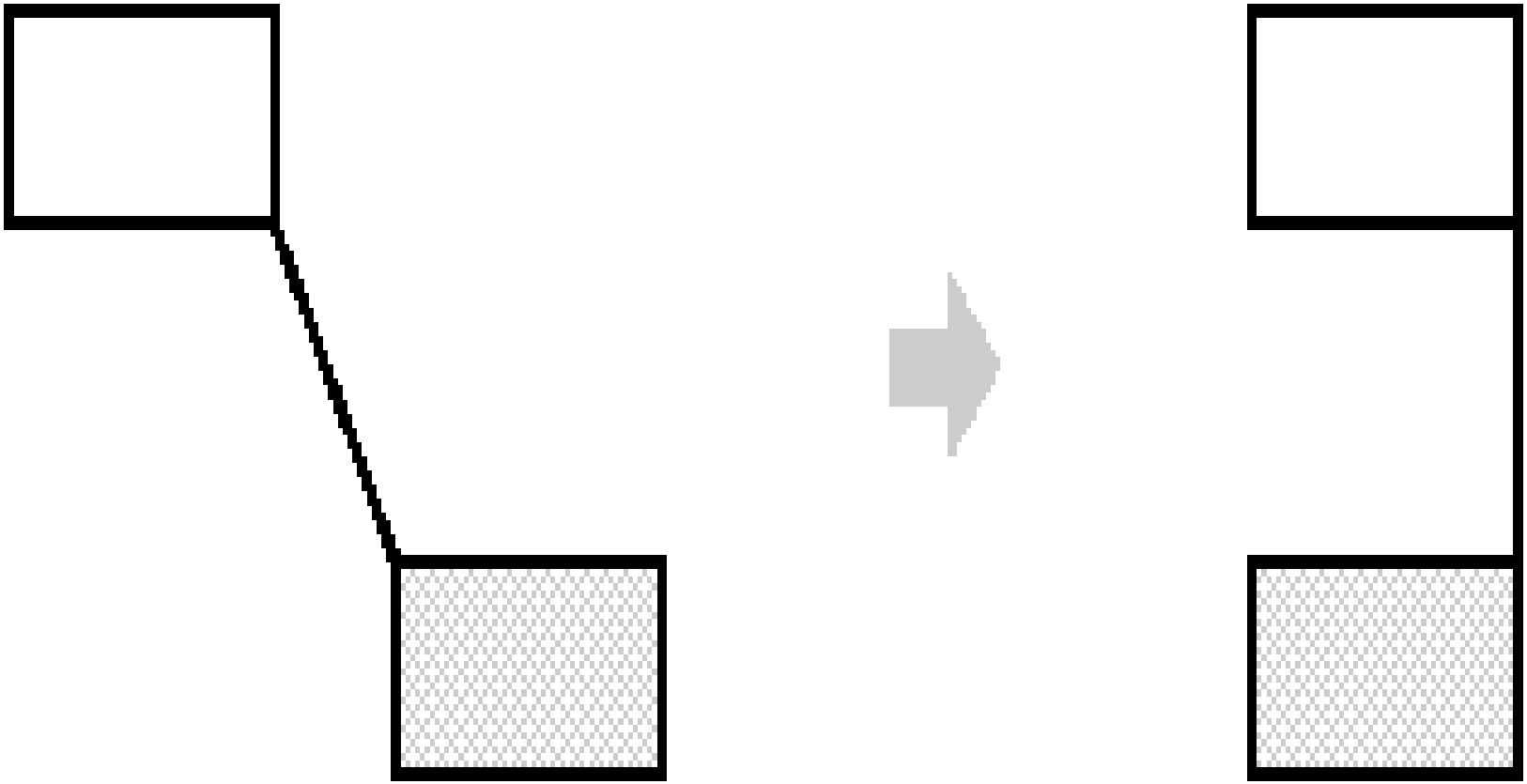
- Sometimes it's necessary to record the internal state of an object
- This is required when implementing checkpoints and undo mechanisms that let users back out of tentative operations or recover from errors
- You must save state information somewhere so that you can restore objects to their previous states

- But objects normally encapsulate some or all of their state, making it inaccessible to other objects and impossible to save externally
- Exposing this state would violate encapsulation, which can compromise the application's reliability and extensibility.
- E.g., a graphical editor that supports connectivity between objects.
- A user can connect two rectangles with a line, and the rectangles stay connected when the user moves either of them
- The editor ensures that the line stretches to maintain connection



- A well-known way to maintain connectivity relationships between objects is with a constraint-solving system
- This functionality can be encapsulated in a **ConstraintSolver** object
- ConstraintSolver records connections as they are made and generates mathematical equations that describe them
- It solves these equations whenever the user makes a connection or otherwise modifies the diagram.
- ConstraintSolver uses results of its calculations to rearrange graphics so that they maintain the proper connections.

- Supporting undo in this application isn't as easy
- An obvious way to undo a move operation is to store the original distance moved and move the object back an equivalent distance
- This does not guarantee all objects will appear where they did before
- Suppose there is some slack in the connection
- Simply moving the rectangle back to its original location won't necessarily achieve the desired effect



- In general, the ConstraintSolver's public interface might be insufficient to allow precise reversal of its effects on other objects
- The undo mechanism must work more closely with ConstraintSolver to re-establish previous state, but we should also avoid exposing the ConstraintSolver's internals to the undo mechanism.
- We can solve this problem with the Memento pattern
- A **memento** is an object that stores a snapshot of theinternal state of another object — the memento's **originator**

- The undo mechanism will request a memento from the originator when it needs to checkpoint the originator's state
- The originator initializes the memento with information that characterizes its current state
- Only the originator can store and retrieve information from the memento—the memento is "opaque" to other objects.
- In the graphical editor example just discussed, the ConstraintSolver can act as an originator

- The following sequence of events characterizes the undo process:
- 1. The editor requests a memento from the ConstraintSolver as aside-effect of the move operation.
- 2. The ConstraintSolver creates and returns a memento, an instance of a class SolverState in this case
  - A SolverState memento contains data structures that describe the current state of the ConstraintSolver's internal equations and variables.
- 3. Later when the user undoes the move operation, the editor gives the SolverState back to the ConstraintSolver.

- 4. Based on the information in the SolverState, the ConstraintSolver changes its internal structures to return its equations and variables to their exact previous state

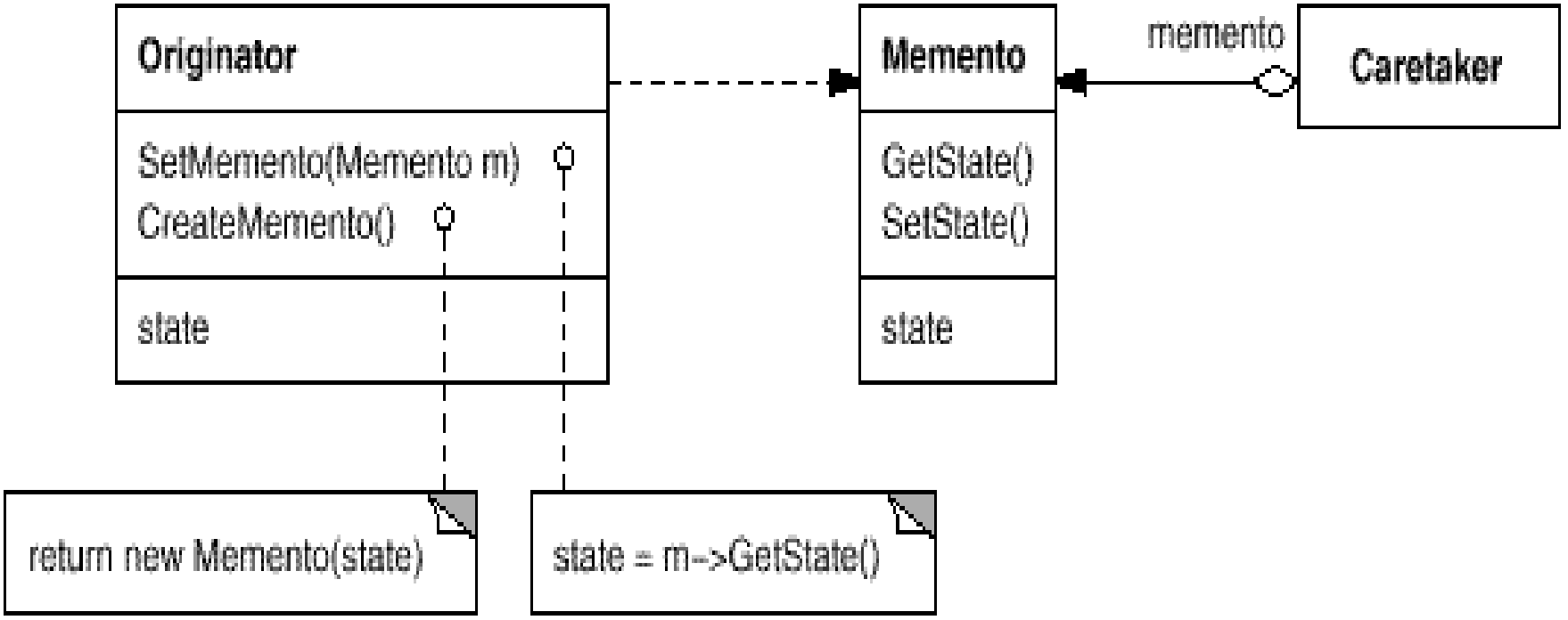
This arrangement lets the ConstraintSolver entrust other objects with the information it needs to revert to a previous state without exposing its internal structure and representations.

Applicability

Use the Memento pattern when

- a snapshot of (some portion of) an object's state must be saved so that it can be restored to that state later, *and*
- a direct interface to obtaining the state would expose implementation details and break the object's encapsulation.

Structure


Originator
SetMemento(Memento m)
CreateMemento()
state
Memento
GetState()
SetState()
state
memento
Caretaker
return new Memento(state)
state = m->GetState()

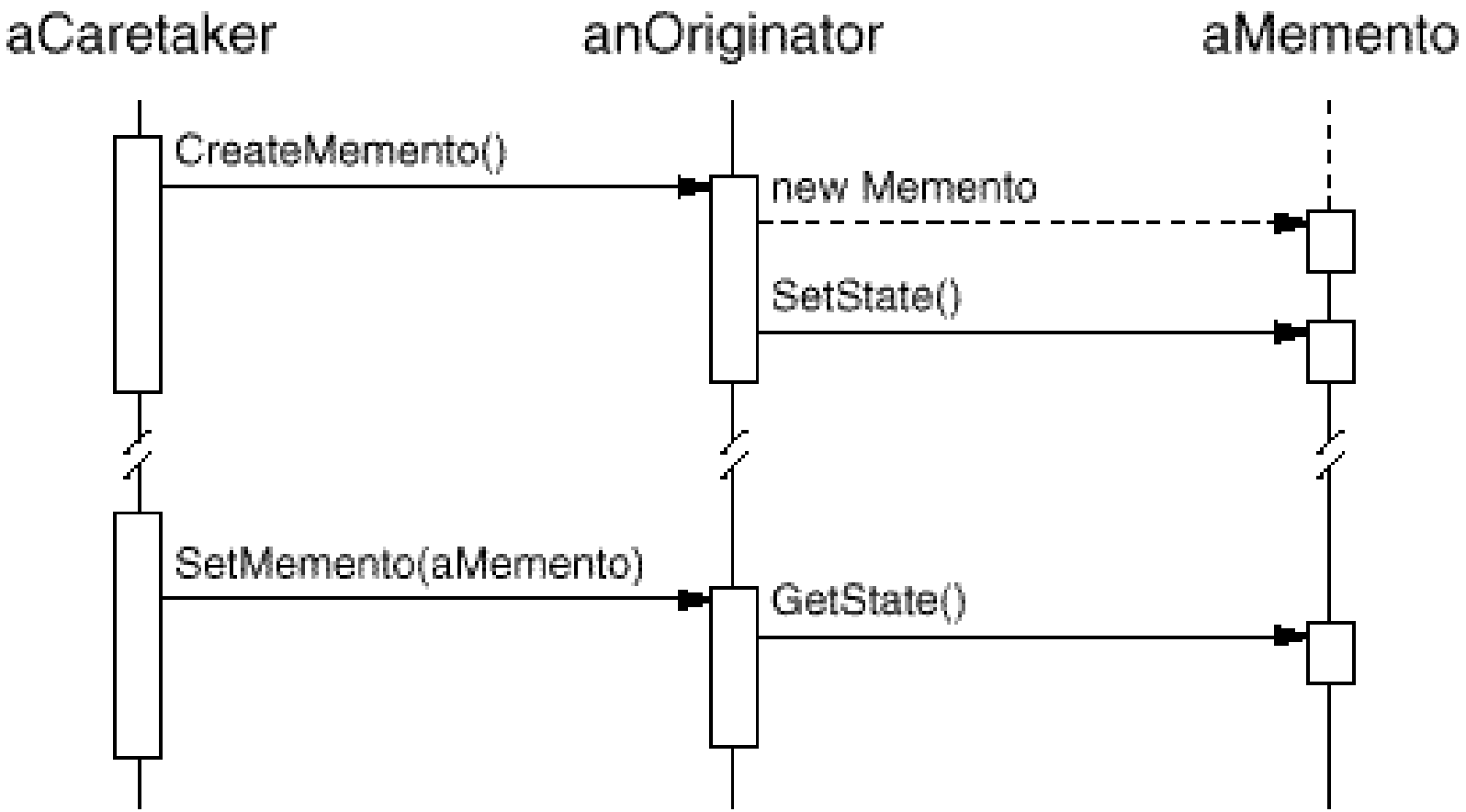
Participants

- **Memento** (SolverState)
  - stores internal state of the Originator object. The memento may store as much or as little of the originator's internal state as necessary at its originator's discretion
  - protects against access by objects other than the originator.
    - Mementos have effectively two interfaces. Caretaker sees a *narrow* interface to the Memento—it can only pass the memento to other objects.
    - Originator, in contrast, sees a *wide* interface, one that lets it access all the data necessary to restore itself to its previous state
    - Ideally, only the originator that produced the memento would be permitted to access the memento's internal state

- **Originator** (ConstraintSolver)
  - creates a memento containing a snapshot of its current internal state.
  - uses the memento to restore its internal state.
- **Caretaker** (undo mechanism)
  - is responsible for the memento's safekeeping.
  - never operates on or examines the contents of a memento

Collaborations

- A caretaker requests a memento from an originator, holds it for a time, and passes it back to the originator, as the following interaction diagram illustrates:



- Sometimes the caretaker won't pass the memento back to the originator, because the originator might never need to revert to an earlier state.

- Mementos are passive
- Only the originator that created a memento willassign or retrieve its state.

Consequences

The Memento pattern has several consequences:

- 1. *Preserving encapsulation boundaries.*
  - Memento avoids exposing information that only an originator should manage but that must be stored nevertheless outside the originator
  - The pattern shields other objects from potentially complex Originator internals, thereby preserving encapsulation boundaries.

- 2. *It simplifies Originator*
  - In other encapsulation-preserving designs, Originator keeps the versions of internal state that clients have requested
  - That puts all the storage management burden on Originator
  - Having clients manage the state they ask for simplifies Originator and keeps clients from having to notify originators when they're done

- 3. *Using mementos might be expensive.*
  - Mementos might incur considerable overhead if Originator must copy large amounts of information to store in the memento or if clients create and return mementos to the originator often enough
  - Unless encapsulating and restoring Originator state is cheap, the pattern might not be appropriate
  - See the discussion of incrementality in the Implementation section.

- 4. *Defining narrow and wide interfaces.*
  - It may be difficult in some languages to ensure that only the originator can access the memento's state.
- 5. *Hidden costs in caring for mementos.*
  - A caretaker is responsible for deleting the mementos it cares for.
  - However, the caretaker has no idea how much state is in the memento.
  - Hence an otherwise lightweight caretaker might incur large storage costs when it stores mementos.

Observer
(Dependents, Publish-Subscribe)

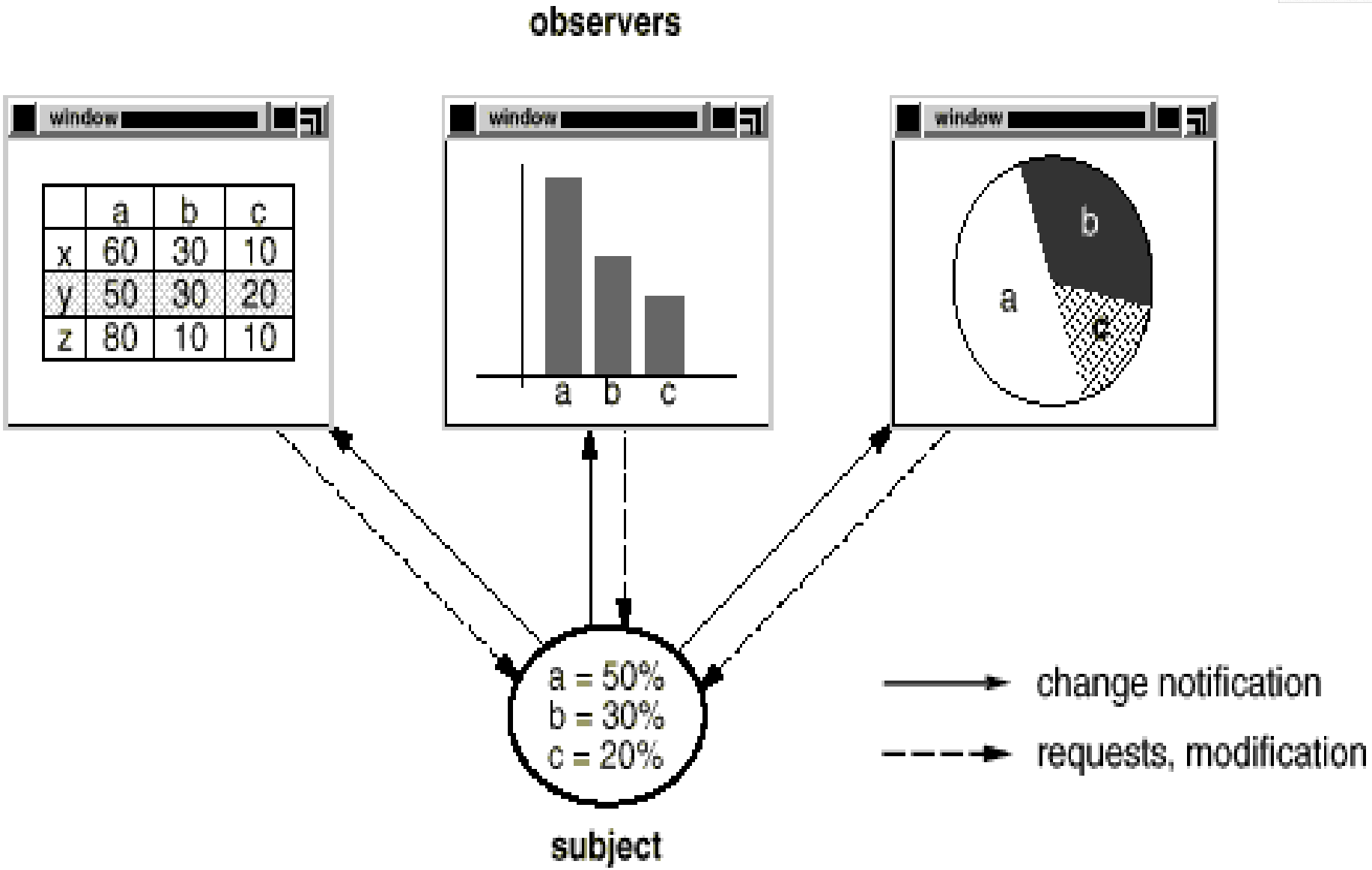
Intent

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

Motivation

- Side-effect of partitioning a system into a collection of cooperating classes is the need to maintain consistency between related objects
  - You don't want to achieve consistency by making the classes tightly coupled, because that reduces their reusability.

- E.g., many GUI toolkits separate the presentational aspects of the user interface from the underlying application data
- Application data & presentations can be Reused separately
- Both a spreadsheet and bar chart objects can depict information in the same application data object using different presentations
- The spreadsheet and the bar chart don't know about each other, thereby letting you reuse only the one you need
- Change in information in the spreadsheet, changes the bar chart immediately, and vice versa.


observers
window
window
window
a b c
x 60 30 10
y 50 30 20
z 80 10 10
a
b
c
a = 50%
b = 30%
c = 20%
subject
change notification
requests, modification

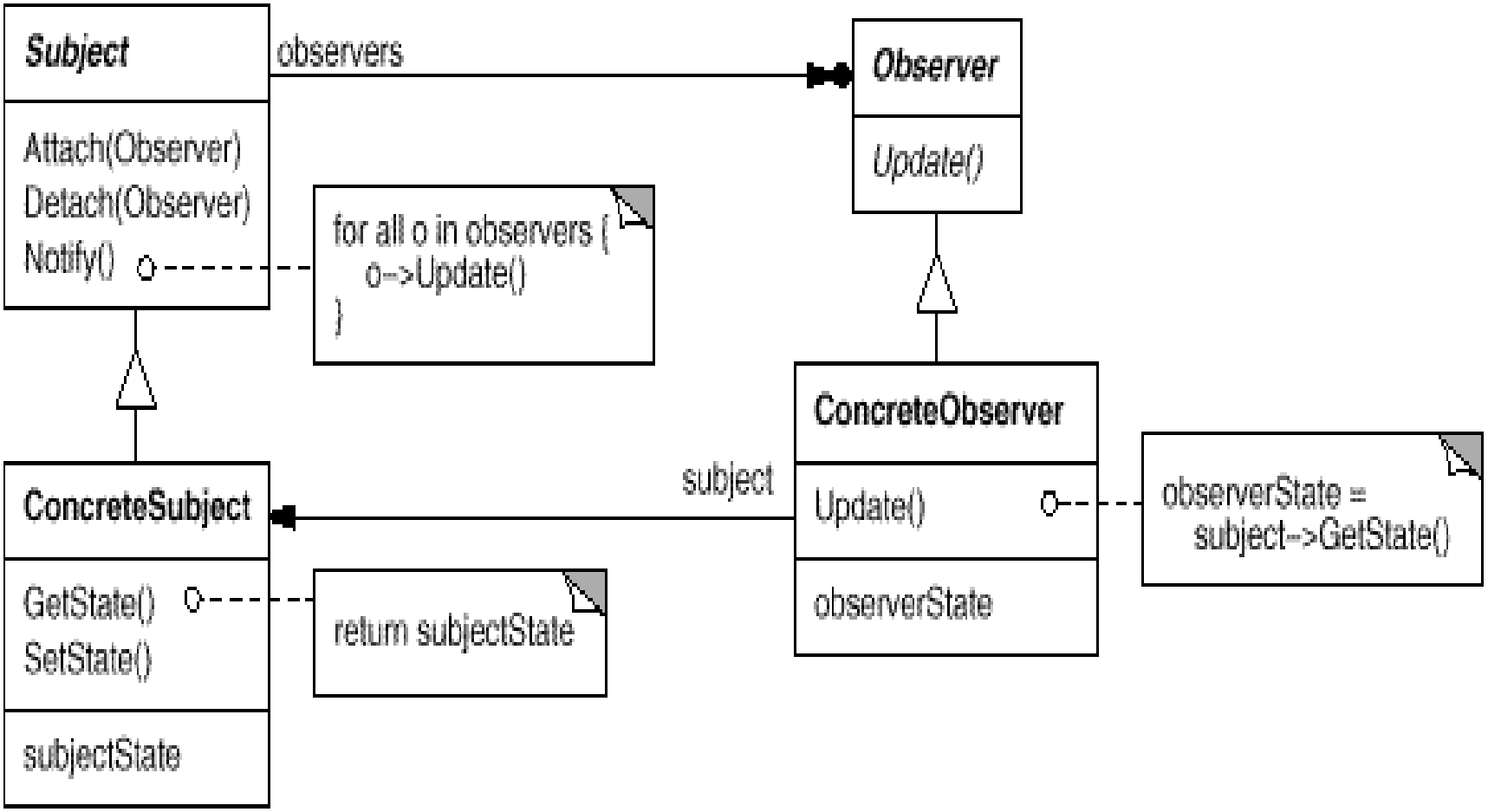
- This behavior implies that the spreadsheet & bar chart are dependent on data object; therefore should be notified of any change in its state
- And there's no reason to limit the number of dependent objects
- Observer pattern describes how to establish these relationships
- Key objects in this pattern are **subject** and **observer**
- A subject may have any number of dependent observers
- All observers are notified when subject undergoes change in state
- Each observer will query the subject to synchronize its state with subject's state

- This kind of interaction is also known as **publish-subscribe**
- The subject is the publisher of notifications
- It sends out these notifications without having to know who its observers are
- Any number of observers can subscribe to receive notifications.

Applicability

Use the Observer pattern in any of the following situations:

- When an abstraction has two aspects, one dependent on the other
  - Encapsulating these aspects in separate objects lets you vary and reuse them independently.
- 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 be able to notify other objects without making assumptions about who these objects are
- In other words, you don't want these objects tightly coupled.

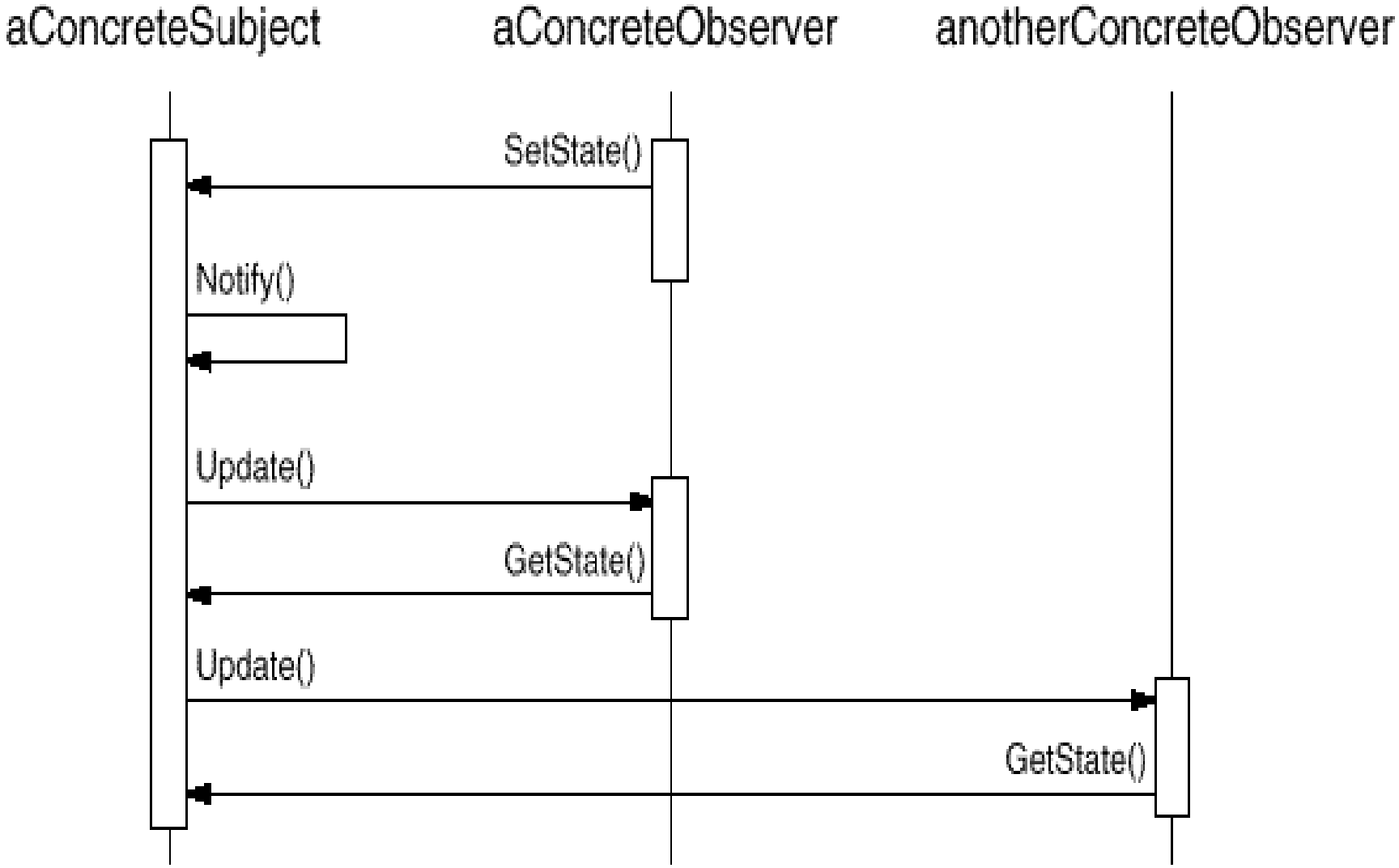

Subject
observers
Observer
Attach(Observer)
Detach(Observer)
Notify()
Update()
for all o in observers {
o->Update()
}
ConcreteObserver
subject
ConcreteSubject
Update()
observerState =
subject->GetState()
GetState()
SetState()
observerState
return subjectState
subjectState

Participants

- **Subject**
  - knows its observers. Any number of Observer objects may observe a subject.
  - provides an interface for attaching and detaching Observer objects.
- **Observer**
  - defines an updating interface for objects that should be notified of changes in a subject.
- **ConcreteSubject**
  - stores state of interest to ConcreteObserver objects.
  - sends a notification to its observers when its state changes.
- **ConcreteObserver**
  - maintains a reference to a ConcreteSubject object.
  - stores state that should stay consistent with the subject's.
  - implements the Observer updating interface to keep its state consistent with the subject's

Collaborations

- ConcreteSubject notifies its observers whenever a change occurs that could make its observers' state inconsistent with its own.
- After being informed of a change in the concrete subject, a ConcreteObserver object may query the subject for information
- ConcreteObserver uses this information to reconcile its state with that of the subject
- The following interaction diagram illustrates the collaborations between a subject and two observers:


aConcreteSubject
aConcreteObserver
anotherConcreteObserver
SetState()
Notify()
Update()
GetState()
Update()
GetState()

- Note how the Observer object that initiates the change request postpones its update until it gets a notification from the subject
- Notify is not always called by the subject
- It can be called by an observer or by another kind of object entirely
- The Implementation section discusses some common variations.

Consequences

- The Observer pattern lets you vary subjects and observers independently
- You can reuse subjects without reusing their observers, and vice versa
- It lets you add observers without modifying the subject or other observers
- Further benefits and liabilities of the Observer pattern include the following:

- 1. *Abstract coupling between Subject and Observer*
  - All a subject knows is that it has a list of observers, each conforming to the simple interface of the abstract Observer class
  - The subject doesn't know the concrete class of any observer. Thus the coupling between subjects and observers is abstract and minimal
  - Because Subject and Observer aren't tightly coupled, they can belong to different layers of abstraction in a system
  - A lower-level subject can communicate and inform a higher-level observer, thereby keeping the system's layering intact
  - If Subject and Observer are lumped together, then the resulting object must either span two layers (and violate the layering), or it must be forced to live in one layer or the other (which might compromise the layering abstraction).

- 2. *Support for broadcast communication*
  - Unlike an ordinary request, the notification that a subject sends needn't specify its receiver
  - The notification is broadcast automatically to all interested objects that subscribed to it
  - The subject doesn't care how many interested objects exist; its only responsibility is to notify its observers
  - This gives you the freedom to add and remove observers at any time
  - It's up to the observer to handle or ignore a notification.

- 3. *Unexpected updates*
  - Because observers have no knowledge of each other's presence, they can be blind to the ultimate cost of changing the subject
  - A seemingly innocuous operation on the subject may cause a cascade of updates to observers and their dependent objects
  - Moreover, dependency criteria that aren't well-defined or maintained usually lead to spurious updates, which can be hard to track down
  - This problem is aggravated by the fact that the simple update protocol provides no details on *what* changed in the subject
  - Without additional protocol to help observers discover what changed, they maybe forced to work hard to deduce the changes.

Implementation

Several issues related to implementation of dependency mechanism:

- 1. *Mapping subjects to their observers*
  - Simplest way for a subject to keep track of observers it should notify is to store references to them explicitly in the subject
  - Such storage is too expensive for many subjects & few observers
  - One solution is to trade space for time by using an associative look-up (e.g., a hash table) to maintain the subject-to-observer mapping
  - A subject with no observers does not incur storage overhead
  - This approach increases the cost of accessing the observers.

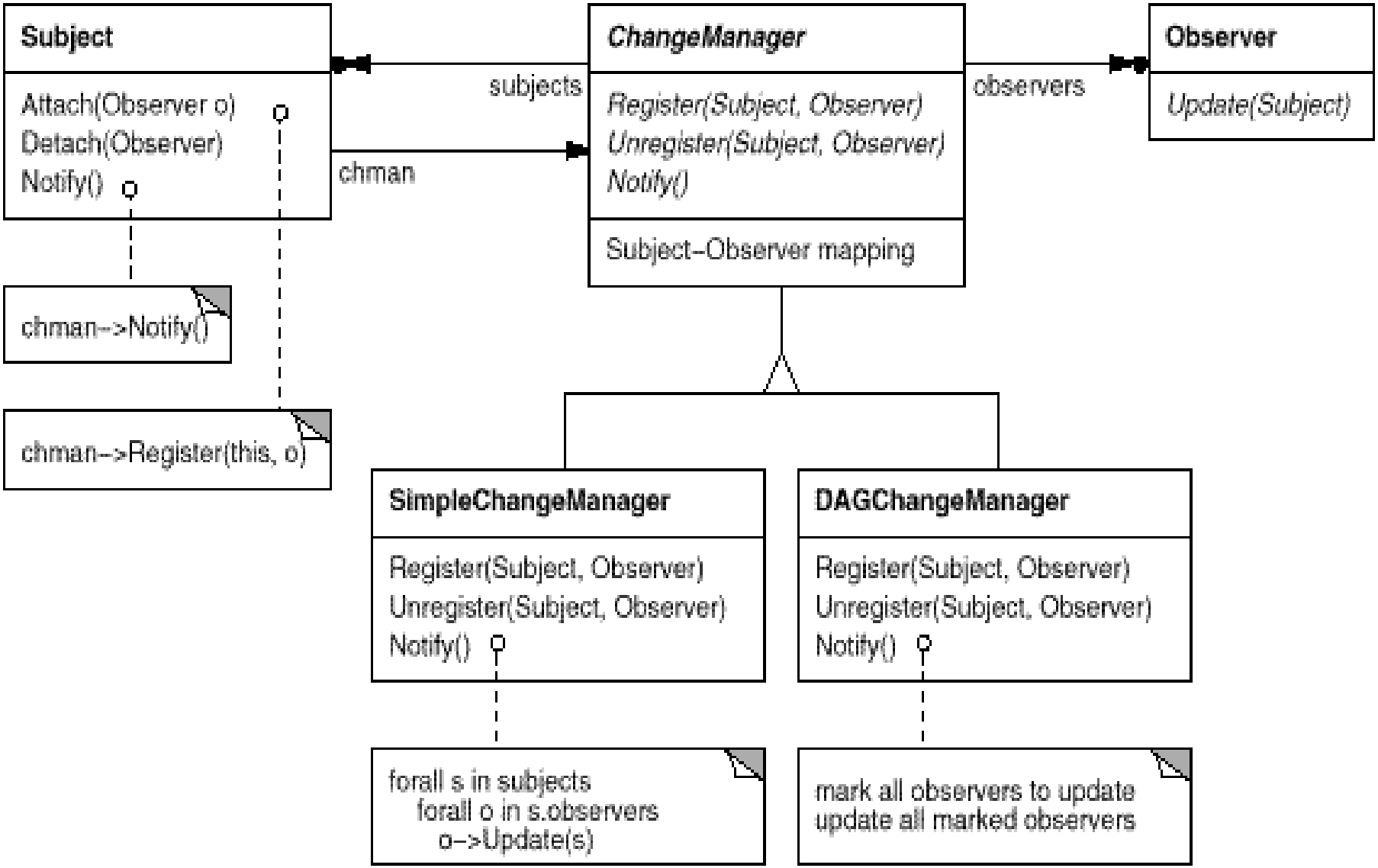
- 2. *Observing more than one subject*
  - It might make sense in some situations for an observer to depend on more than one subject
  - E.g., a spreadsheet depend on more than one data source
  - It's necessary to extend the Update interface in such cases to let the observer know *which* subject is sending the notification
  - The subject can simply pass itself as a parameter in the Update operation, thereby letting the observer know which subject to examine.

- 3. *Who triggers the update?*
  - Subject & its observers rely on notification mechanism to stay consistent
  - But what object actually calls Notify to trigger the update?
  - Here are two options:
  - A. Have state-setting operations on Subject call Notify after they change the subject's state
  - The advantage of this approach is that clients don't have to remember to call Notify on the subject
  - The disadvantage is that several consecutive operations will cause several consecutive updates, which may be inefficient.

- B. Make clients responsible for calling Notify at right time
- The advantage here is that the client can wait to trigger the update until after a series of state changes has been made, thereby avoiding needless intermediate updates
- Disadvantage is that clients have an added responsibility to trigger the update
- That makes errors more likely, since clients might forget to call Notify

- 4. *Dangling references to deleted subjects*
- Deleting a subject should not produce dangling references in its observers
- One way to avoid dangling references is to make the subject notify its observers as it is deleted so that they can reset their reference to it
- Simply deleting the observers is not an option, because other objects may reference them, or they may be observing other subjects as well.

- 5. *Making sure Subject state is self-consistent before notification*
- It's important to make sure Subject state is self-consistent before calling
- Notify, because observers query the subject for its current state in the course of updating their own state.


Subject
Attach(Observer o)
Detach(Observer)
Notify()
subjects
chman
ChangeManager
Register(Subject, Observer)
Unregister(Subject, Observer)
Notify()
Subject–Observer mapping
observers
Observer
Update(Subject)
chman->Notify()
chman->Register(this, o)
SimpleChangeManager
Register(Subject, Observer)
Unregister(Subject, Observer)
Notify()
DAGChangeManager
Register(Subject, Observer)
Unregister(Subject, Observer)
Notify()
forall s in subjects
forall o in s.observers
o->Update(s)
mark all observers to update
update all marked observers