Behavioral Patterns

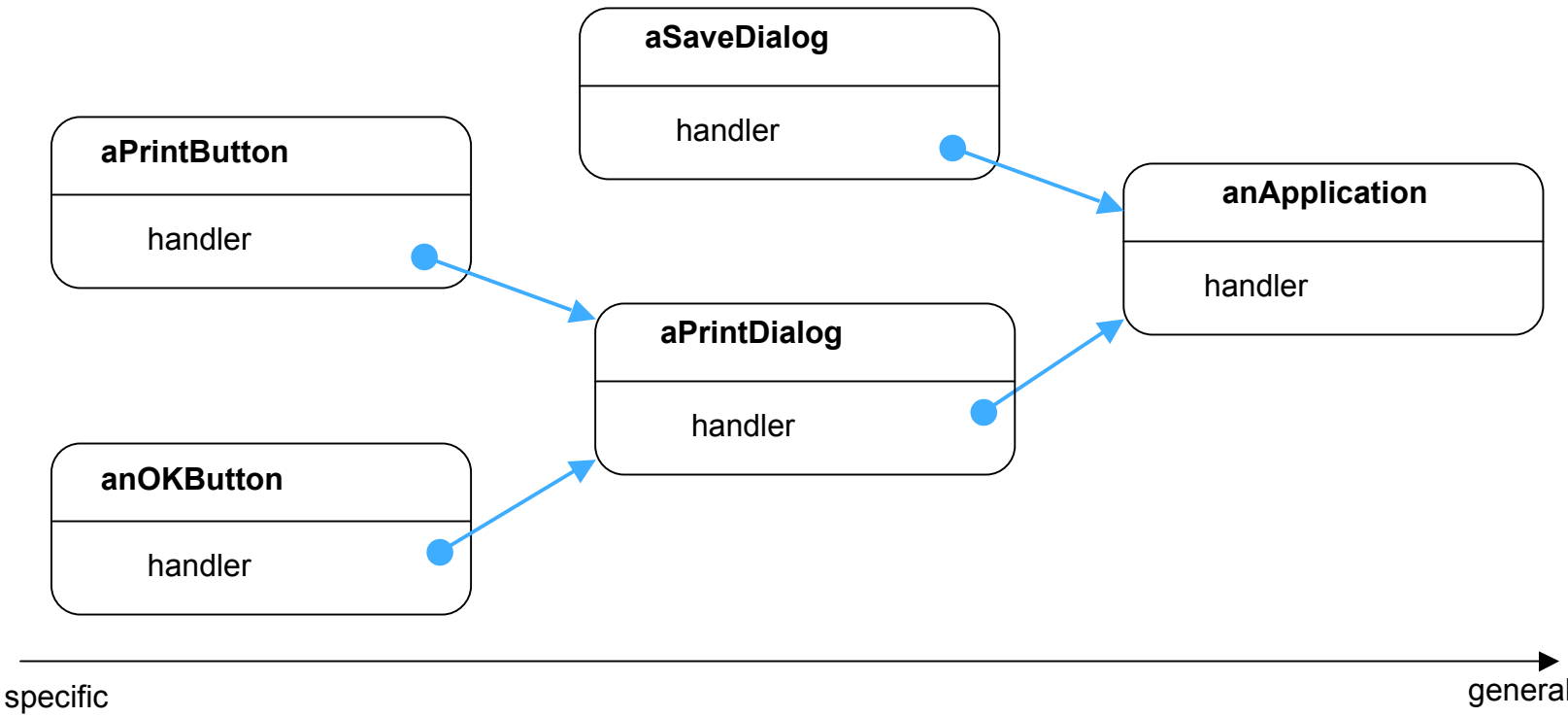
- Concerned with algorithms and the assignment of responsibilities between objects.
- Describe communication flows among objects.

- Behavioral **class** patterns use inheritance to distributed behavior between classes.
- Behavioral **object** patterns use object composition rather than inheritance - they describe how groups of peer objects cooperate.

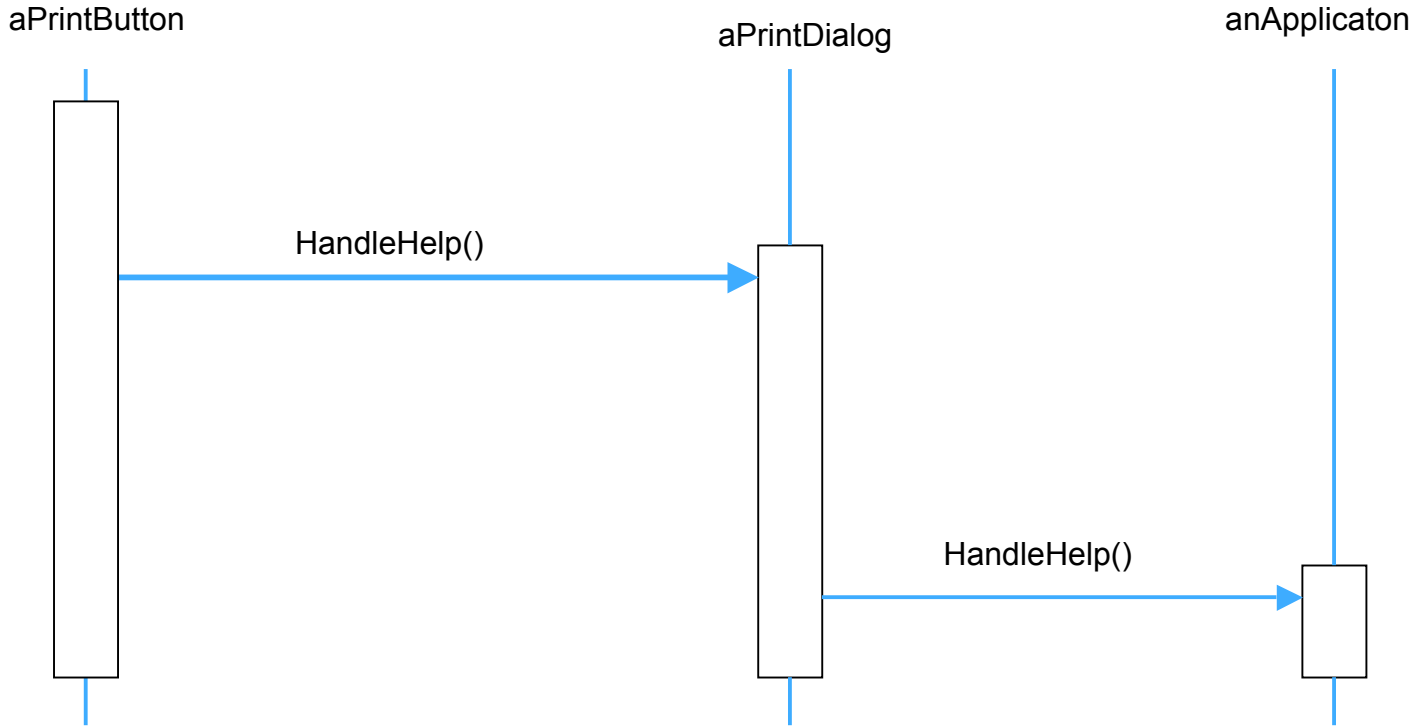
CHAIN OF RESPONSIBILITY (Object Behavioral)

- 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 and pass the request along the chain until an Object handles it.
- Motivation:
  - Consider a context-sensitive help facility for a GUI. Users can obtain help info on any part of the UI by just clicking on it.
  - Help provided depends on the part of the UI selected and its context.
  - Object that provides help is not directly known to object (e.g. button) that initiates the request.
  - Chain of responsibility allows to decouple senders and receivers of requests.

CHAIN OF RESPONSIBILITY
Motivation

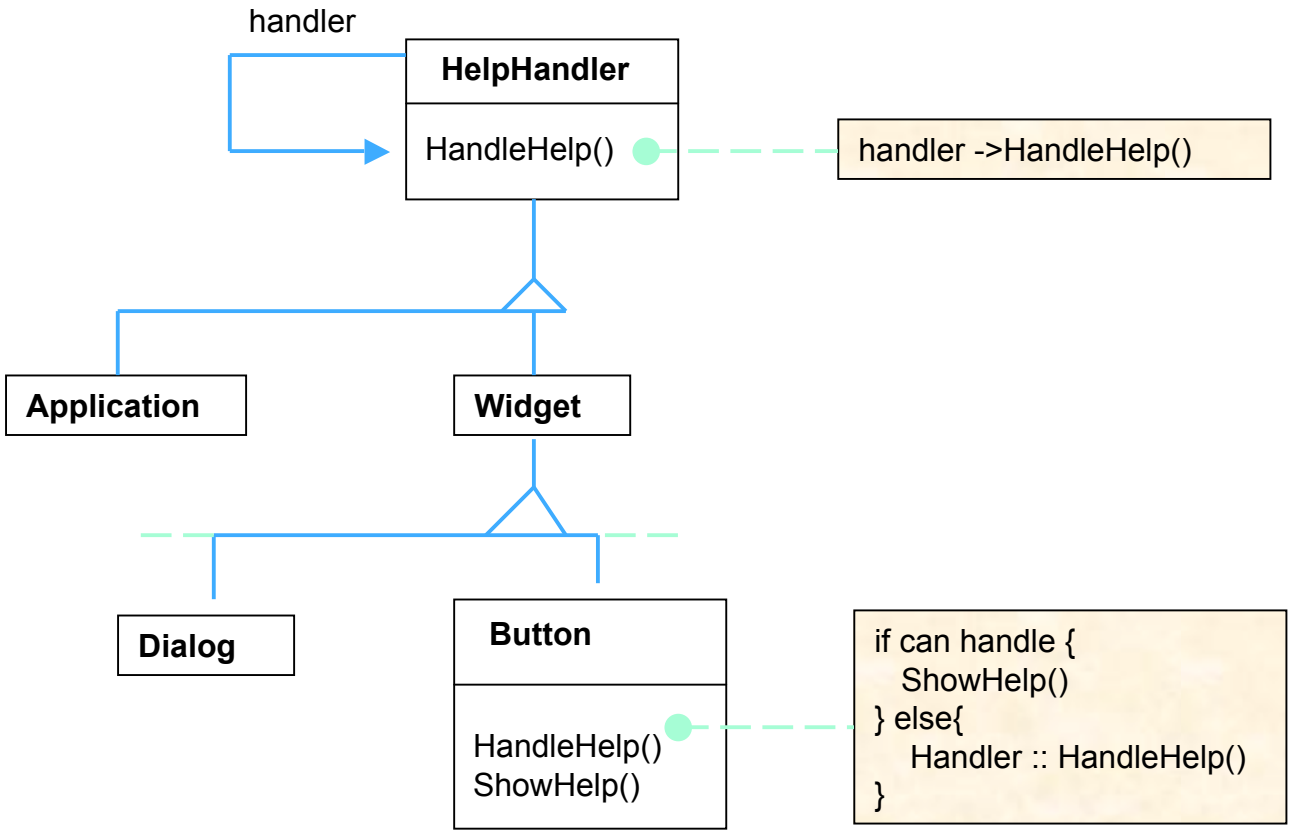


CHAIN OF RESPONDINILITY
Motivation



- An object in the chain receives the request and either handles it or forwards it to the next candidate on the chain.
- The request has an implicit receiver.

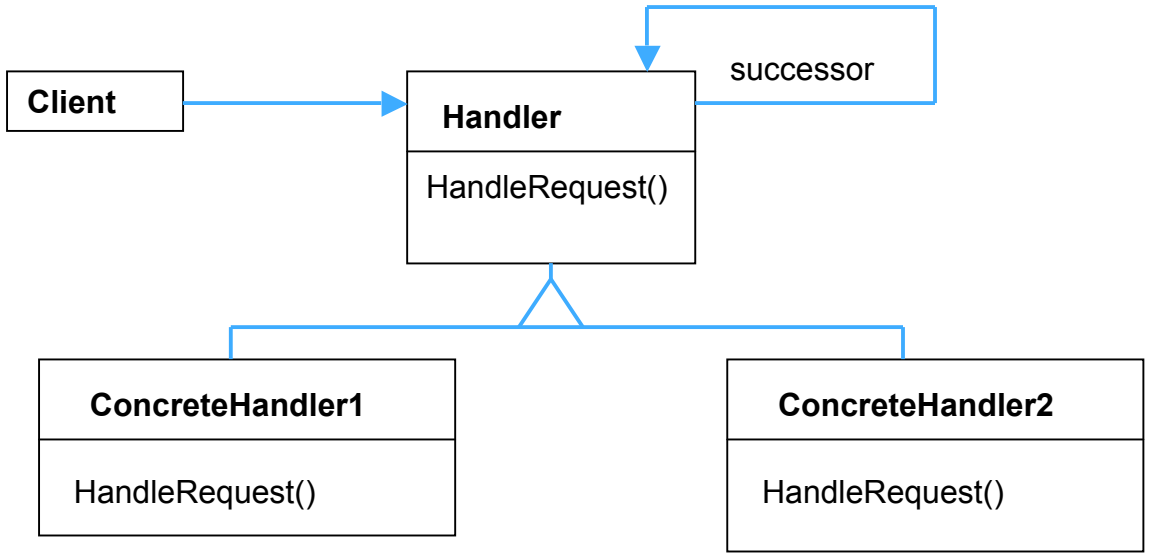
CHAIN OF RESPONSIBILITY
Motivation



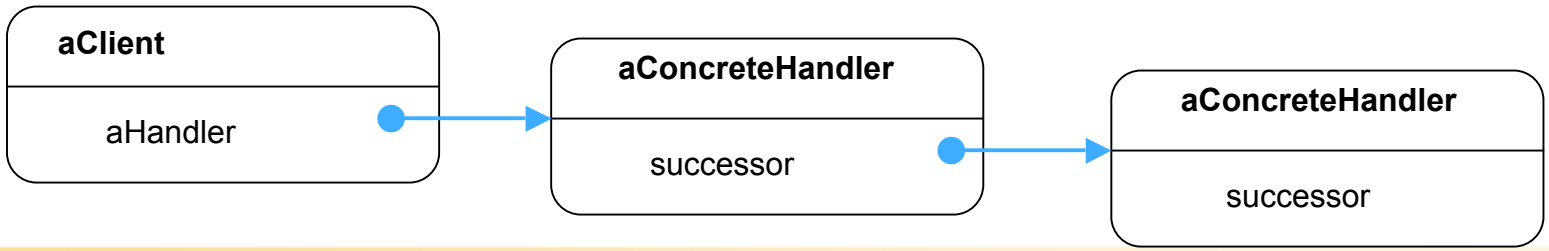
Applicability

- Use Chain of Responsibility when:
  - More than one object may handle a request, and the handler is not known a priori. The handler should be ascertained 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.

CHAIN OF RESPONSIBILITY
Structure



A typical object structure might look like this:



Participants and Collaborations

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.
  - Either handles requests or forwards it to its successor.
- Client
  - Initiates the request to a ConcreteHandler object on the chain.

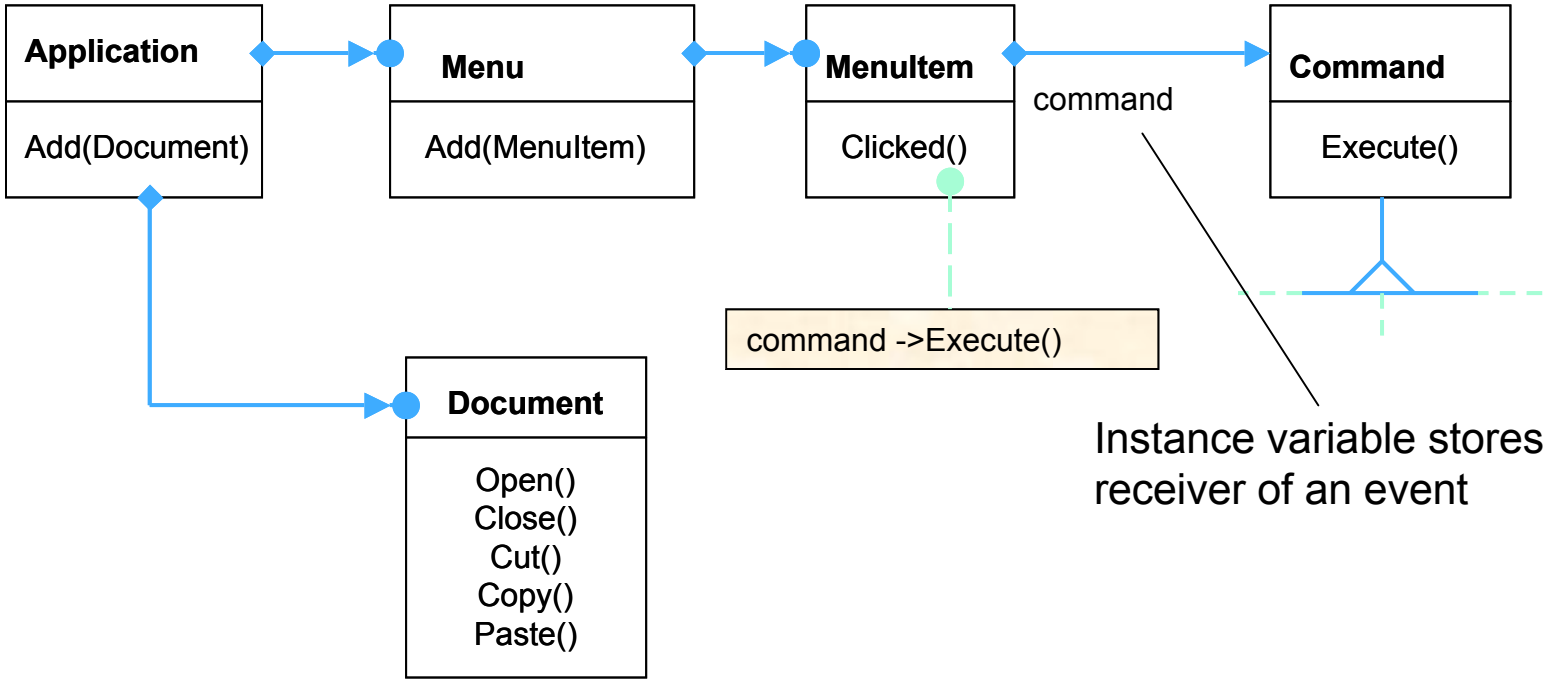
Collaborations:

- When a client issues a request, the request propagates along the chain until a ConcreteHandler object takes responsibility for it.

COMMAND
(Object Behavioral)

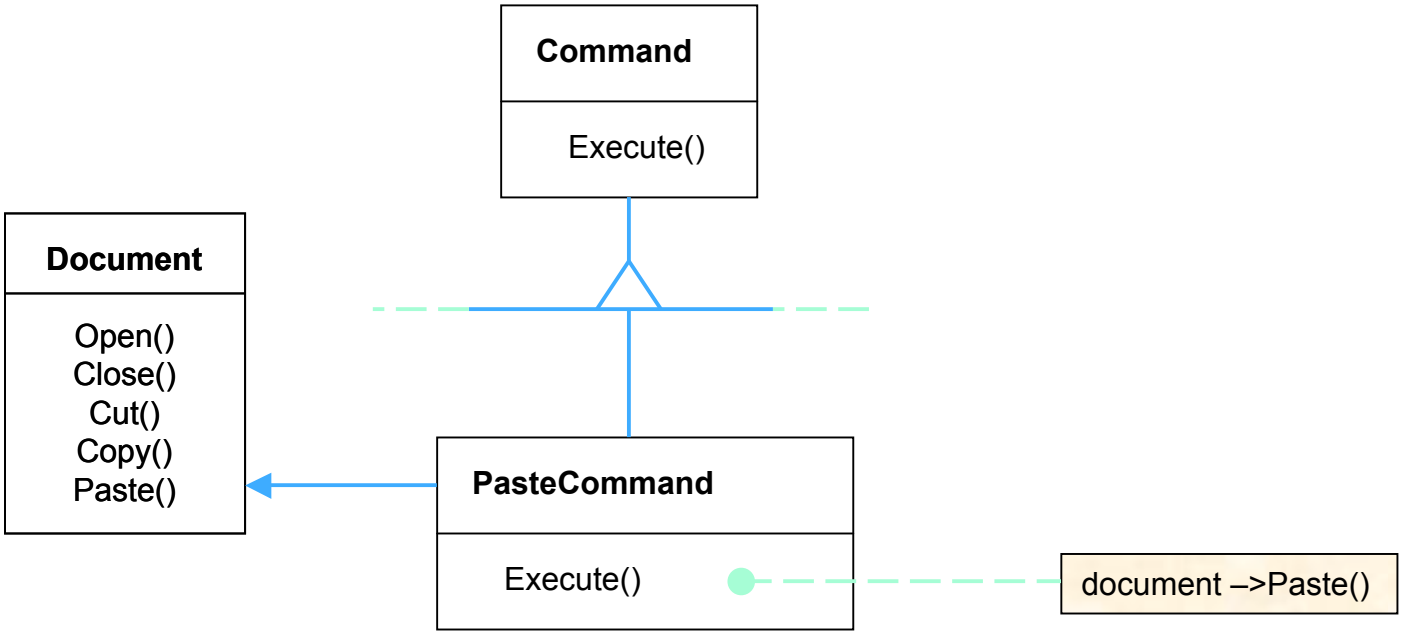
- Intent:
  - Encapsulate a request as an object,
  - Parameterize clients with different requests, queue or log requests,
  - Support undoable operations (Transactions).
- Motivation:
  - Let toolkit objects make requests of unspecified application objects by turning the request itself into an object.
  - This object can be stored and passed around like other objects.
  - Key to this pattern is an abstract Command class which declares an interface for executing operations.

COMMAND
Motivation



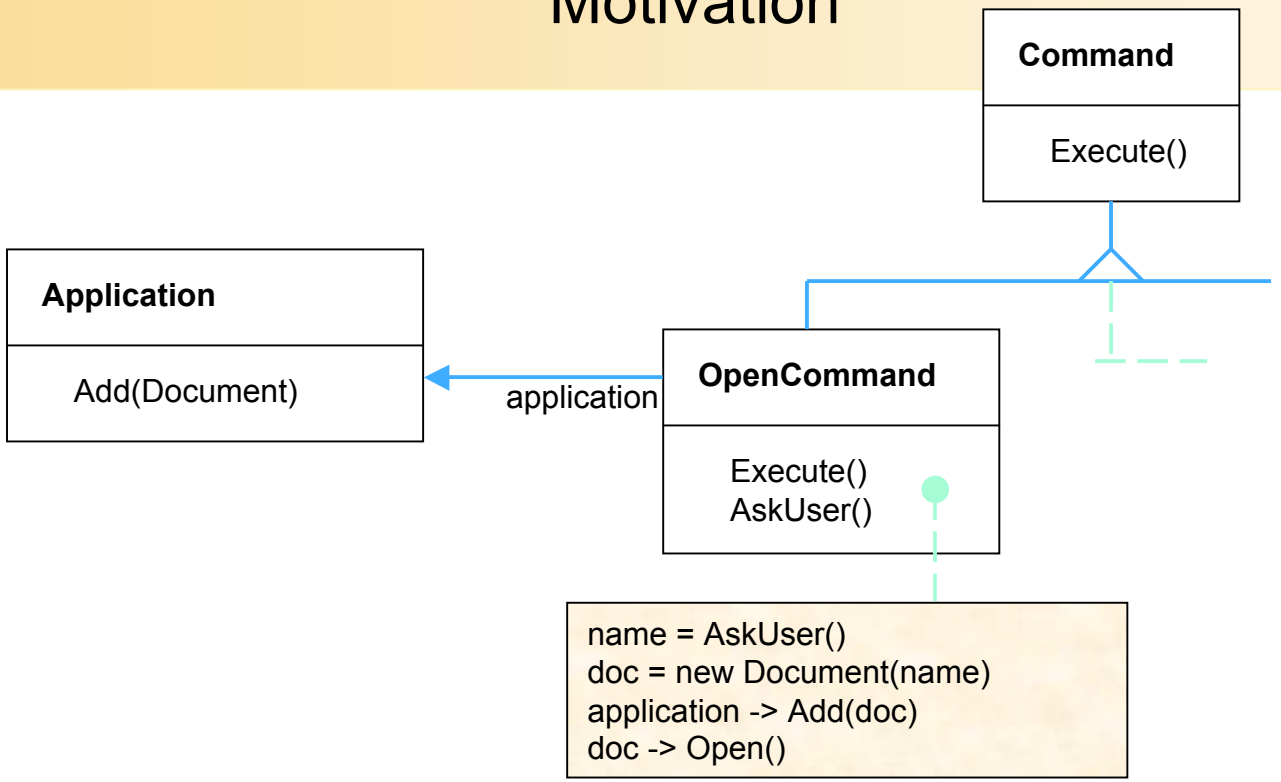
- Menus can be implemented with Command objects. Each choice in a Menu is an instance of a MenuItem class.
- An App creates menus and their menu items along with the rest of the UI.

COMMAND
Motivation



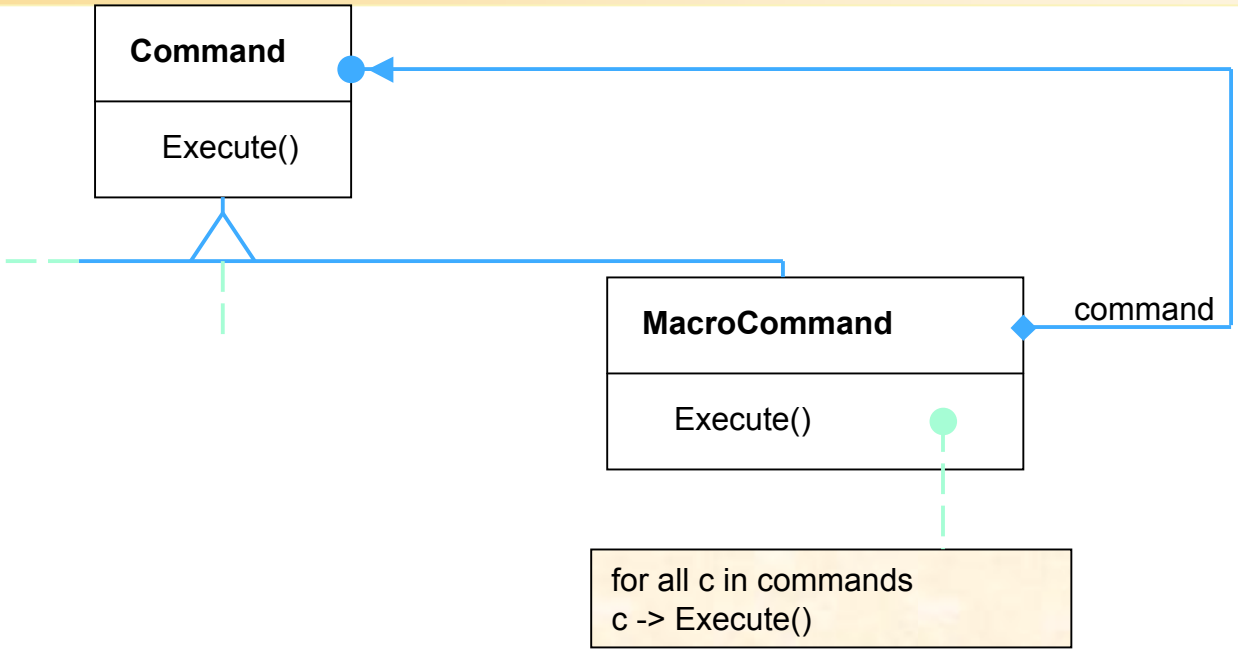
- PasteCommand supports pasting text from the clipboard into a document.
- PasteCommand‘s receiver is the Document object given at instantiation.
- The Execute operation invokes Paste on the receiving document.

COMMAND
Motivation



OpenDocument‘s Execute operation prompts the user for a name,
creates the corresponding document object,
adds document to the receiving app, and opens the document

COMMAND
Motivation



- MacroCommand is a concrete Command subclass
- It simply executes a sequence of commands.
- MacroCommand has no explicit receiver - commands in the sequence define their own receivers.

Applicability

Use the Command pattern when you want to:

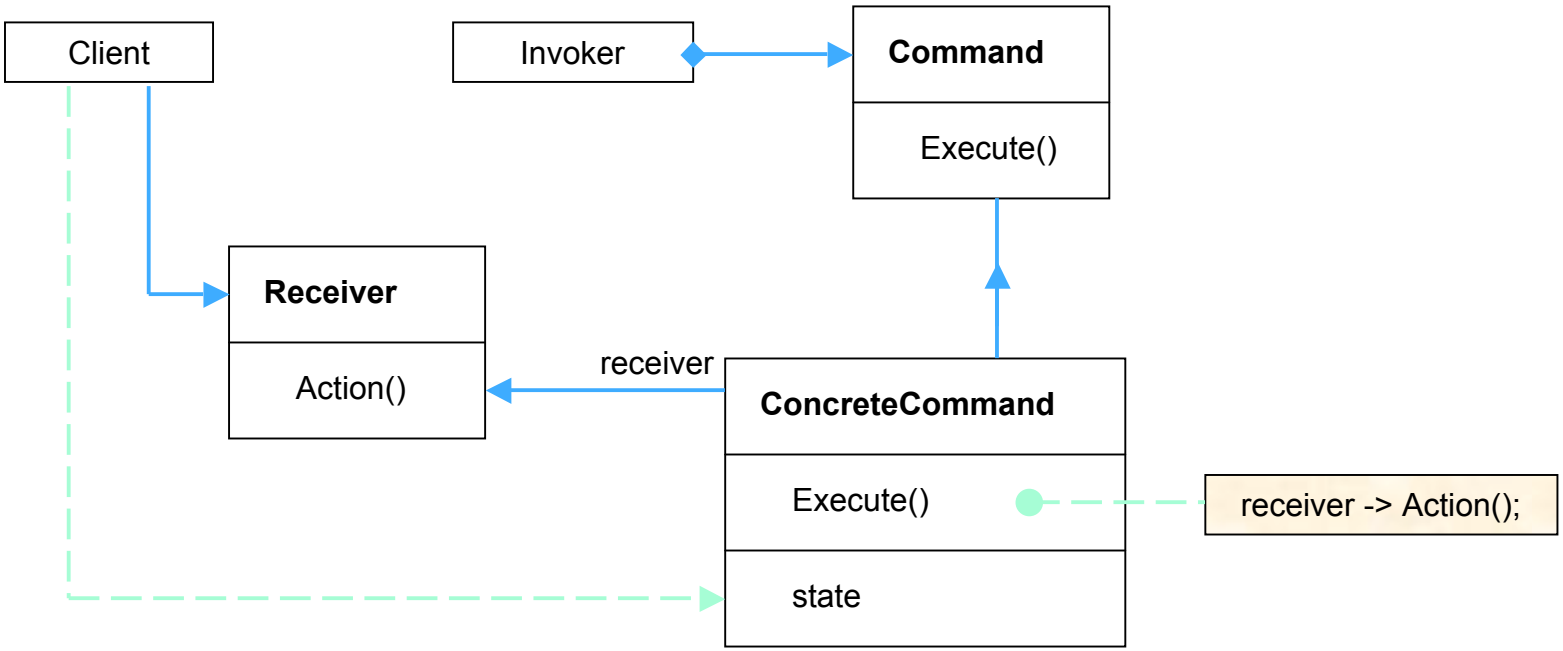
- Parameterize objects by an action to perform (as MenuItem did)
  - Commands are an object-oriented replacement for callbacks.
- Specify, queue, and execute requests at different times.
  - A Command object can have a lifetime independent of the original request.
  - If the receiver of a request can be represented in an address space-independent way, then you can transfer a command object for the request to a different process and fulfill the request there.
- Support undo.
  - The Command´s Execute operation can store state for reversing its effects in the command itself.
  - The Command interface must have an added Un-Execute operation that reverses the effects of a previous call to Execute.
  - Executed commands are stored in a history list.

Applicability II

Use the Command pattern when you want to:

- 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.
  - Recovering from a crash involves reloading logged commands from disk and reexecuting them with the Execute operation.
- Structure a system around high-level operations built on primitives operations.
  - Such a structure is common in information systems that support transactions.
  - A transaction encapsulates a set of changes to data.
  - The Command pattern offers a way to model transactions.
  - Commands have a common interface, letting you invoke all transactions the same way.
  - The pattern also makes it easy to extend the system with new transactions.

COMMAND
Structure



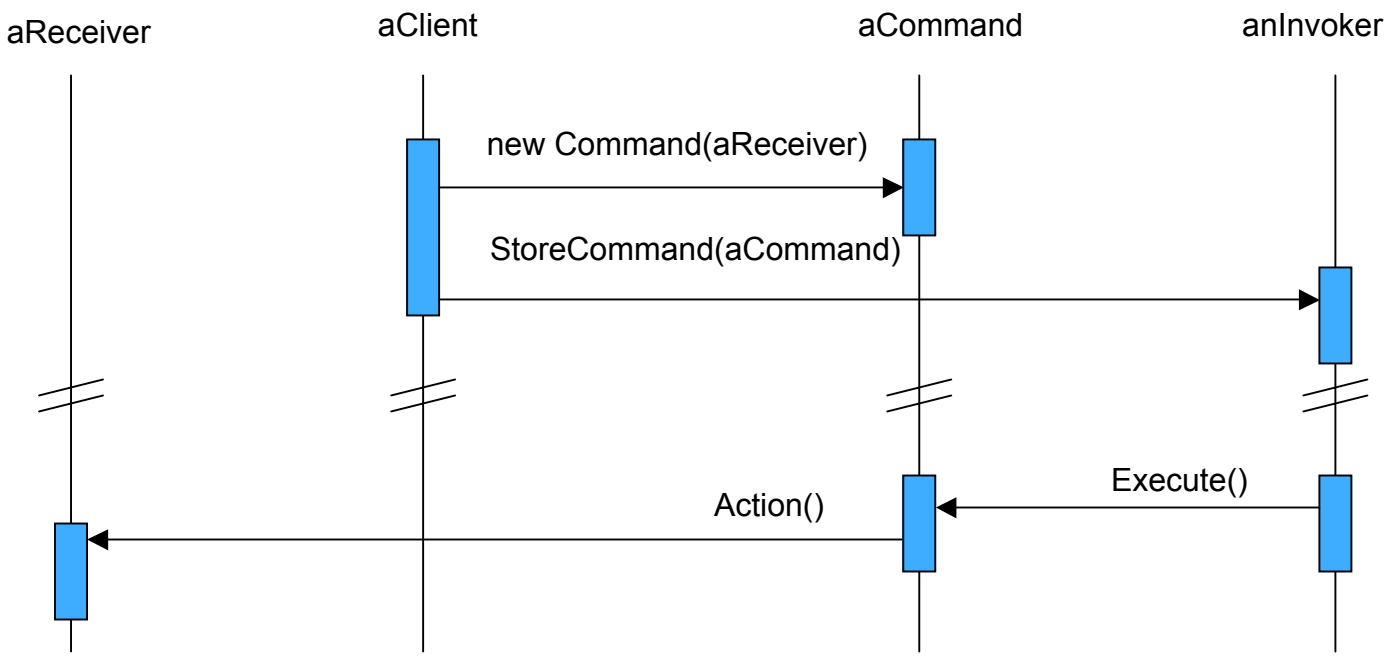
Participants

- 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 ops on Receiver
- Client (Application)
  - Creates a ConcreteCommand object and sets ist 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 server 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. When commands are undoable, ConcreteCommand stores state for undoing the command prior to invoking Execute.
- The ConcreteCommand object invokes operations on its receiver to carry out the request.

Interactions between objects

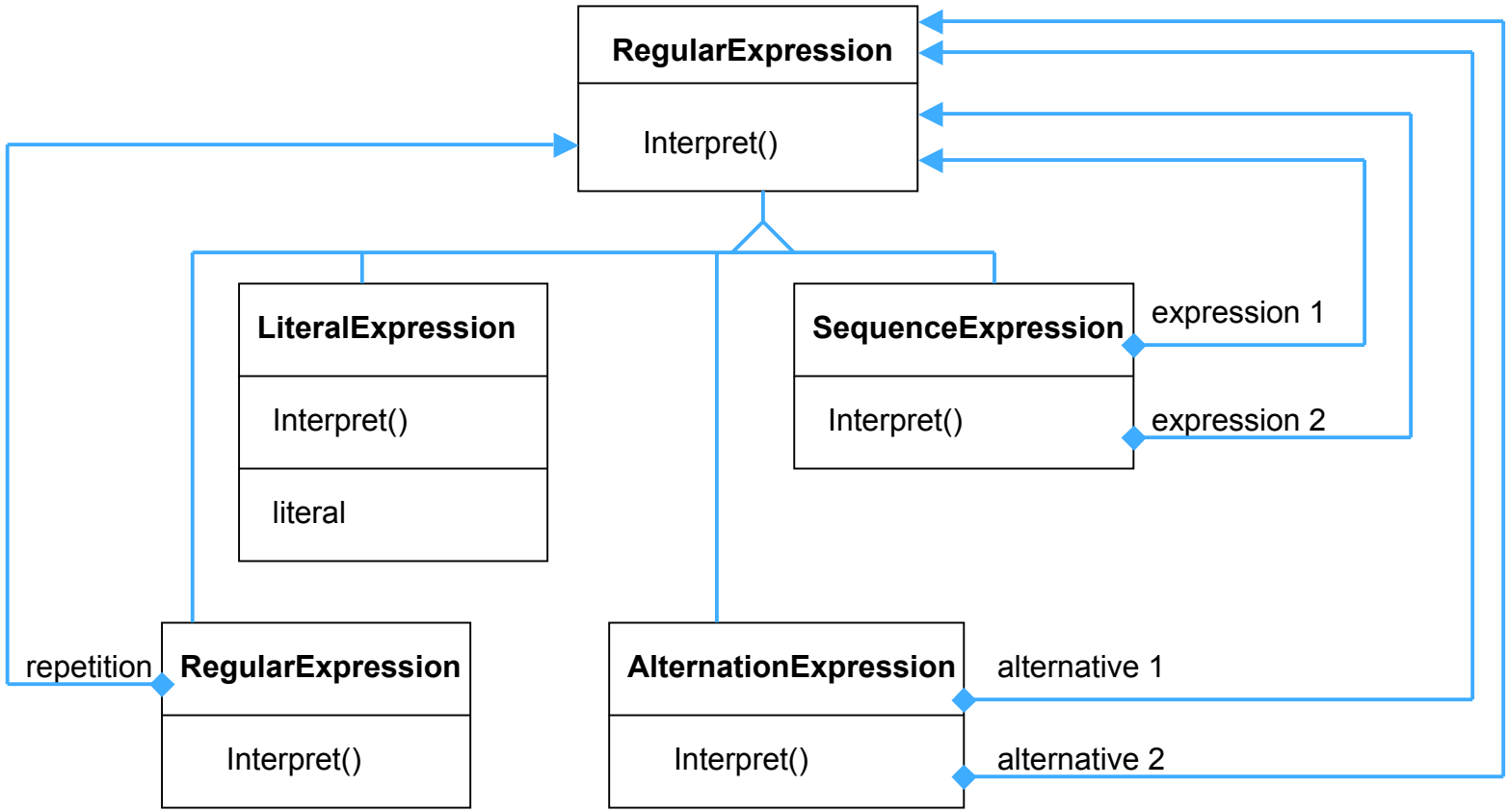


INTERPRETER
(Class Behavioral)

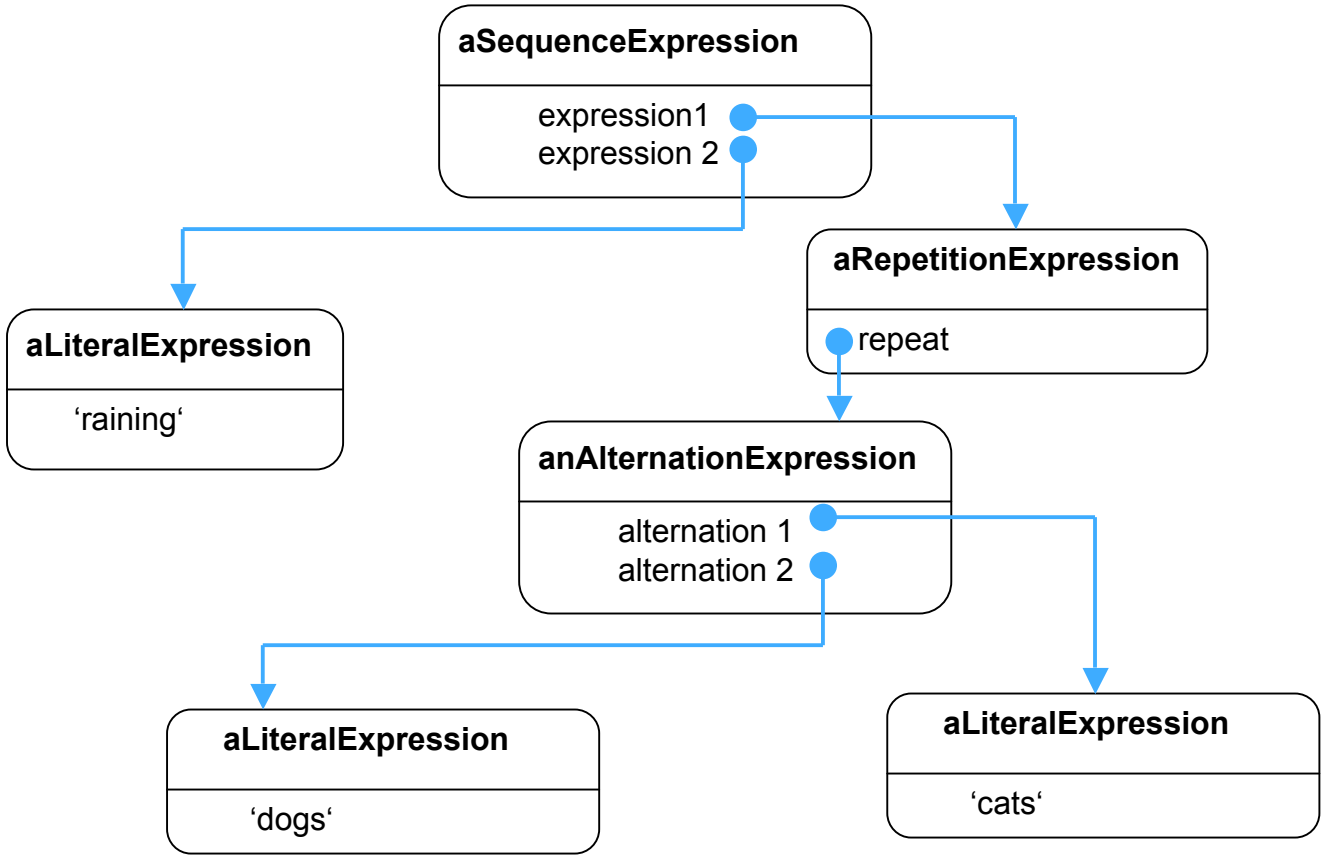
- Intent:

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

INTERPRETER
Motivation



INTERPRETER
Motivation

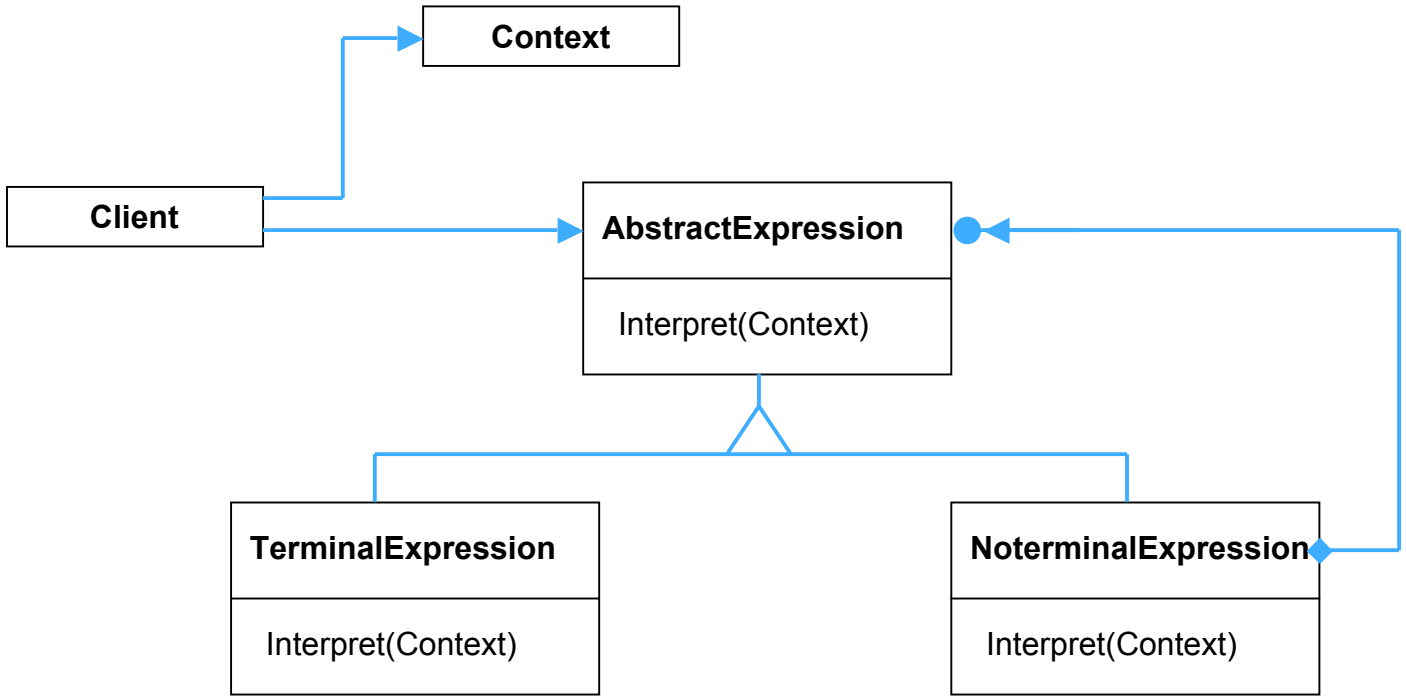


Applicability

Use the Interpreter pattern when there is a language to interpret, and you can represent statments in the language as abstract syntax trees. The Interpreter pattern works best when:

- The grammar is simple. For complex grammars, the class hierarchy for the grammar becomes large and unmanageable. Tools such as 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. The most efficient interpreters are usually not implemented by interpreting parse trees directly but by first translating them into another form. For example, regular expressions are often transformed into state machines. But even then, the translator can be implemented by the Interpreter pattern, so the pattern is still applicable.

INTERPRETER
Structure



Collaborations

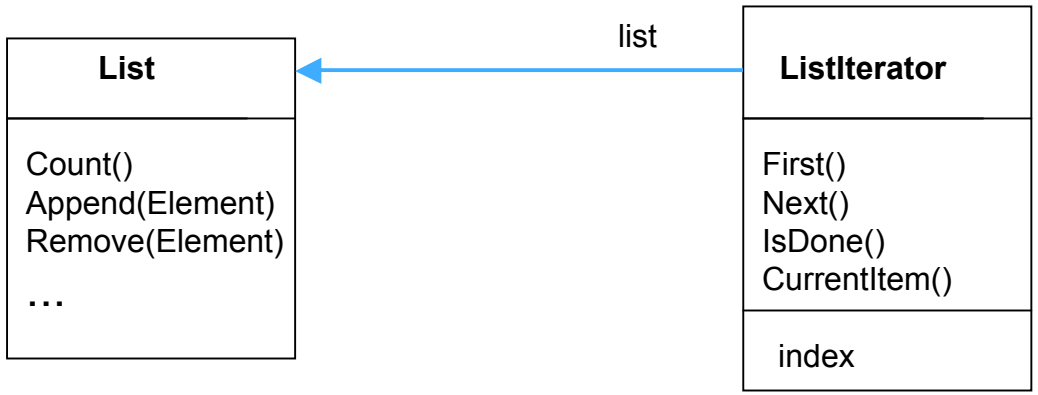
- The client builds (or is given) the sentence as an abstract syntax tree of NonterminalExpression and TerminalExpression instances. Then the client initializes the context and invokes the Interpret operation.
- Each NonterminalExpression node defines Interpret in terms of Interpret on each subexpression. The Interpret operation of each TerminalExpression 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.

ITERATOR
(Object Behavioral)

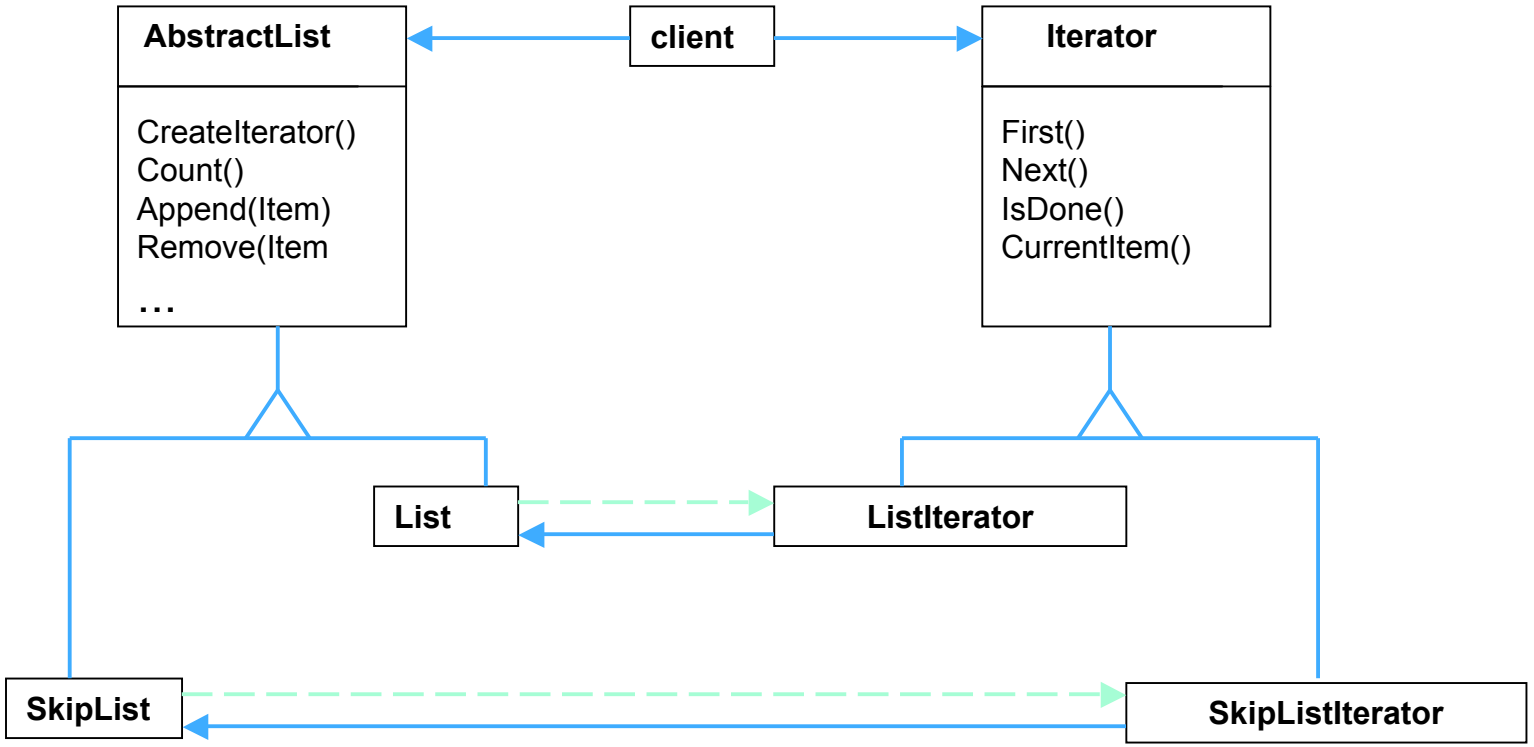
- Intent:

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

ITERATOR
Motivation



ITERATOR
Motivation

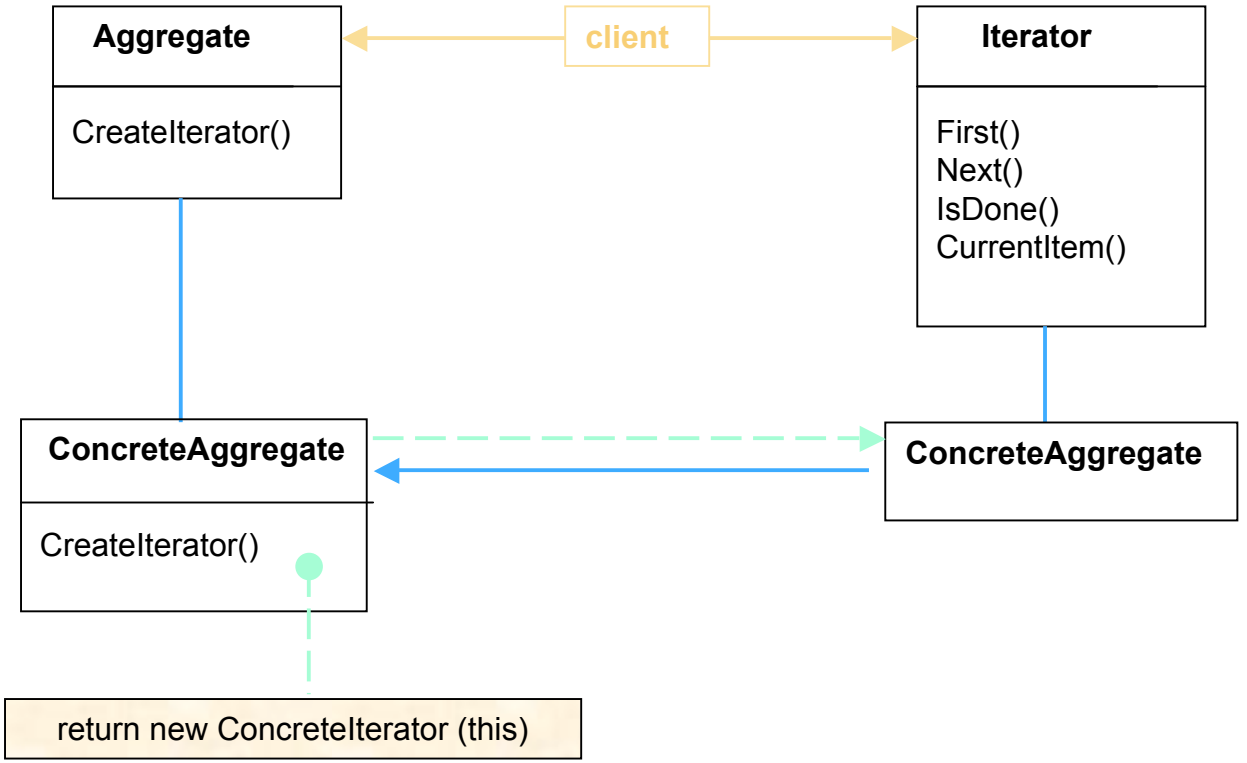


Applicability

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 (that is, to support polymorphic iteration).

ITERATOR
Structure



Collaborations

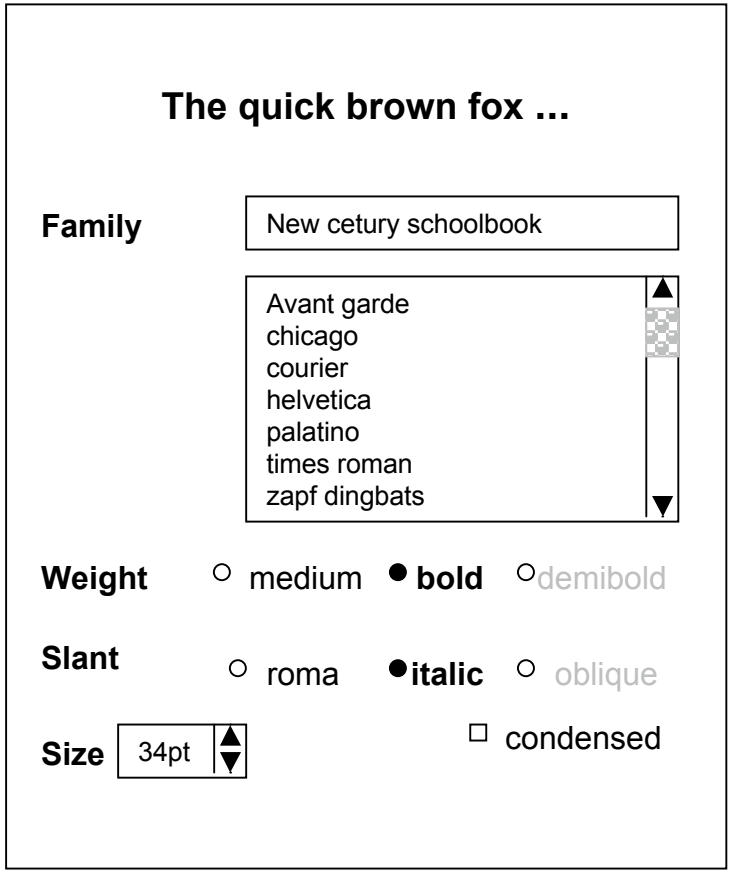
- A ConcreteIterator keeps track of the current object in the aggregate and can compute the succeeding object in the traversal.

MEDIATOR
(Object Behavioral)

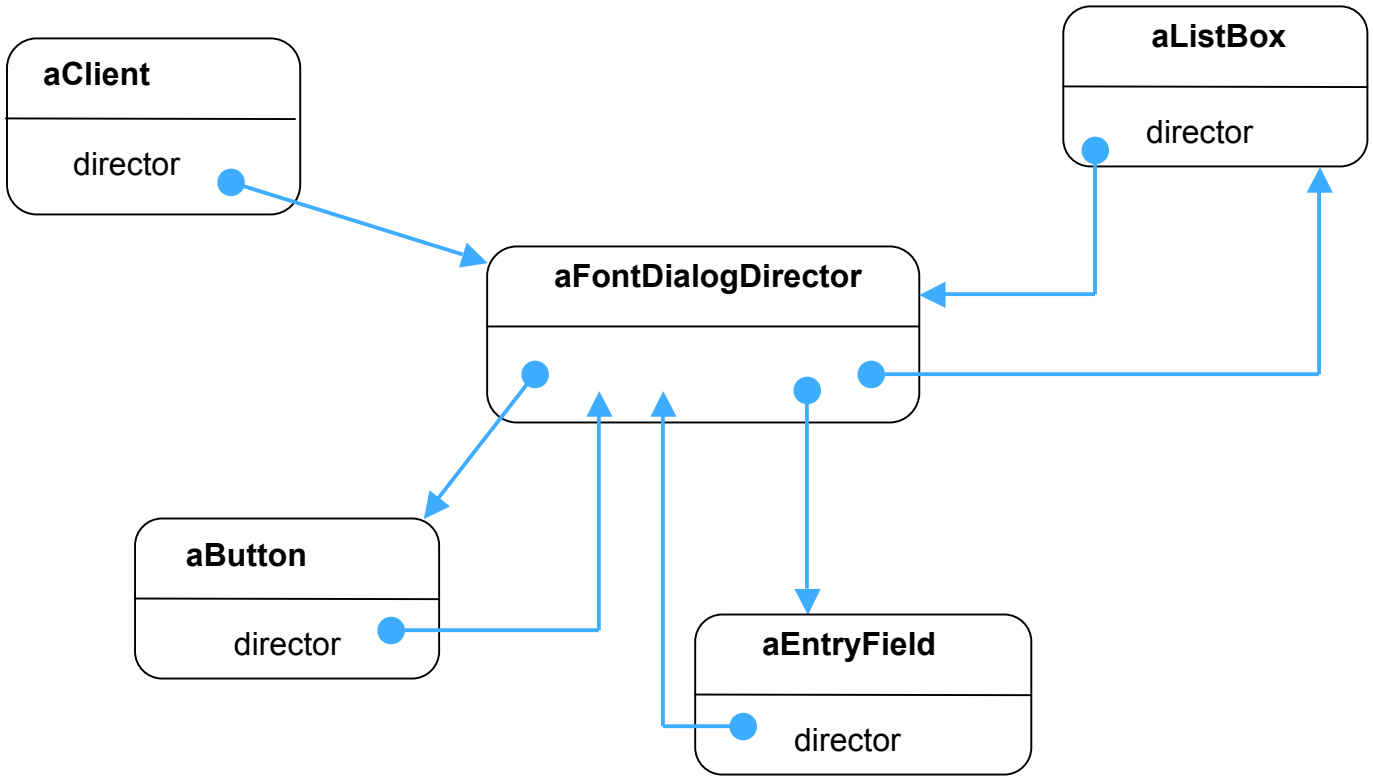
- Intent:

Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.

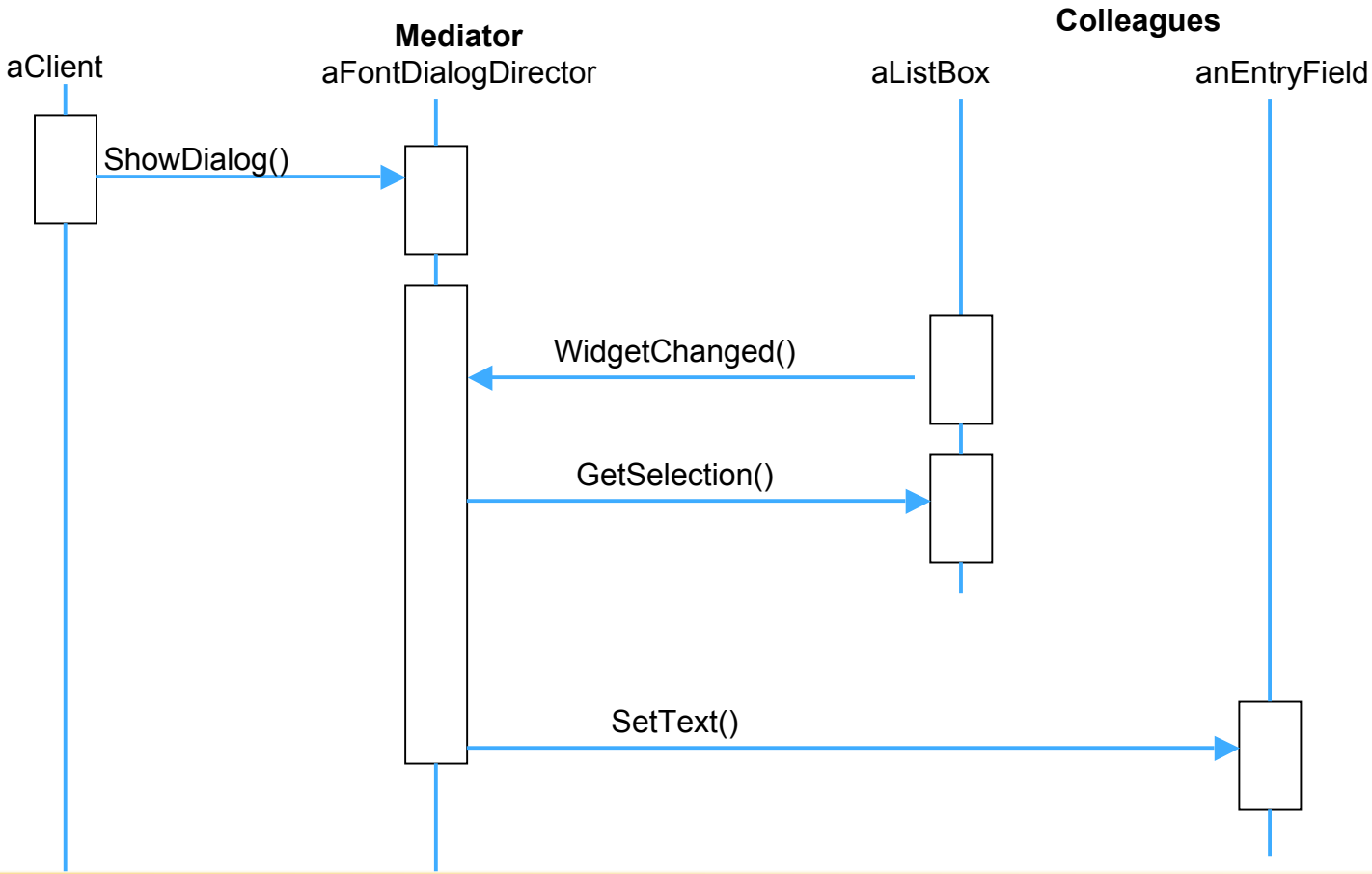
MEDIATOR
Motivation



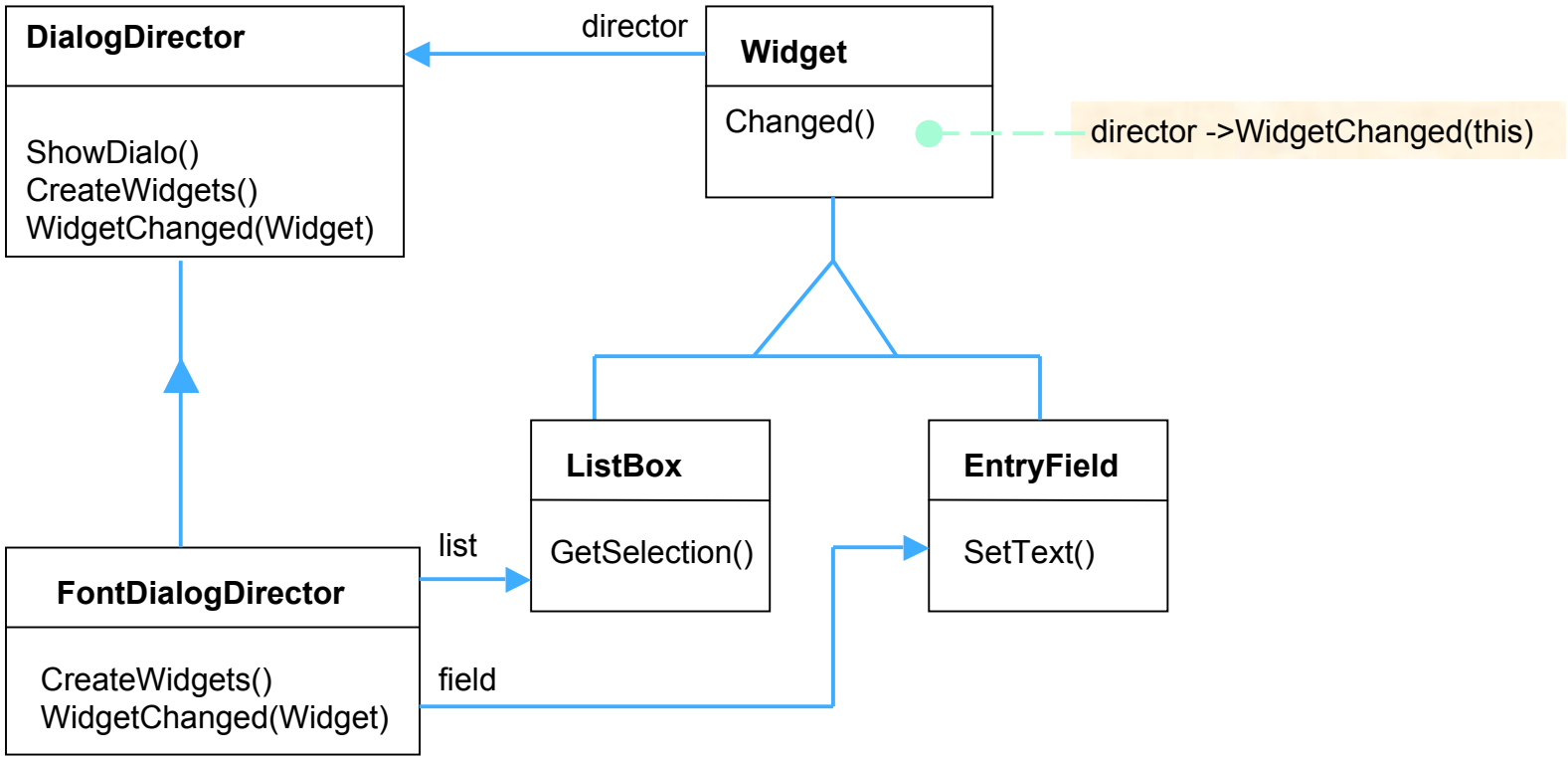
MEDIATOR
Motivation



MEDIATOR
Motivation



MEDIATOR
Motivation

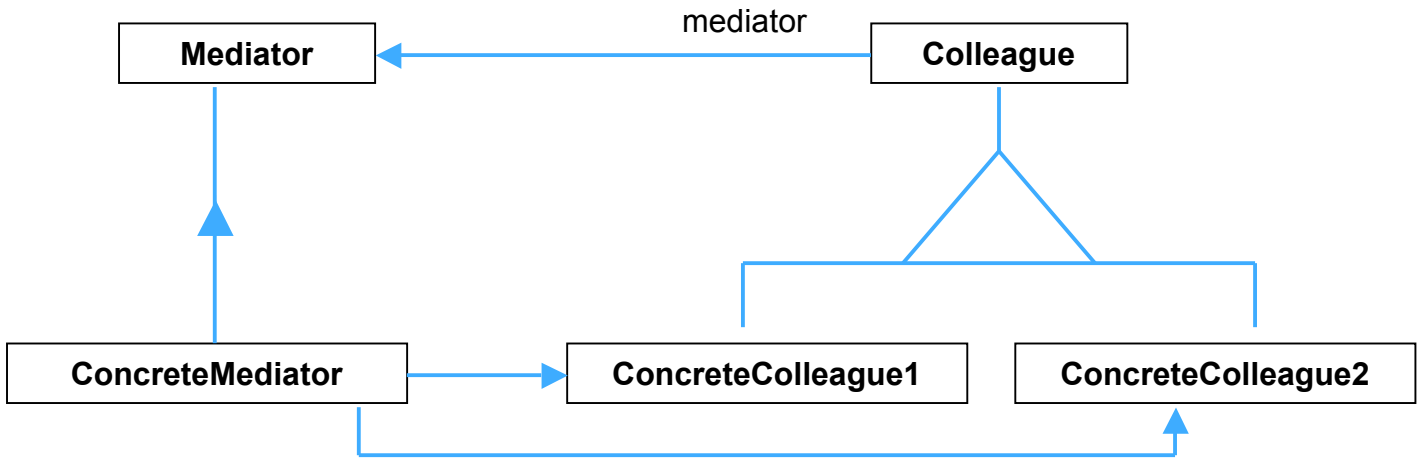


Applicability

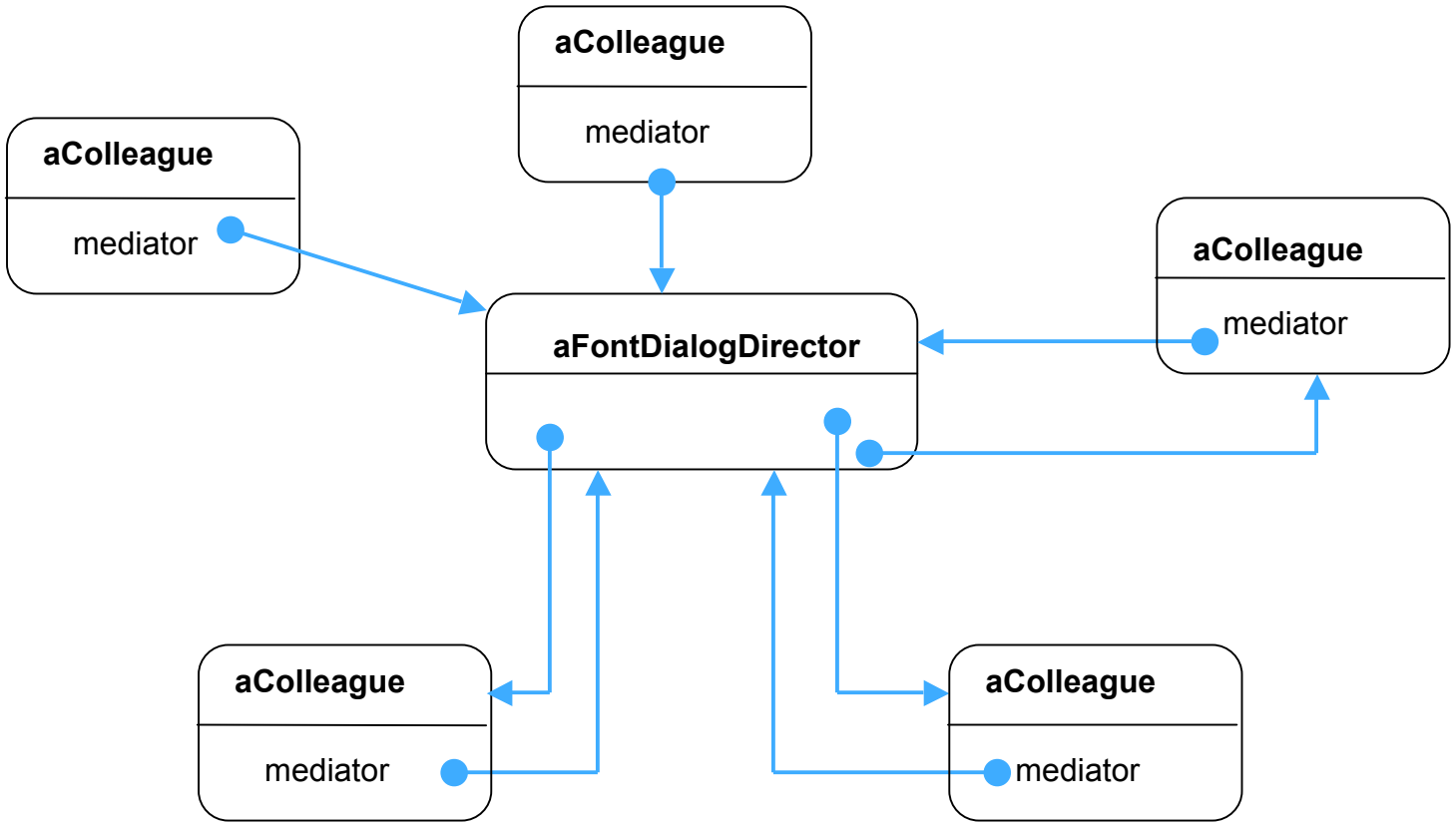
Use the Mediator pattern when

- 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.

MEDIATOR
Structure



MEDIATOR
Structure



Collaborations

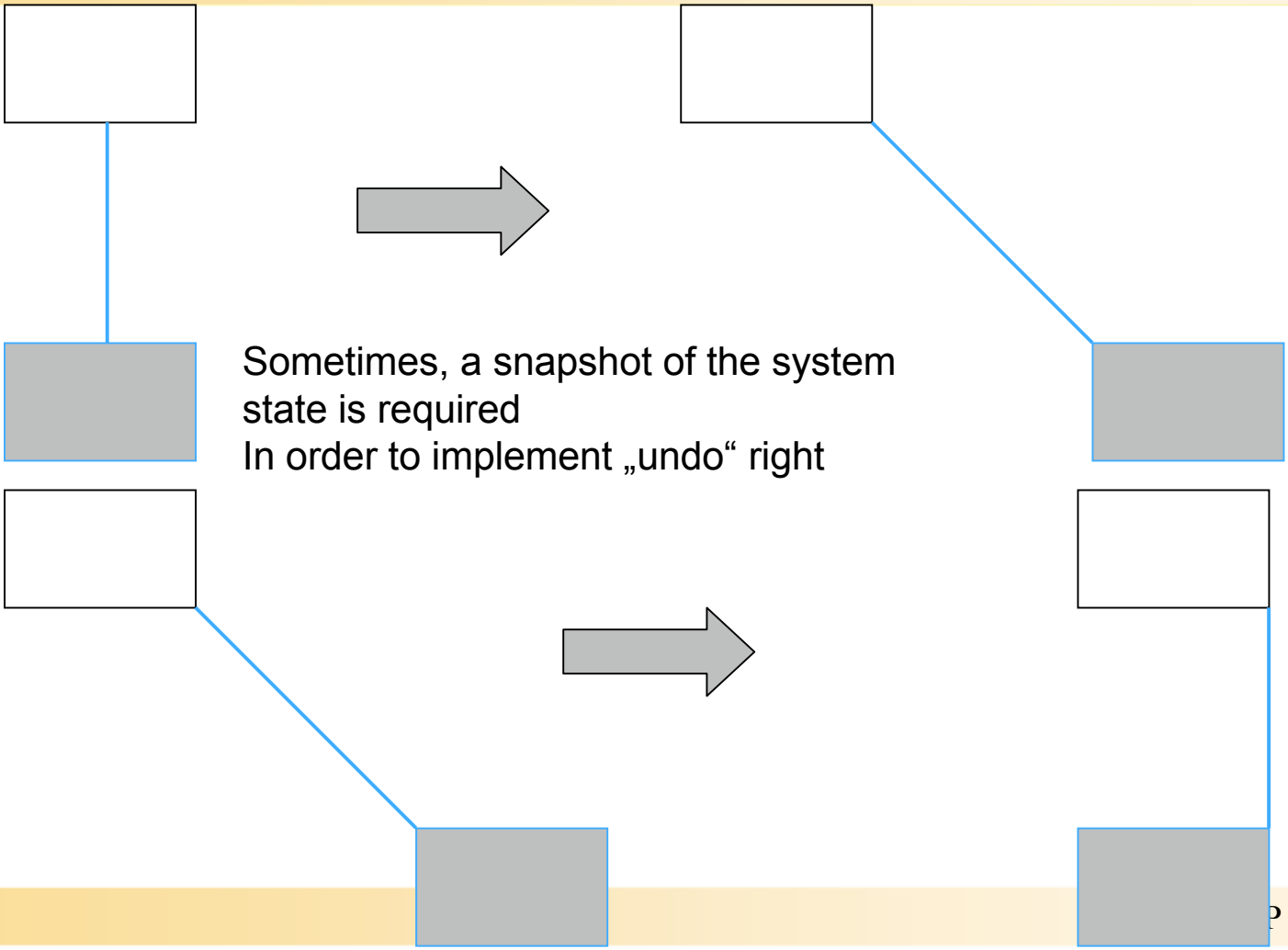
- Colleagues send and receive requests from a Mediator object. The mediator implements the cooperative behavior by routing requests between the appropriate colleague(s).

MEMENTO
(Object Behavioral)

Intent:

- Without violating encapsulation, capture and externalize an object´s internal state so that the object can be restored to this state later.
- Also known as: Token

MEMENTO
Motivation



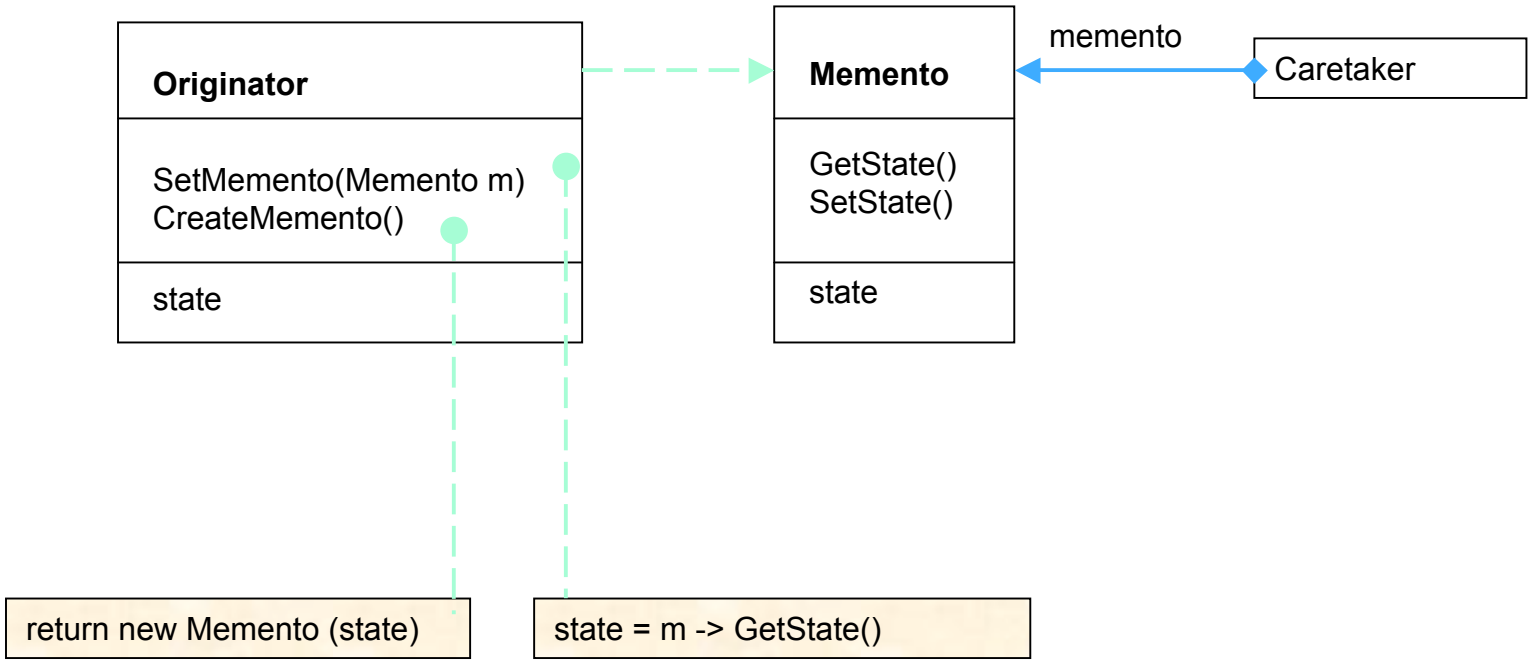
Sometimes, a snapshot of the system state is required
In order to implement „undo“ right

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.

MEMENTO
Structure



Collaborations

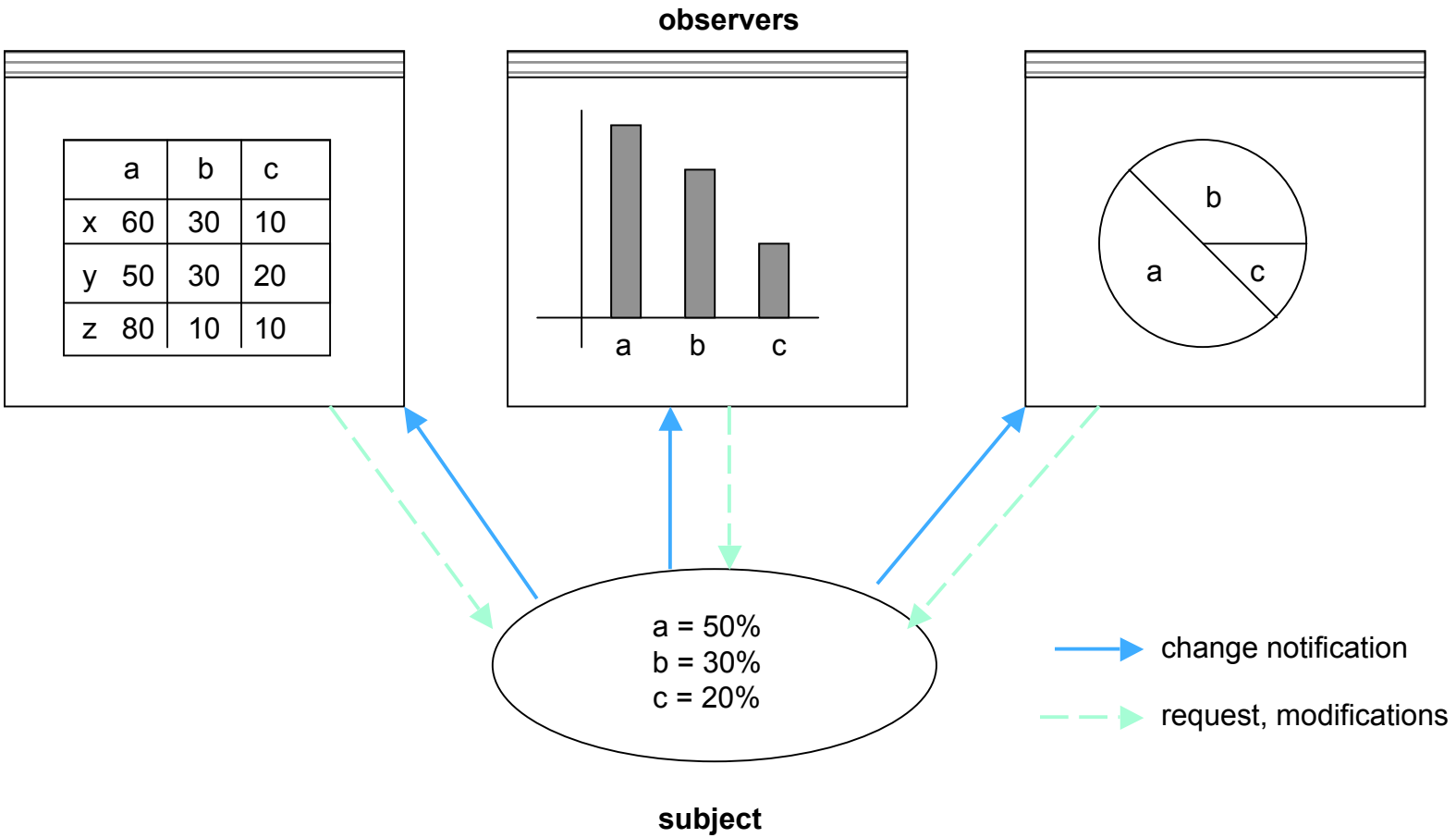
- A caretaker requests a memento from an originator, holds it for a time, and passes it back to the originator
- Sometimes the caretaker won't pass the memento back to the originator, because the originator might never need to revert to an earlier state
- Mementos are passive. Only the originator that created a memento will assign or retrieve its state

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.

OBSERVER
Motivation

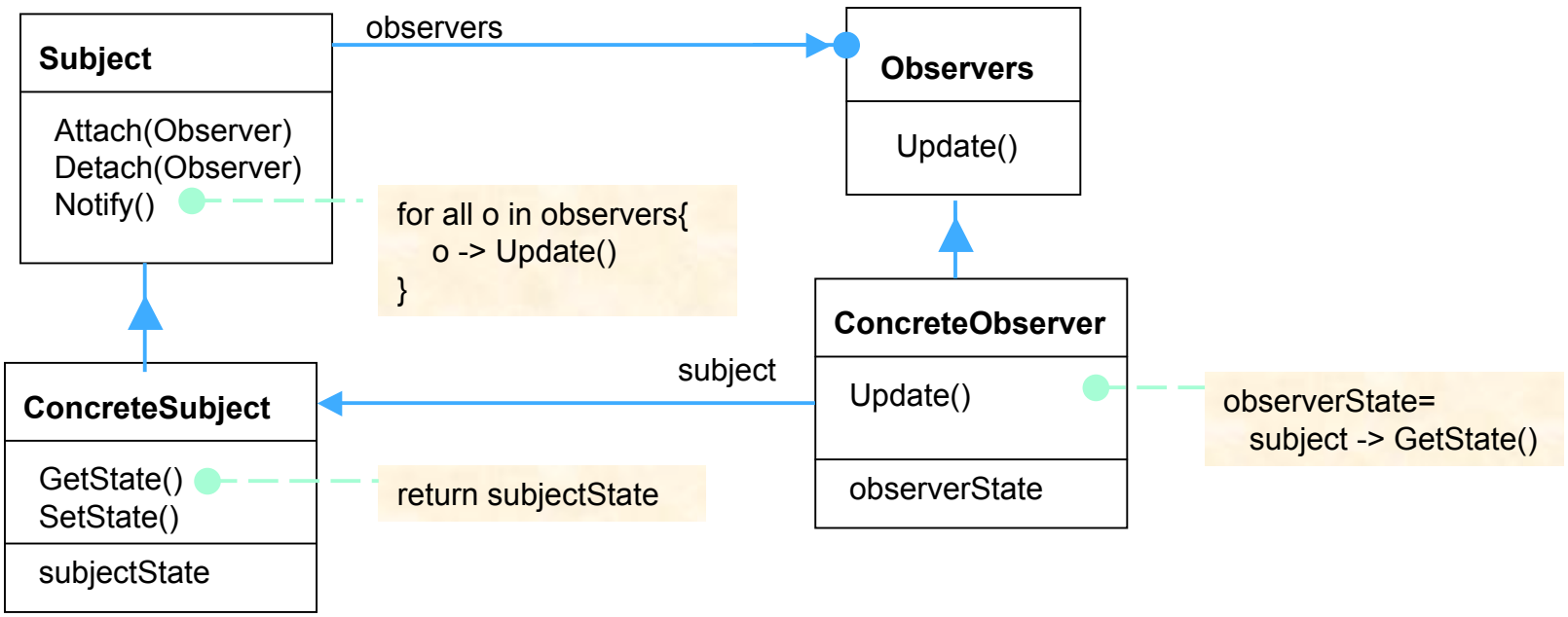


Applicability

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.

OBSERVER
Structure



Participants

- Subject:
  - Knows its observers. Any number of Observer any observe a subject
  - Provides an interface for attaching/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

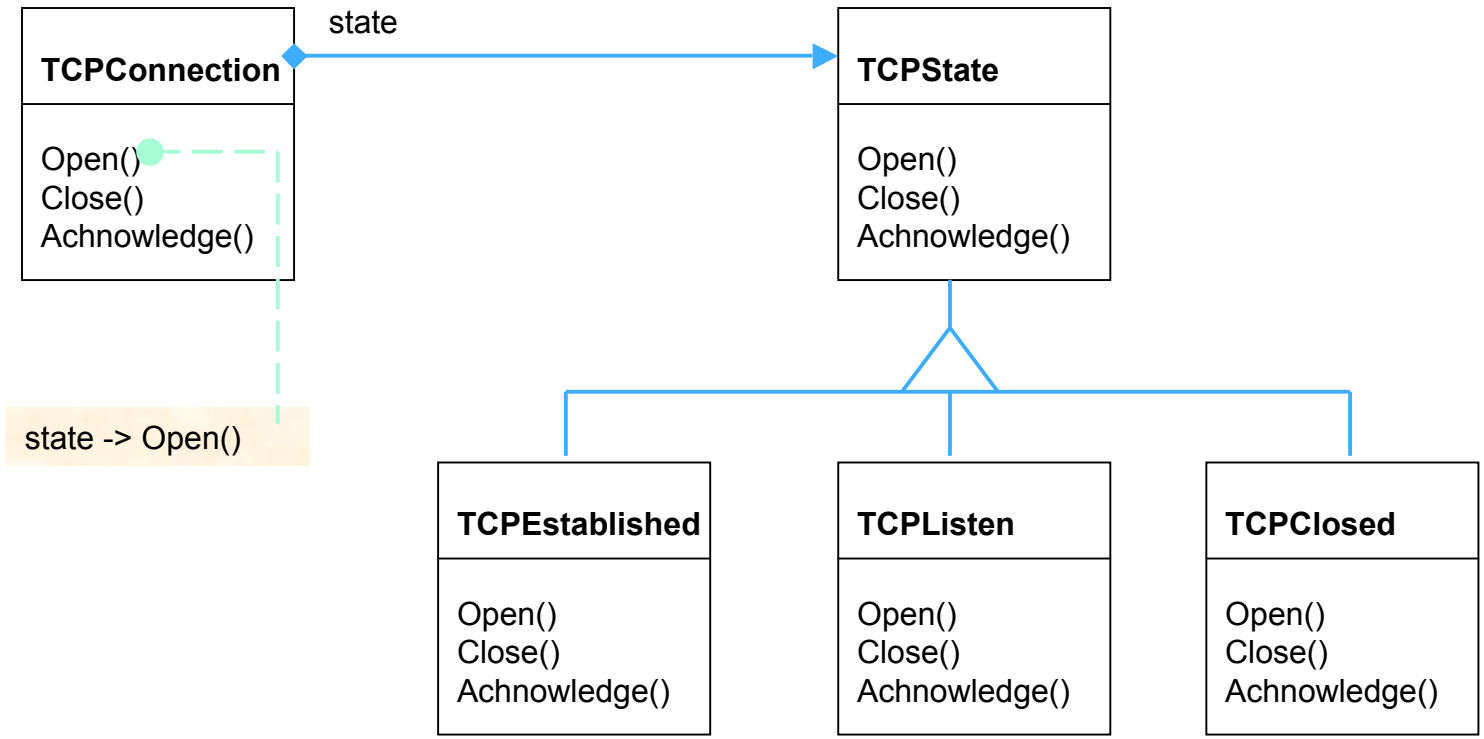
- ConcreteSubject notifies its observers whenever a change occurs that could make its observers´state inconsistent with its own.
- After being informed of a change in the concrete subject, a ConcreteObserver object may query the subject for information. ConcreteObserver uses this information to reconcile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

STATE
(Object Behavioral)

- Intent:

Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.

STATE
Motivation

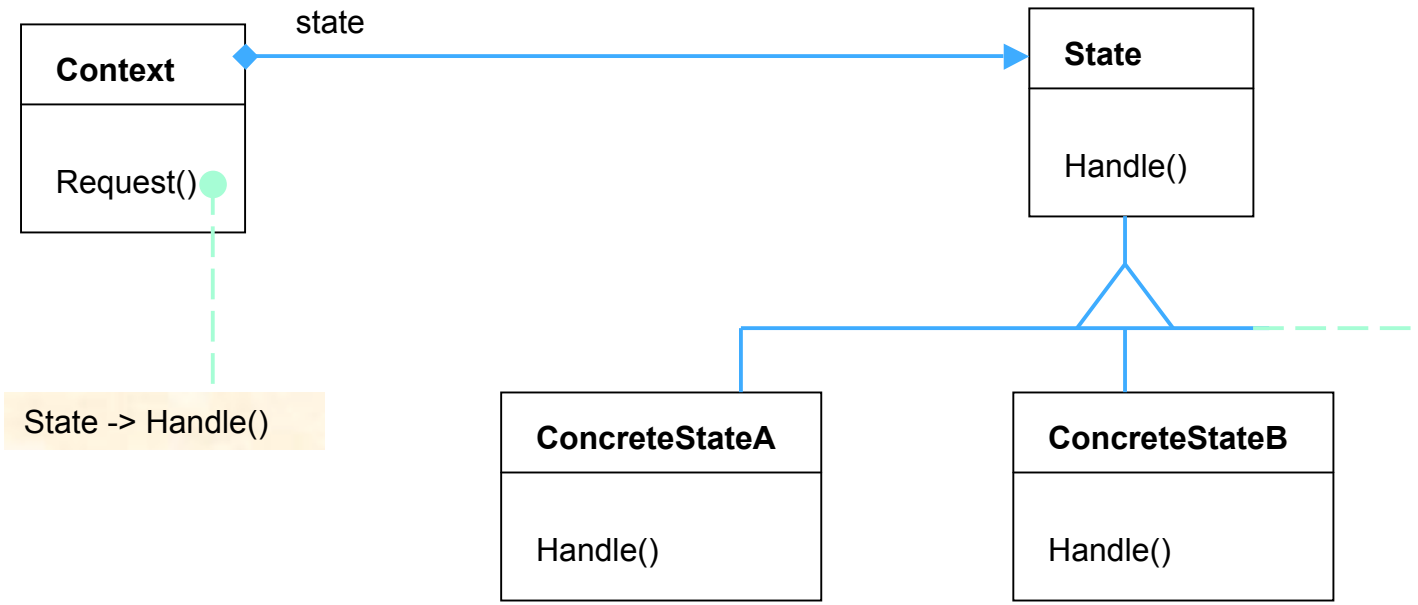


Applicability

Use the State pattern in either of the following cases:

- An object´s behavior depends on its state, and it must change its behavior at run-time depending on that state.
- Operations have large, multipart conditional statements that depend on the object´s state. This state is usually represented by one or more enumerated constants. Often, several operations will contain this same conditional structure. The State pattern puts each branch of the conditional in a separate class. This lets you treat the object´s state as an object in its own right that can vary independently from other objects.

STATE
Structure



Collaborations

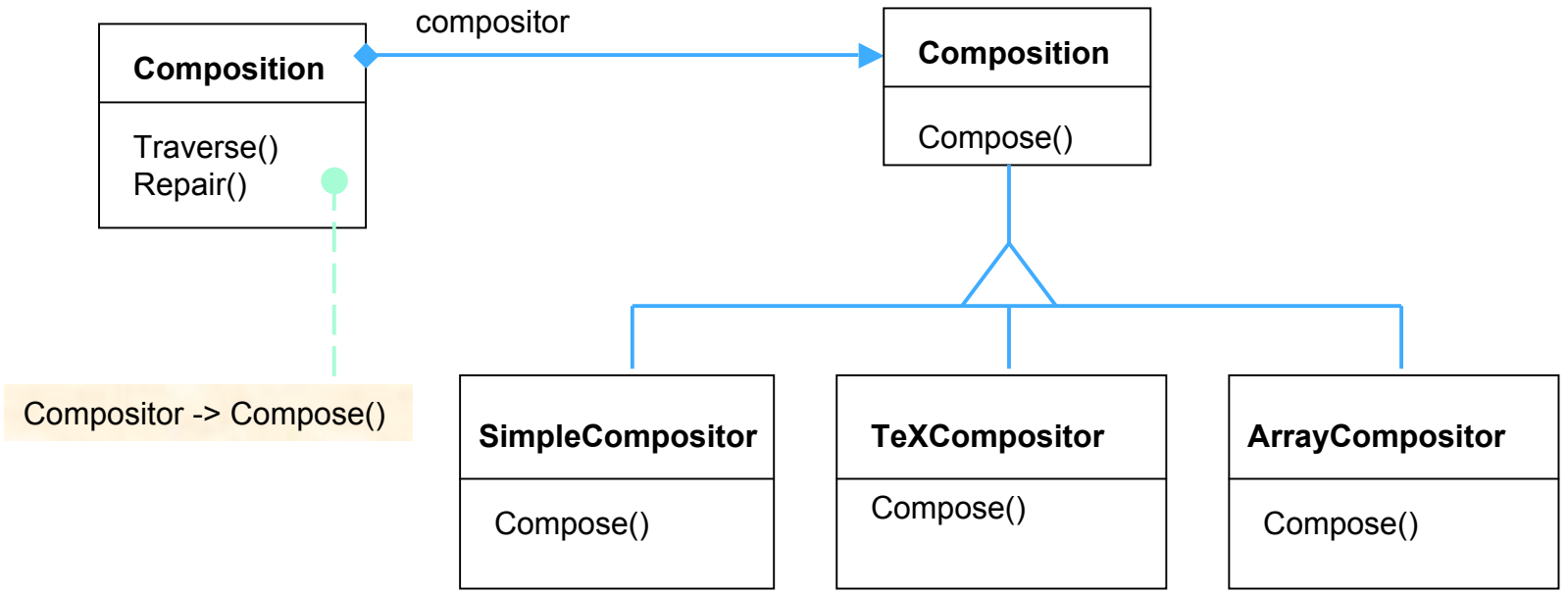
- Context delegates state-specific requests to the current Concrete State object.
- A context may pass itself as an argument to the State object handling the request. This lets the State object access the context if necessary.
- Context is the primary interface for clients. Clients can configure a context with State objects. Once a context is configured, its clients don´t have to deal with the State objects directly.
- Either Context or the Concrete State subclasses can decide which state succeeds another and under what circumstances.

STRATEGY
(Object Behavioral)

- Intent:

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

STRATEGY
Motivation

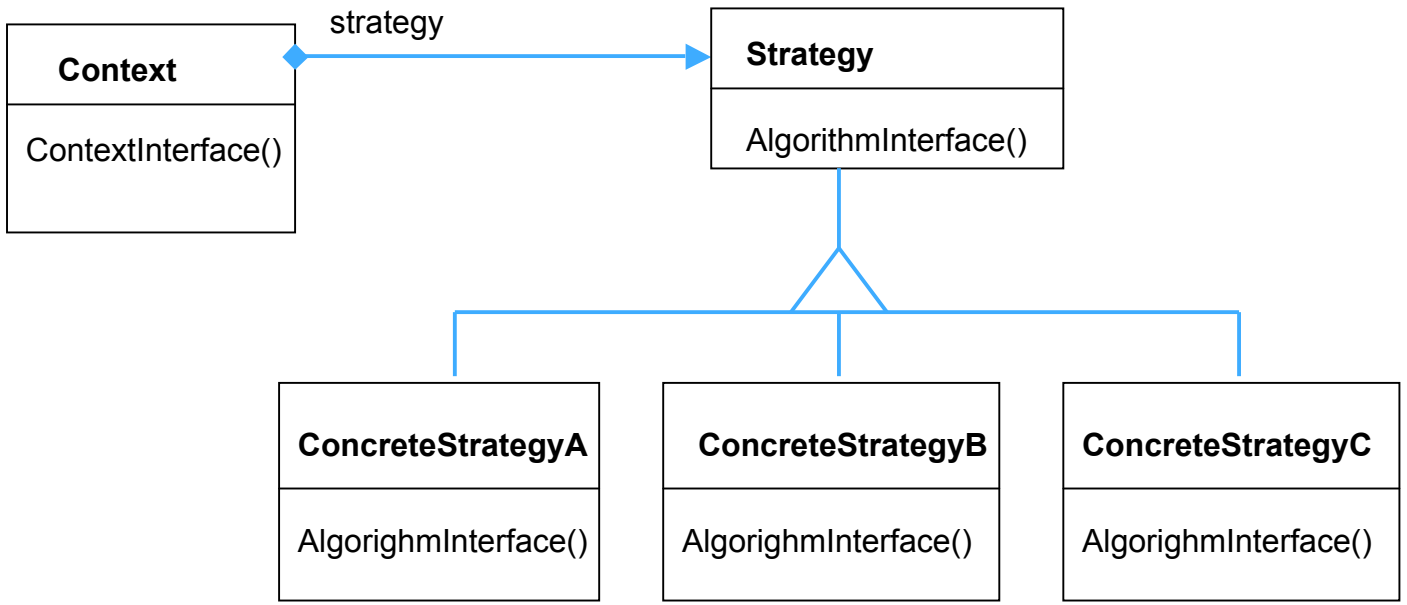


Applicability

Use the Strategy pattern when

- May related classes differ only in their behavior. Strategies provide a way to configure a class with one of many behaviors.
- You need different variants of an algorithm. For example, you might define algorithms reflecting different space/time trade-offs. Strategies can be used when these variants are implemented as a class hierarchy of algorithms [HO87].
- An algorithm uses data that clients shouldn't know about. Use the Strategy pattern to avoid exposing complex, algorithm-specific data structures.
- A class defines many behaviors, and these appear as multiple conditional statements in its operations. Instead of many conditionals, move related conditional branches into their own Strategy class.

STRATEGY
Structure



Collaborations

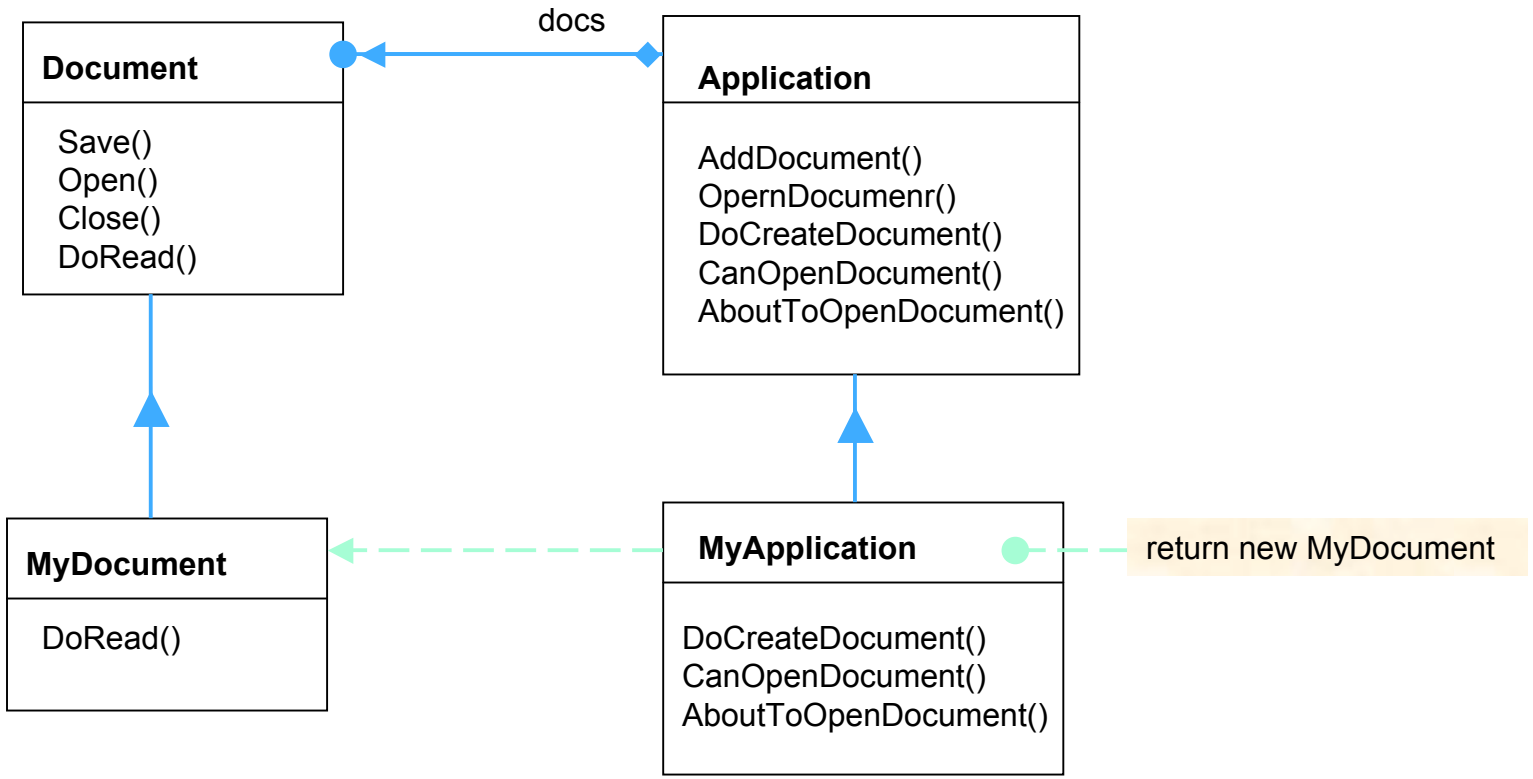
- Strategy and Context interact to implement the chosen algorithm. A context may pass all data required by the algorithm to the strategy when the algorithm is called. Alternatively, the context can pass itself as an argument to Strategy operations. That lets the strategy call back on the context as required.
- A context forwards requests from its clients to its strategy. Clients usually create and pass a ConcreteStrategy object to the context; thereafter, clients interact with the context exclusively. There is often a family of ConcreteStrategy classes for a client to choose from.

TEMPLATE METHOD
(Class Behavioral)

- Intent:

Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm´s structure.

TEMPLATE METHOD
Motivation

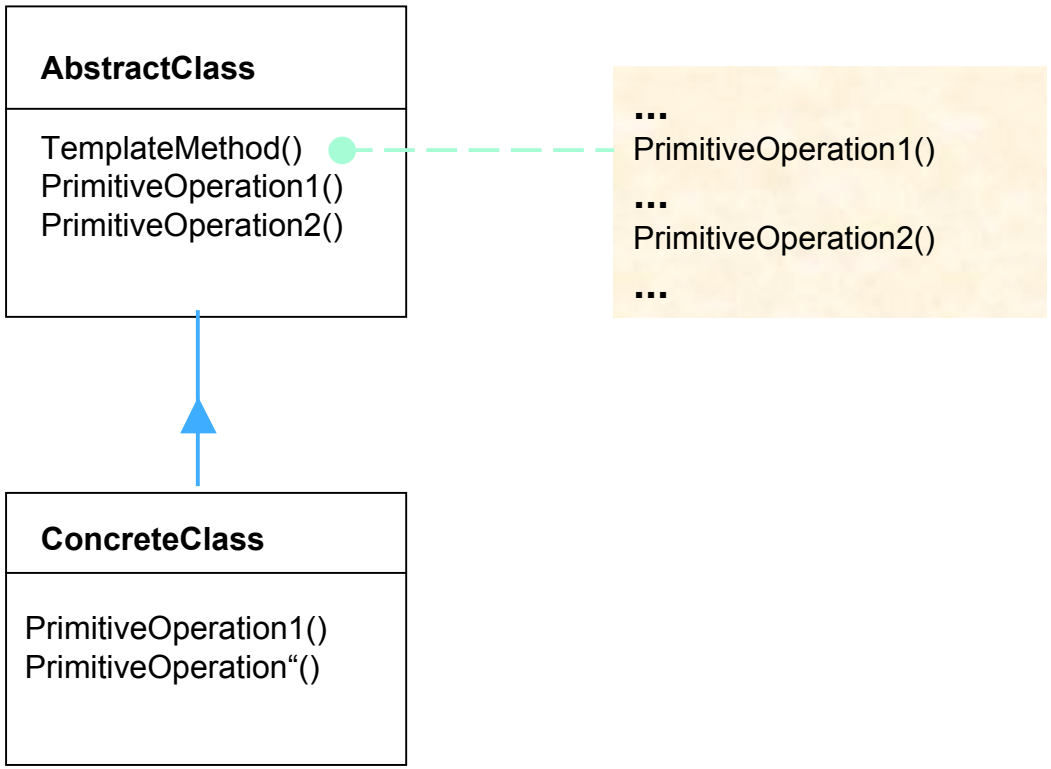


Applicability

The Template Method pattern should be used

- To implement the invariant parts of an algorithm once and leave it up to subclasses to implement the behavior that can vary.
- When common behavior among subclasses should be factored an localized in a common class to avoid code duplication. This is a good example of “refactoring to generalize” as described by Opdyke and Johnson [OJ93].You first identify the differences in the existing code and then separate the differences into new operation. Finally, you replace the differing code with a template method that calls one of these new operations.
- To control subclasses extensions. You can define a template method that calls “hook” operations (see Consequences) at specific points, therevy permitting extensions only at those points.

TEMPLATE METHOD
Structure



Collaborations

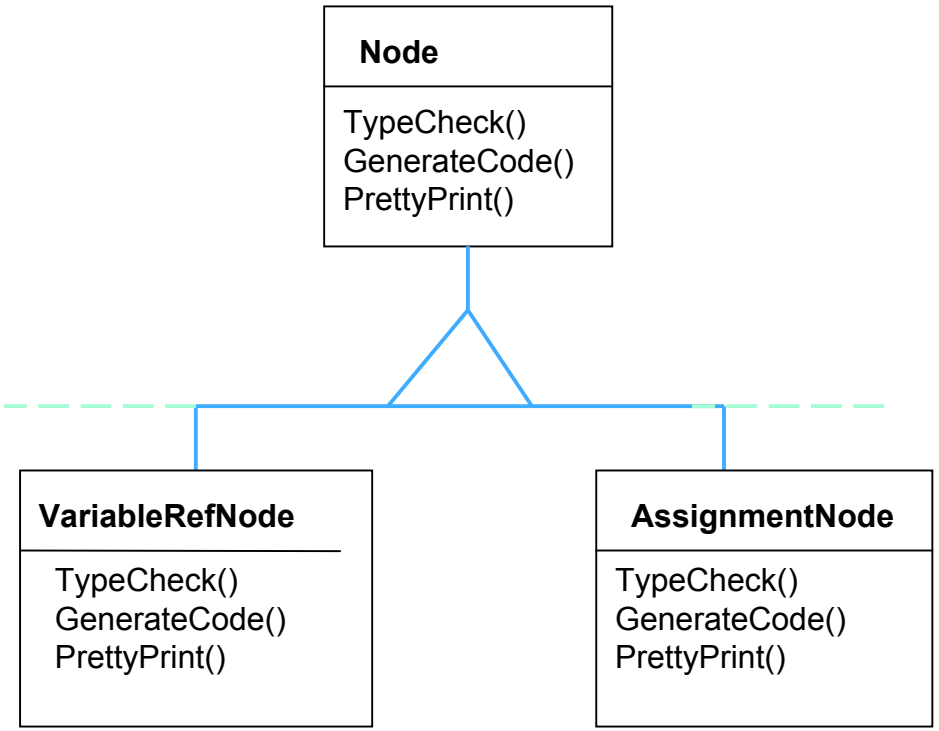
- ConcreteClass relies on AbstractClass to implement the invariant steps of the algorithm.

VISITOR
(Object Behavioral)

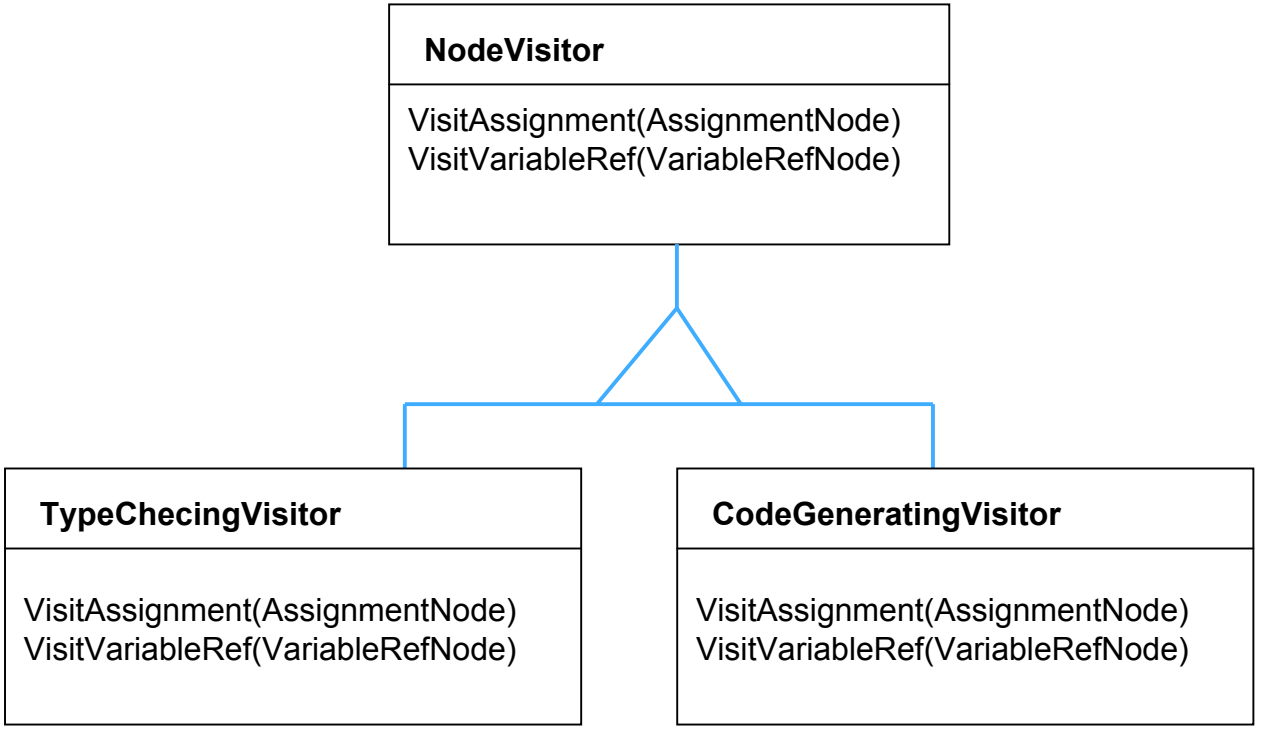
- Intent:

Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.

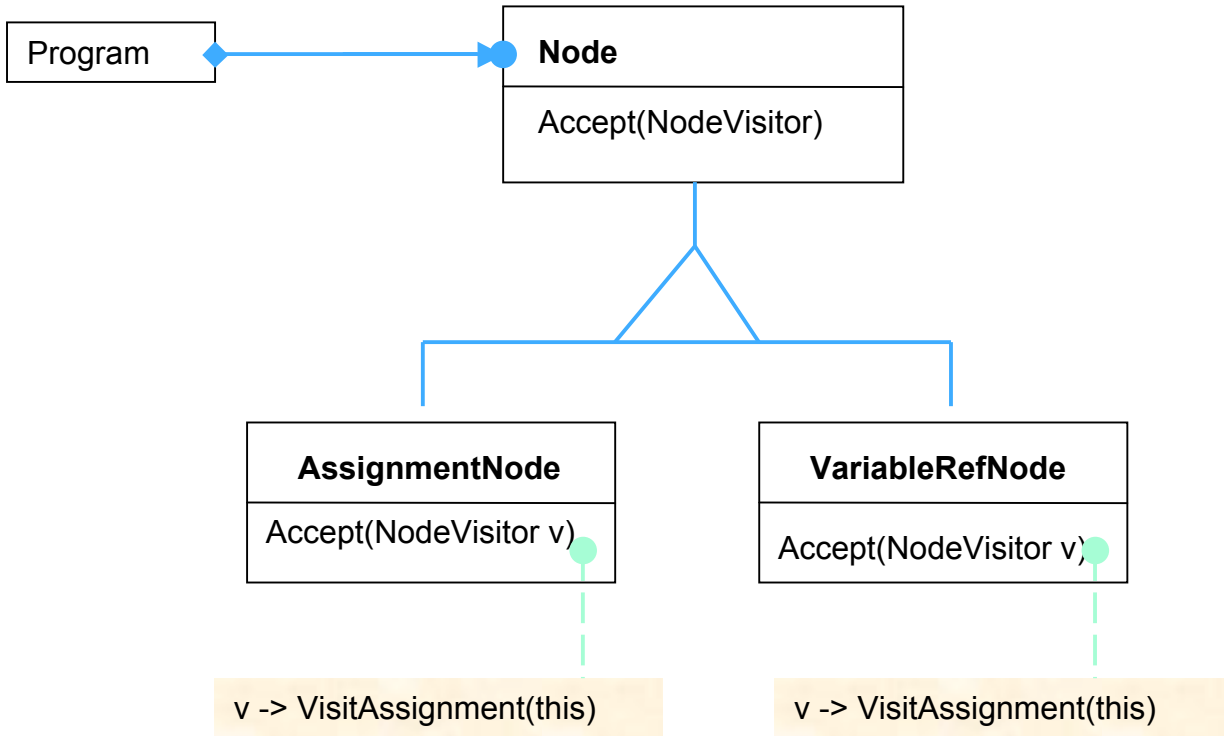
VISITOR
Motivation



VISITOR
Motivation



VISITOR
Motivation

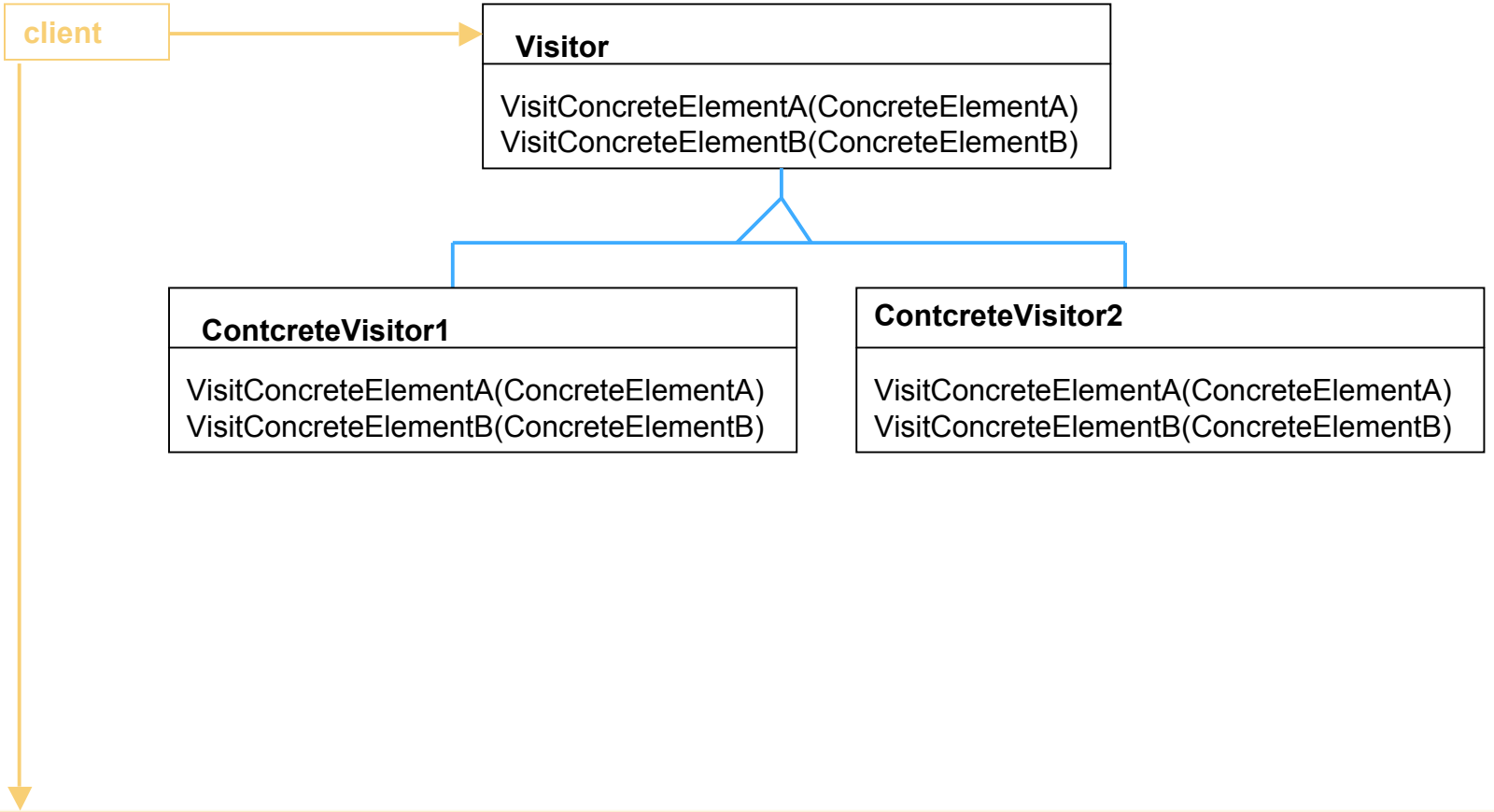


Applicability

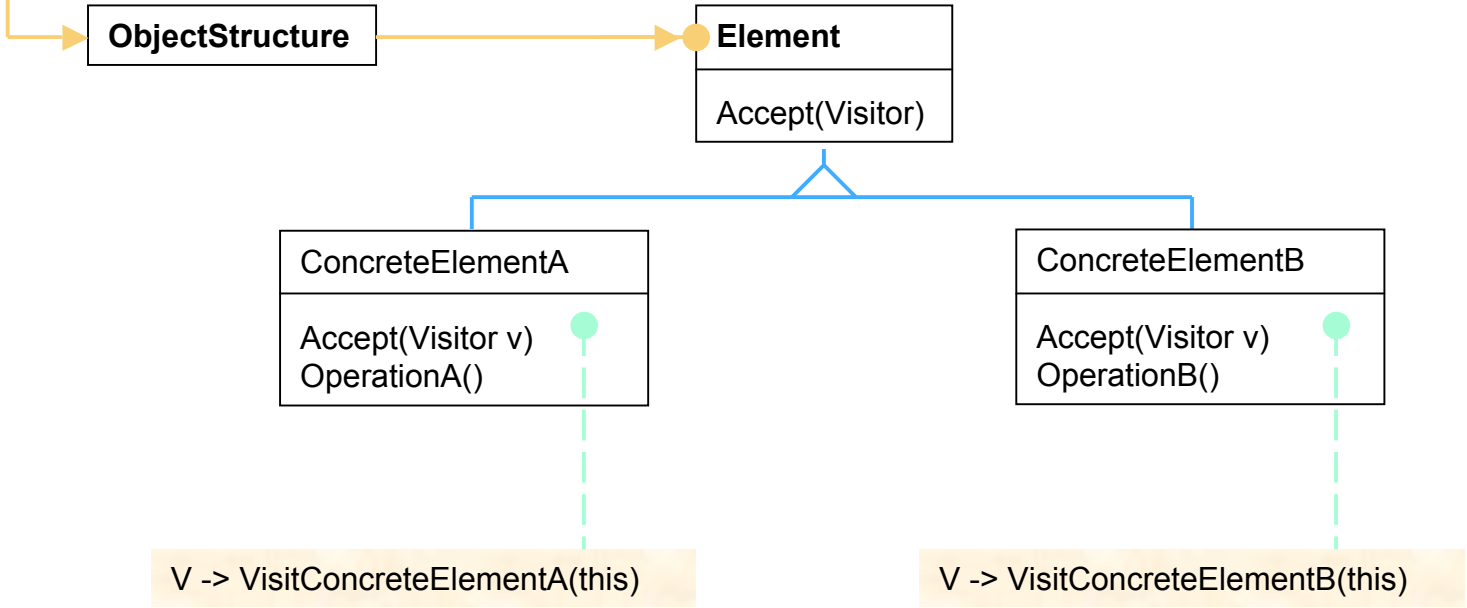
Use the Visitor pattern when

- An object structure contains many classes of objects with differing interfaces, and you want to perform operations on these objects that depend on their concrete classes.
- Many distinct and unrelated operations need to be performed on objects in an object structure, and you want to avoid “polluting” their classes with these operations. Visitor lets you keep related operations together by defining them in one class. When the object structure is shared by many applications, use Visitor to put operations in just those applications that need them.
- The classes defining the object structure rarely change, but you often want to define new operations over the structure. Changing the object structure classes requires redefining the interface to all visitors, which is potentially costly. If the object structure classes change often, then it´s probably better to define the operations in those classes.

VISITOR
Structure (I)



VISITOR
Structure (II)



Collaborations

- A client that uses the Visitor pattern must create a ConcreteVisitor object and then traverse the object structure, visiting each element with the visitor.
- When an element is visited, it calls the Visitor operation that corresponds to its class. The element supplies itself as an argument to this operation to let the visitor access its state, if necessary. The following interaction diagram illustrates the collaborations between an object structure, a visitor, ant two elements: