# Design Patterns (II)



## 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>Facade<br>Flyweight<br>Proxy | Chain of Responsibility<br>Command<br>Iterator<br>Mediator<br>Memento<br>Observer<br>State<br>Strategy<br>Visitor |

Defer object creation to another class

Defer object creation to another object

Describe ways to assemble objects

Describe algorithms and flow control

# Behavioral Patterns

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

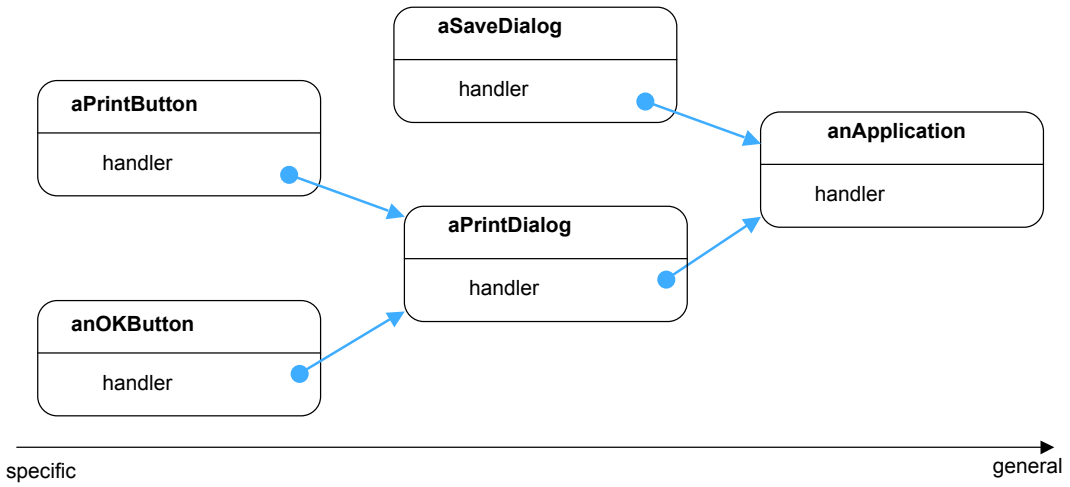
- Behavioral **class** patterns
  - Use inheritance to distribute behavior among classes
- Behavioral **object** patterns
  - Use object composition rather than inheritance
  - Describe how groups of peer objects cooperate for a task
  - Patterns on how peer objects know each other



# CHAIN OF RESPONSIBILITY (Object Behavioral)

- Intent:
  - Avoid coupling the sender of a request to its receiver
  - Give more than one object a chance to handle a request
  - Chain the receiving objects
  - Pass the request along until an Object handles it
- Motivation:
  - Example: Context-sensitive help facility for a GUI
    - Users can obtain help info on any widget
    - Help provided depends on the chosen widget and its context
  - Object that provides help is not directly known to object (e.g. button) that initiates the request
  - Decouple senders and receivers of requests

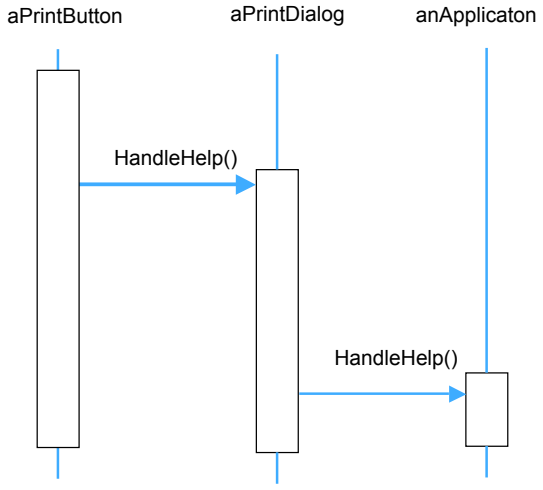
# CHAIN OF RESPONSIBILITY
## Motivation



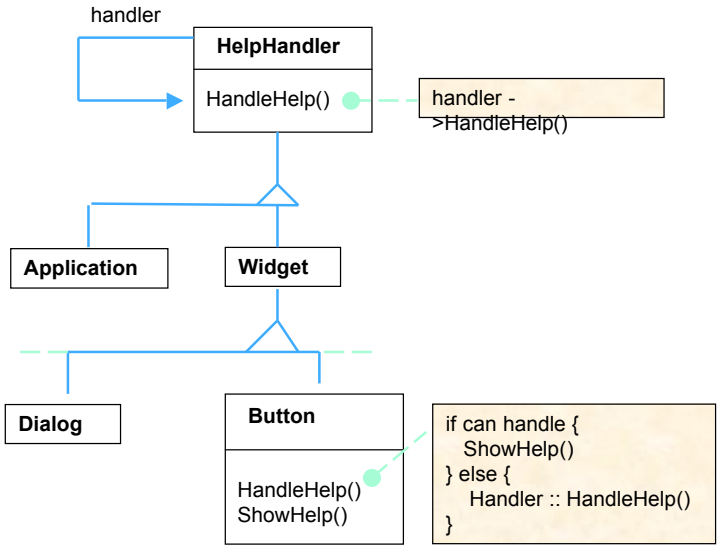



# CHAIN OF RESPONSIBILITY
## Motivation

- Object in the chain receives the request
- Handles it or forwards it
- Requestor has „implicit receiver“ for the request
- Final receiver handles or ignores the request



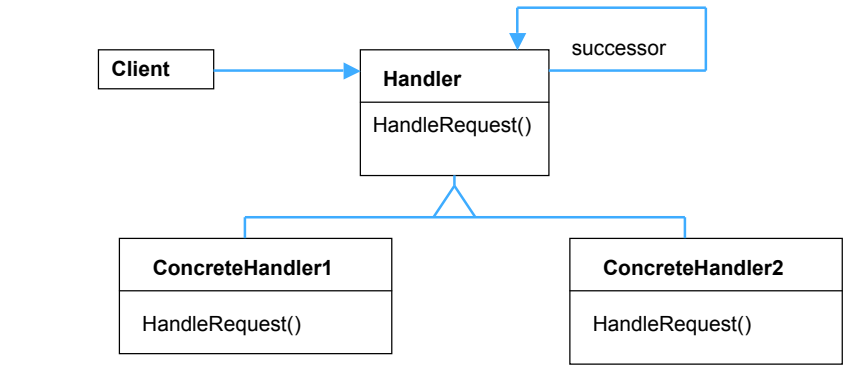
# CHAIN OF RESPONSIBILITY
## Motivation

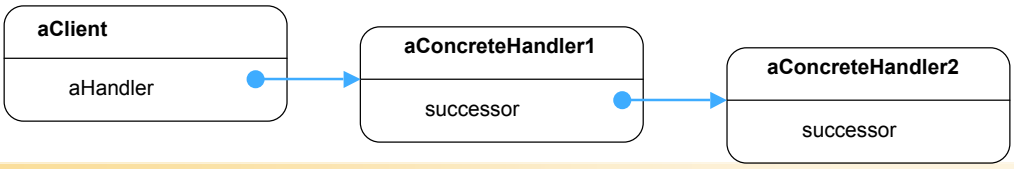



- HelpHandler forwards requests by default
- Subclasses can override operation
- Requests are fulfilled in the subclass, or handled by the default implementation



# CHAIN OF RESPONSIBILITY
## Structure



# CHAIN OF RESPONSIBILITY Participants

- Handler (HelpHandler)
  - Defines an interface for handling requests
  - (optional) implements the successor link
- ConcreteHandler (PrintButton, PrintDialog)
  - Handles requests it is responsible for
  - Either handles requests or forwards it to its successor, usually through the *Handler*
- Client
  - Initiates the request to a *ConcreteHandler* object on the chain



# CHAIN OF RESPONSIBILITY Applicability / Benefits

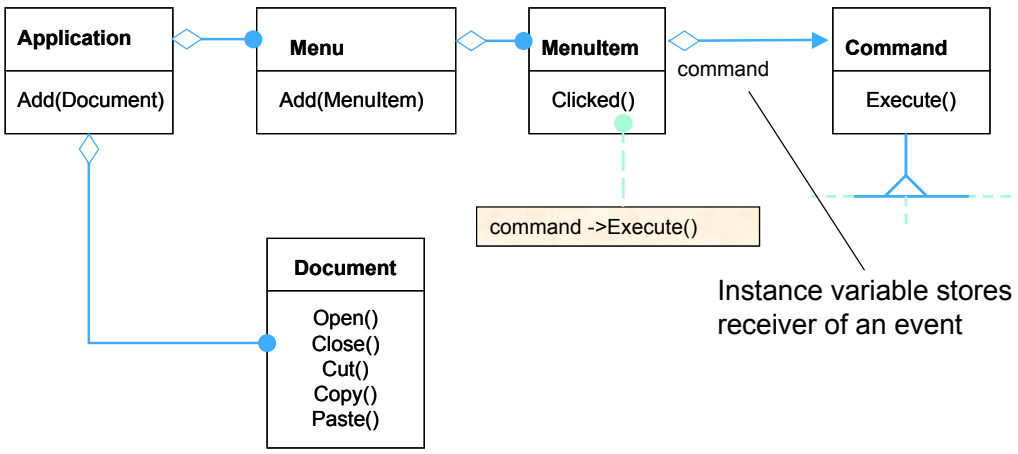
- Use Chain of Responsibility when:
  - More than one object may handle a request
  - The handler is not known a priori
  - The handler should be identified automatically
  - You don't want specify the receiver explicitly
  - The handler objects are specified dynamically
- Benefits:
  - Reduced coupling
    - Sender and receiver have no explicit knowledge of each other
    - Single reference to successor
  - Flexible assignment of object responsibilities

# COMMAND
# (Object Behavioral)

- Intent:
  - Encapsulate a request as an object
    - Parameterize clients with different requests (queue or log requests)
    - Support undoable operations (Transactions)
  - Decouple requesting object from performing object
- Motivation:
  - Decouple GUI toolkit request from
    - Operation being requested
    - Receiver of the request
  - Abstract command class
    - interface for executing operations
    - Requestor does not know which Command subclass is used
  - Concrete Command subclass specifies receiver-action pair
    - Final receiver as instance variable

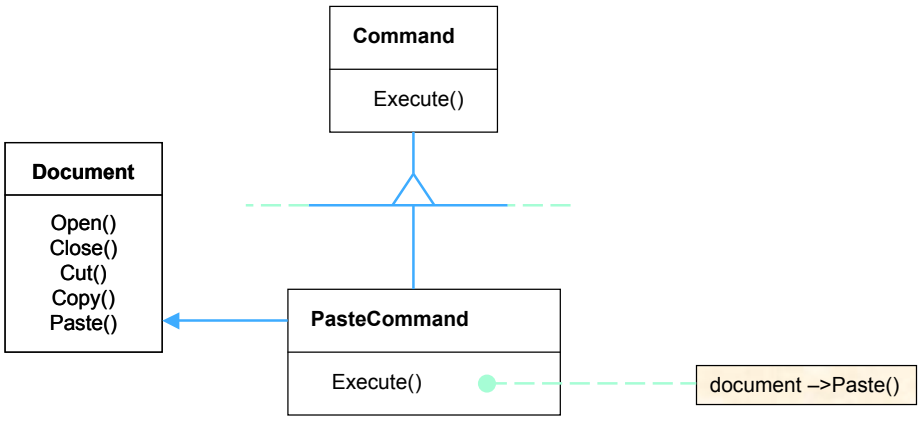


# COMMAND
# Motivation




- Each possible choice in a *Menu* is an instance of a *MenuItem* class
- *Application* creates menus and their menu items

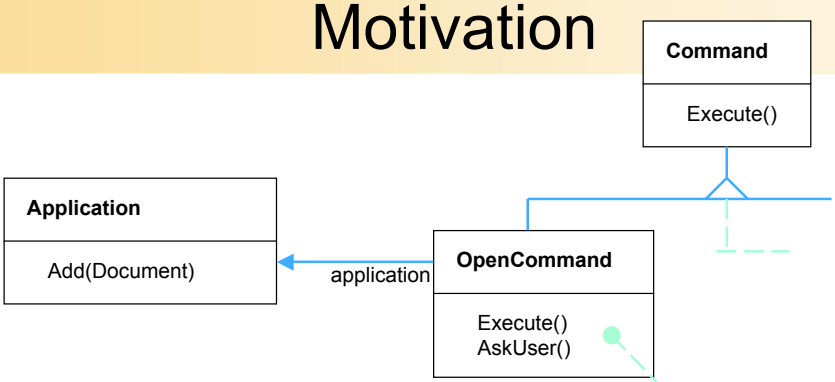
# COMMAND
# Motivation




- *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.



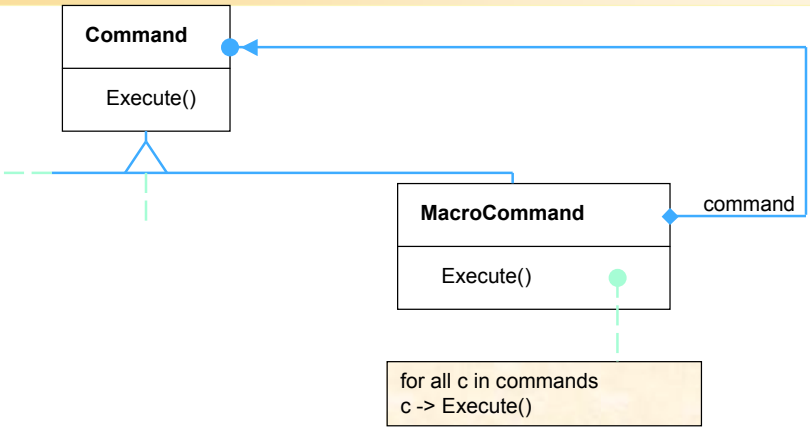
# COMMAND
# Motivation




- *OpenCommand's* operation *Execute*
  - prompts the user for a name
  - creates the corresponding document object
  - adds document to the receiving app
  - opens the document

```
name = AskUser()
doc = new Document(name)
application -> Add(doc)
doc -> Open()
```

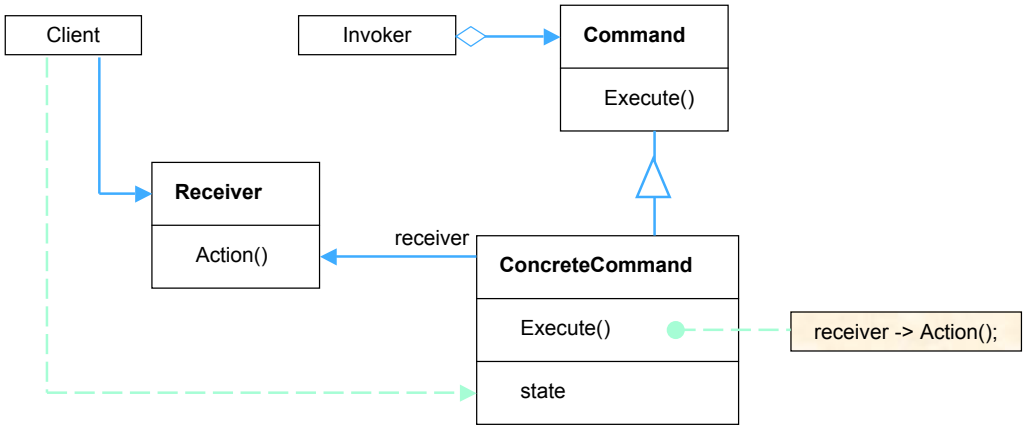
# COMMAND
# Motivation




- *MacroCommand* is a concrete *Command* subclass
- Executes a sequence of commands
- *MacroCommand* has no explicit receiver – Command objects in the sequence define their own receivers.



# COMMAND
# Structure

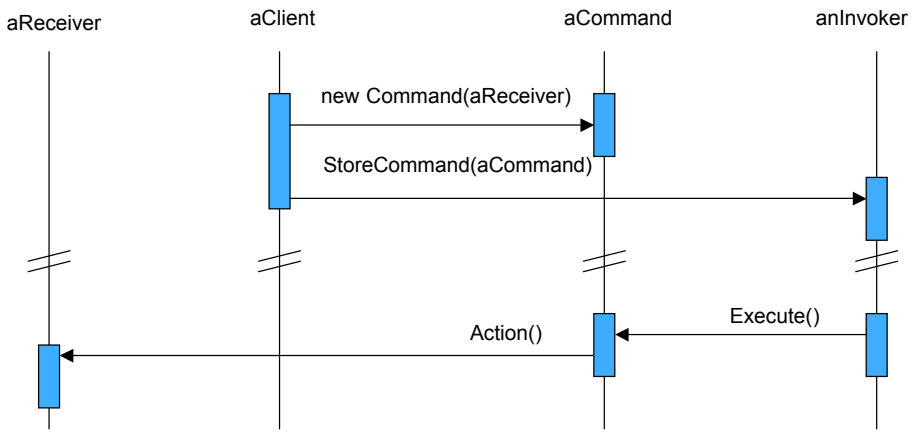


# COMMAND
# Participants

- Command
  - Declares interface for operation execution
- ConcreteCommand (PasteCommand, OpenCommand)
  - Defines binding between *Receiver* and an action
  - Implements operation execution by invoking *Receiver*
- Client (Application)
  - Creates a *ConcreteCommand* object and sets the *Receiver*
- Invoker (MenuItem)
  - Asks the *Command* to carry out the request (stores *ConcreteCommand*)
- Receiver (Document, Application)
  - Knows how to perform the operation(s) for a request



# COMMAND
# Interaction Between Objects



# COMMAND
# Applicability

Use the Command pattern when you want to:

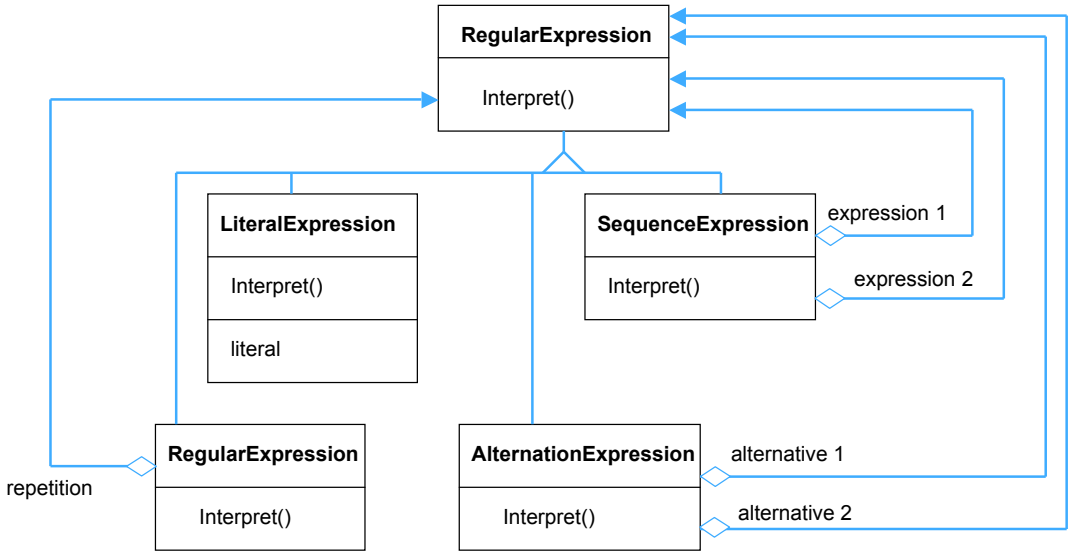
- Parameterize objects by an action to perform
  - Commands = OO replacement for callback function registration
- Decouple request specification and execution
  - *Command* object's lifetime is independent from original request
  - *Command* object might be transferred to another process
- Implement undo operation
  - Command object maintains state information for reversing its effects
  - Additional *Unexecute()* operation
  - Saving / loading operations for state allows crash fault tolerance
- Model transactional behavior
  - Encapsulation of set of data changes



# INTERPRETER
# (Class Behavioral)

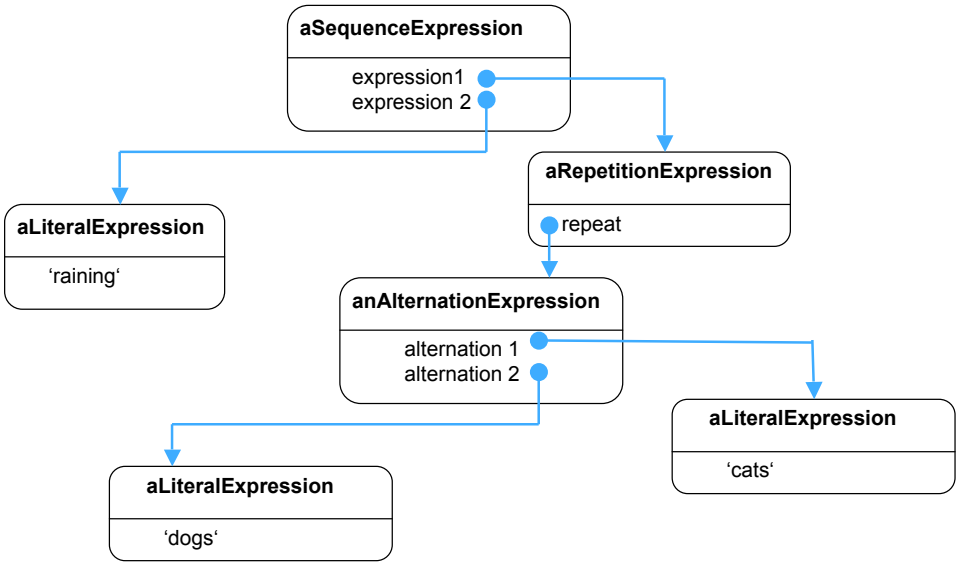
- Intent:
  - Define representation of language through its grammar
  - Build something that uses the representation to interpret sentences
- Motivation:
  - Interpreter for problems represented through language sentences
  - Example: Regular expressions
    - Implementation of search algorithms uses given pattern language
  - Example grammar
    - `expression ::= literal | alternation | sequence | repetition | '(' expression ')'`
    - `alternation ::= expression '|' expression`
    - `sequence ::= expression '&' expression`
    - `Repetition ::= expression '*'`

# INTERPRETER
# Motivation



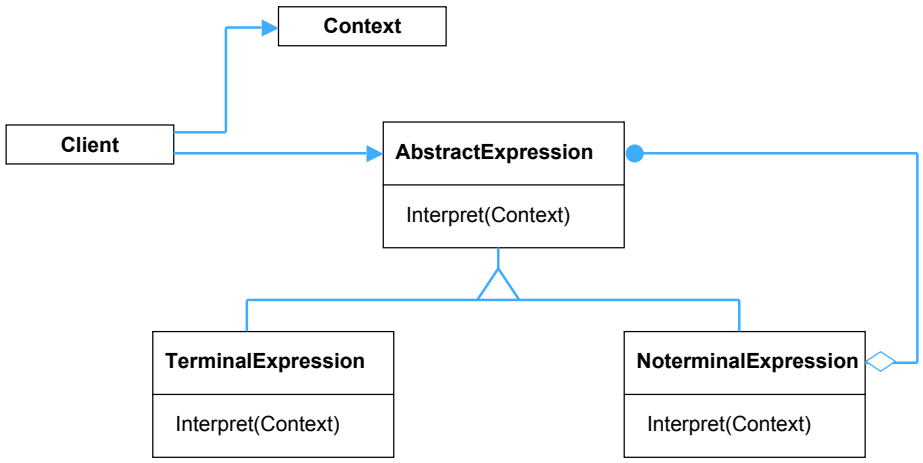



# INTERPRETER
# Motivation




```
Raining & (dogs | cats) *
```

# INTERPRETER
# Structure






# INTERPRETER
# Participants

- AbstractExpression (RegularExpression)
  - Declares abstract Interpret operation
- TerminalExpression (LiteralExpression)
  - Implements *Interpret* operation according to symbol
- NonterminalExpression (AlternationExpression, …)
  - One class for each rule in the grammar
  - Holds instances of *AbstractExpression* for each symbol in it
  - Implements *Interpret* operation, mostly through recursion on the *AbstractExpression* instances
- Context
  - Global interpreter information, manipulated by *Interpret* implementations
  - Initialized by *Client*
- Client
  - Builds representation of sentence through *NonterminalExpression* and *TerminalExpression* classes
  - Invokes *Interpret* operation of root symbol

# INTERPRETER
# Applicability

Use the Interpreter pattern when:

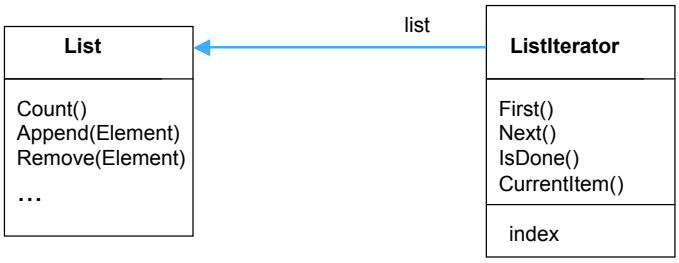
- You have abstract syntax trees
  - there is a language to interpret
  - statements are representable as AST
- The grammar is simple
  - Large and unmanageable class hierarchy in complex cases (use parser generators)
- Efficiency is not a critical concern
  - most efficient interpreters first translating parse trees into another form
  - Example: Regular expressions → state machines
  - Translator itself could be an interpreter



# ITERATOR
# (Object Behavioral)

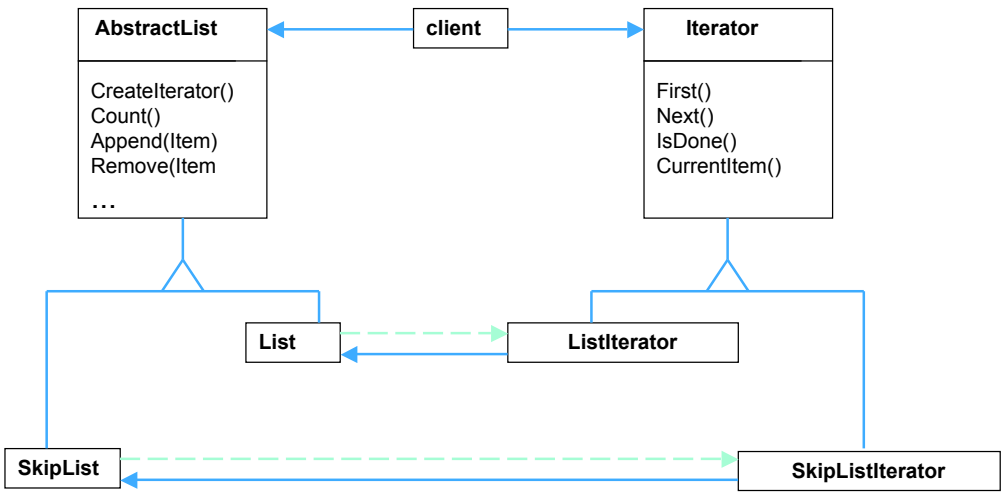
- Intent:
  - Access elements of an aggregate object sequentially
  - Don't expose underlying representation
- Motivation:
  - Traversal of aggregate list object
  - Allow multiple pending traversals
  - Separate traversal operations from list interface
- Solution:
  - Take responsibility for access and traversal out of the list interface
  - Iterator class for list element access
    - Current element is managed in the Iterator implementation
  - Decouple aggregate class from client

# ITERATOR
# Motivation



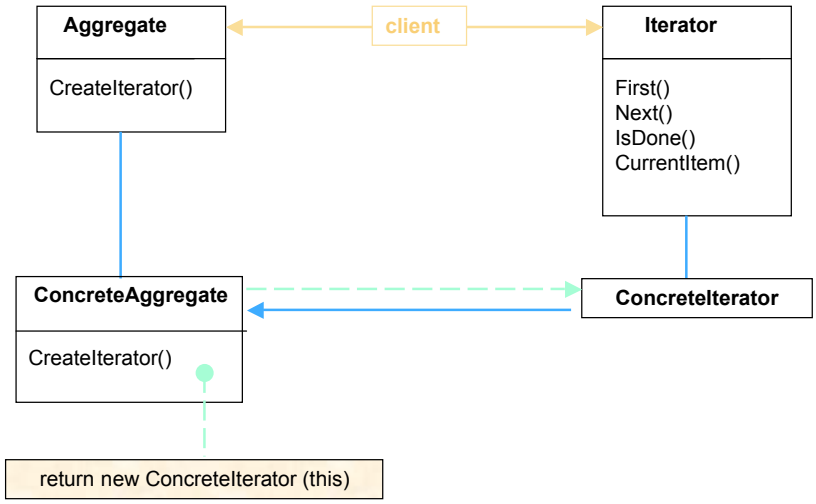



# ITERATOR
# Polymorphic Iteration




Iteration mechanism independent of concrete aggregate class

# ITERATOR
# Structure






# ITERATOR
# Applicability / Benefits

Use the *Iterator* pattern to access aggregate content

- No exposing of internal representation
- Support for multiple traversals
- Uniform traversal interface for different aggregates

Benefits:

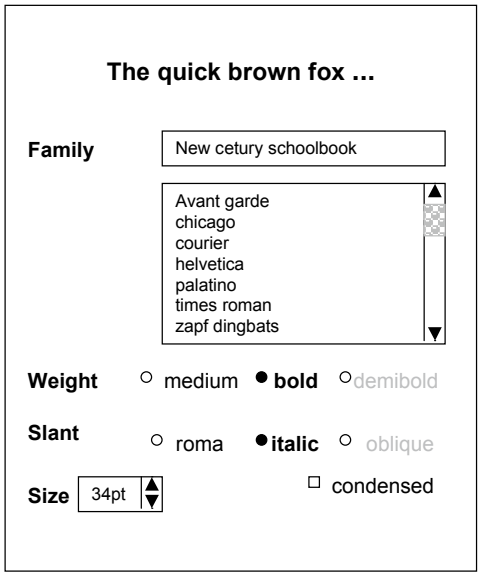
- Support for variation in the traversal of an aggregate
  - e.g. parse order
  - New traversals through *Iterator* sublasses
- Simplification of *Aggregate* interface
- Each iterator keeps track of it's own traversal state

# MEDIATOR
# (Object Behavioral)

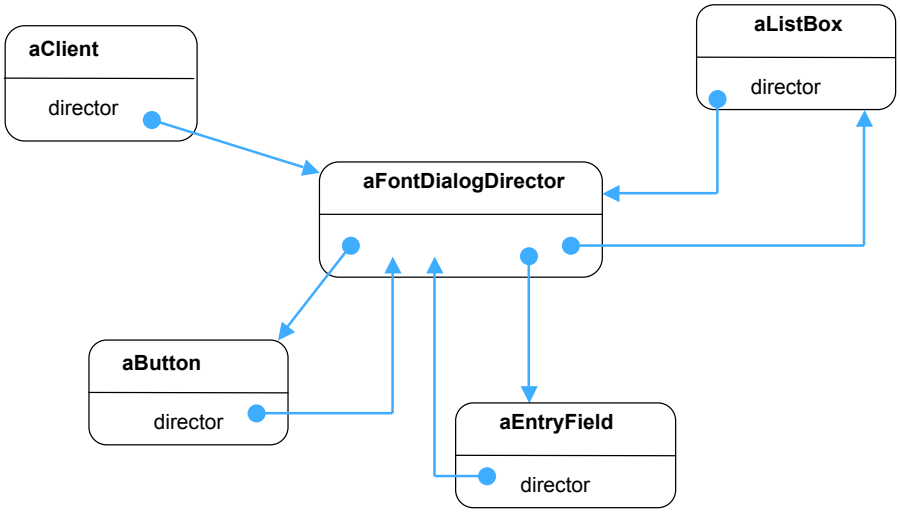
- Intent:
  - Define object which encapsulates interaction of objects
  - Keep objects from referring to each other, allow variation of interaction
- Motivation:
  - OO-design might lead to structure with many connections
  - Example: Implementation of dialog box
    - Window with widgets
    - Most widgets depend on each other
    - New dialogs with same widgets have different behavior
  - Define control and coordination intermediary → director
    - Hub of communication for widgets
    - Every widget only need to know the director object



# MEDIATOR
# Motivation

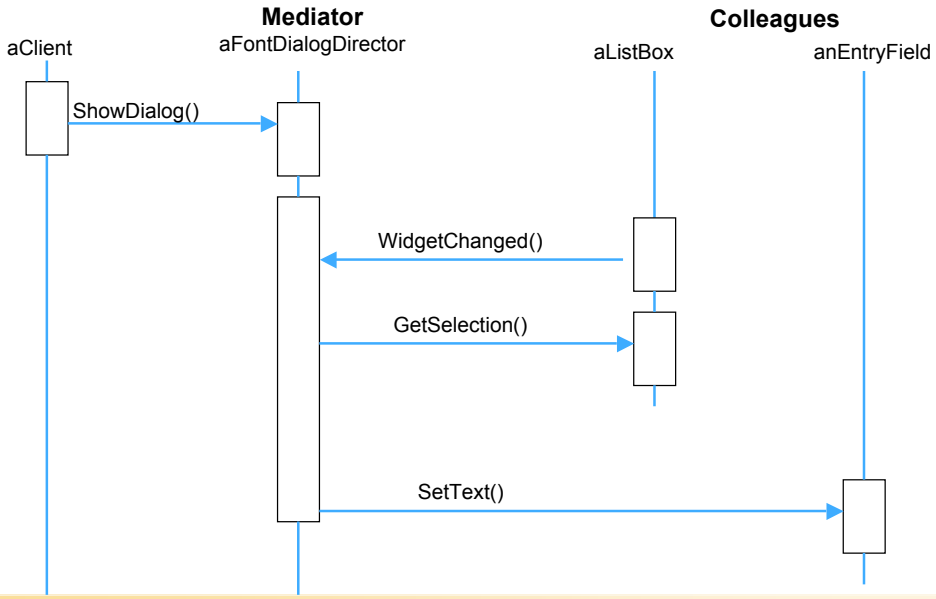


# MEDIATOR
# Motivation

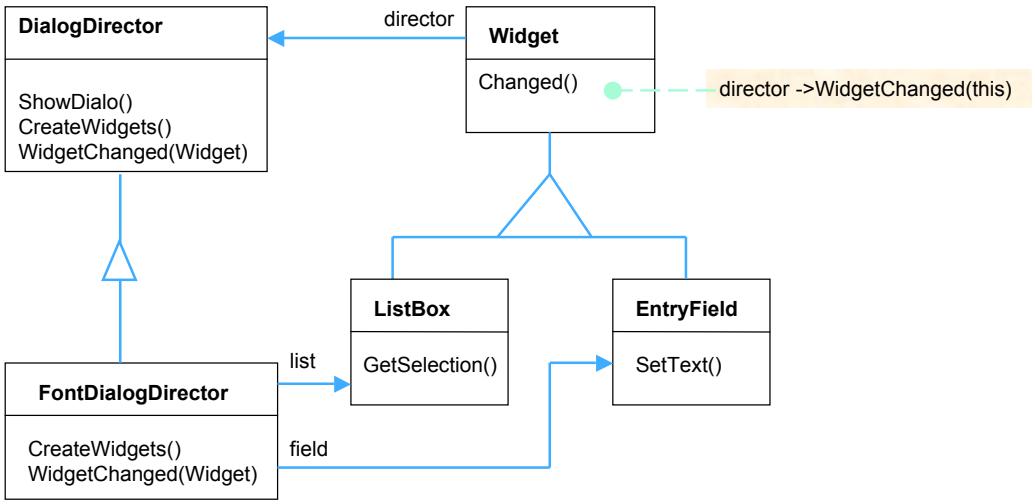





# MEDIATOR
# Motivation

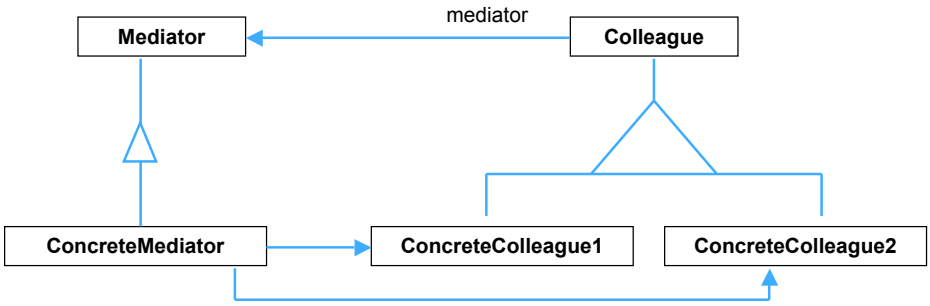


# MEDIATOR
# Motivation

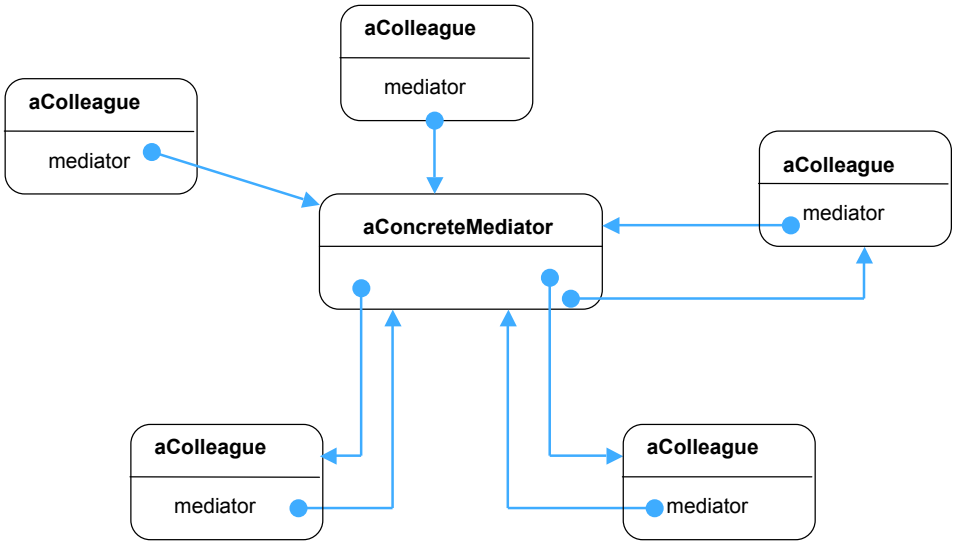





# MEDIATOR
# Structure



# MEDIATOR
# Typical Object Structure






# MEDIATOR
# Applicability / Benefits

Use the Mediator pattern when:

- Multiple objects …
  - … communicate in a complex way
  - … have unstructured / difficult dependencies
  - … prevent reuse of single objects through the tight interdependencies
  - … should be easily configurable with another behavior

Benefits:

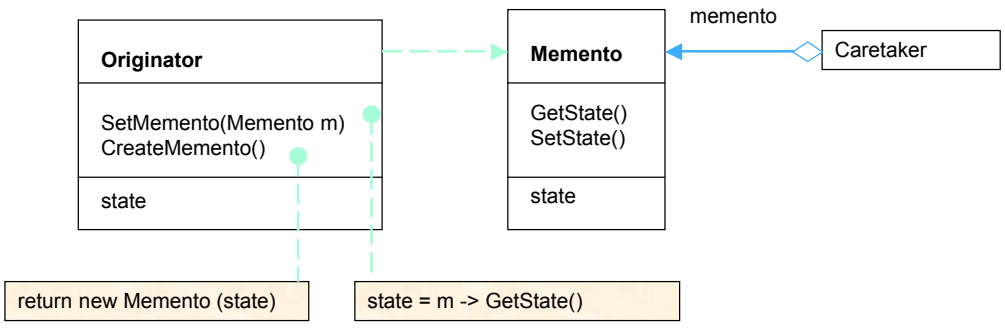
- Limits subclassing in case of behavior change
- Decouples colleague objects
- Simplifies object protocols (one-to-many vs. many-to-many)
- Abstraction of object cooperation
- Provides centralized control

# MEMENTO
# (Object Behavioral)

- Intent:
  - Capture and externalize an object's internal state
  - Provide capability to restore object later
  - Keep encapsulation principle
- Motivation:
  - Save state information for later restore
    - Checkpointing mechanisms
    - Undo mechanisms
  - *Memento* object
    - Storage for state snapshot of another (originator) object
    - Read / written on request by originator object
    - Opaque to other objects



# MEMENTO
# Structure




- *Caretaker* requests *Memento* from *Originator*
  - holds it for a time
  - passes it (eventually) back to the originator
- Only the *Originator* of a *Memento* can assign / retrieve its state

# MEMENTO
# Participants

- Memento
  - Stores internal state of the *Originator* object
  - Protects against access by objects other than the *Originator*
    - Narrow interface for *Caretaker*
    - Wide interface for *Originator*
    - In best case, only one *Originator* has access to the state data
- Originator
  - Creates *Memento* containing current state snapshot
  - Uses *Memento* to restore internal state
- Caretaker
  - Responsible for *Memento's* safekeeping
  - Never operates / examines content of a *Memento*



# MEMENTO
# Applicability

Use the Memento pattern when:

- Snapshot of object state is needed
  - Later restore
- Direct interface would break encapsulation
  - Exposing of implementation details

Benefits:

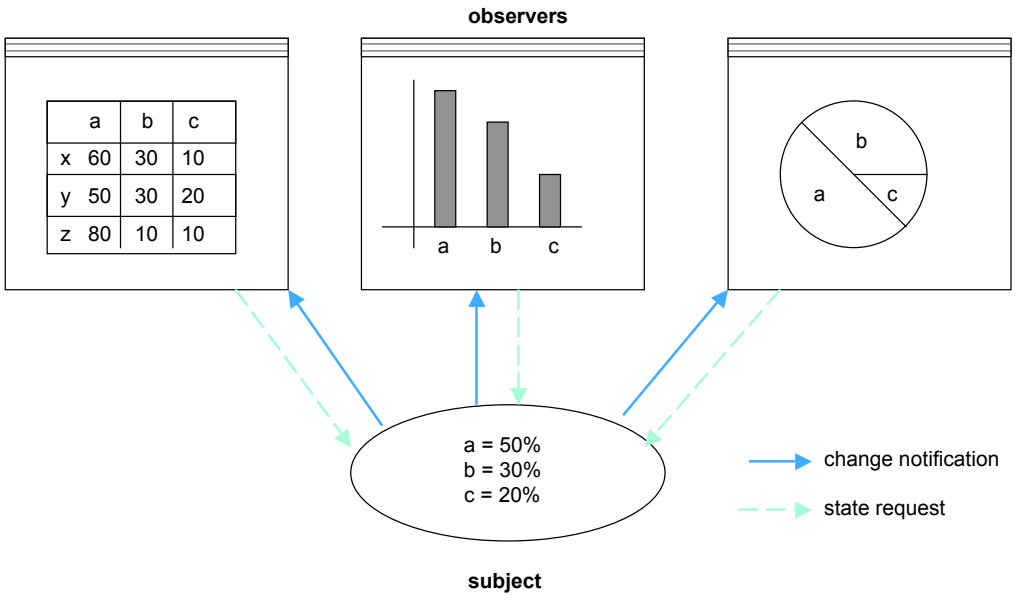
- Preserves encapsulation boundaries
  - Shields other objects from potentially complex *Originator* internals
- Simplifies *Originator*
  - Storage management handled externally

# OBSERVER
# (Object Behavioral)

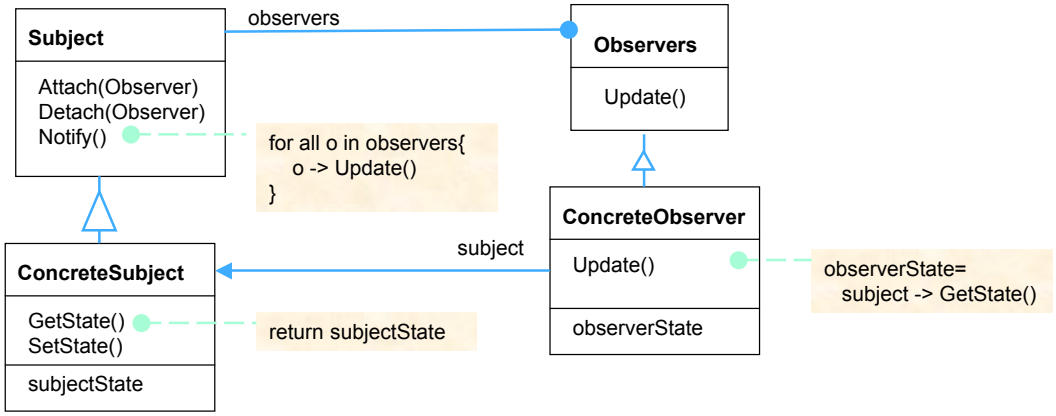
- Intent:
  - Define one-to-many dependency between objects
  - Notification of dependent objects about state change
- Motivation:
  - Need to maintain consistency between related objects
  - Example: GUI toolkit
    - Separate presentation aspects from application data
    - Different visualization of same data
    - No dependency between visualization objects, but all update on data change
  - *Subject* and it's dependent *Observers*
  - **Publish-subscribe interaction**



# OBSERVER
# Motivation



# OBSERVER
# Structure






# OBSERVER
# Participants

- Subject:
  - Knows its *Observers*
  - Provides interface for attaching/detaching *Observer* objects
- Observer:
  - Defines an *Update* interface, for changes in the subject
- ConcreteSubject:
  - Stores state, which is of interest to *ConcreteObserver* objects
  - Sends a notification to its observers when its state changes
- ConcreteObserver:
  - Maintains a reference to a *ConcreteSubject* object
  - Own state, consistent with *Subject* state through *Update* interface

# OBSERVER
# Collaborations

- *ConcreteSubject* notifies its observers whenever the observer's state becomes invalid
- *ConcreteObserver* object may query the *Subject* for information in case of notification
- Notification might be triggered by another *Observer*



# OBSERVER
# Applicability

Use the Observer pattern when:

- Abstraction has two dependent aspects
- Dependencies on object state change are unclear
- Need notification without knowledge about *Observers*

Benefits:

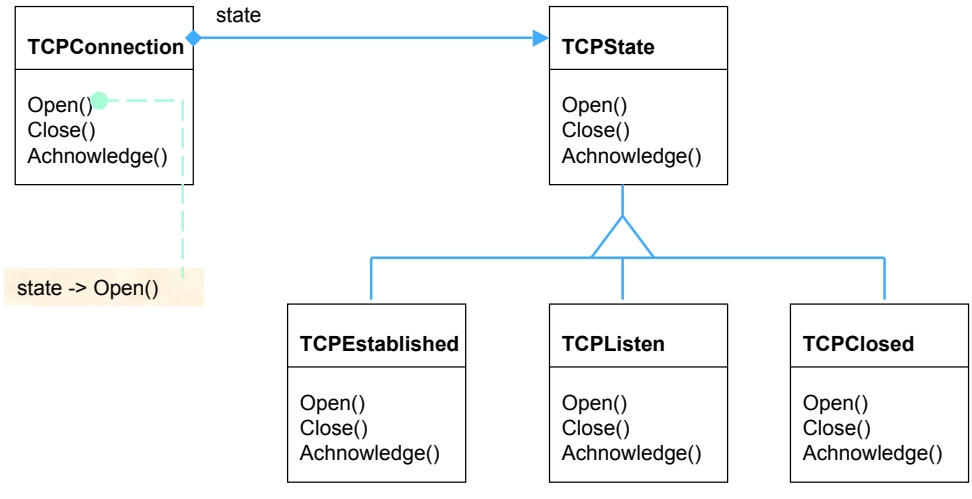
- Abstract coupling between *Subject* and *Observers*
- Support for broadcast communication

# STATE
# (Object Behavioral)

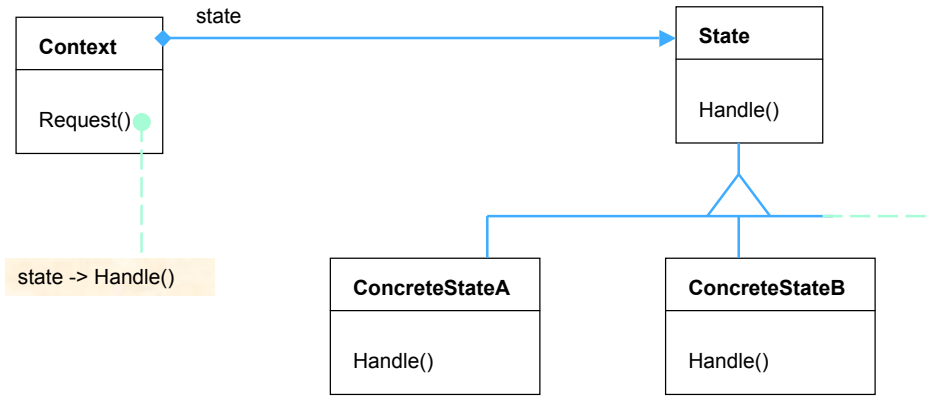
- Intent:
  - Allow object to alter its behavior, depending on internal state change
  - Object appears to change its class
- Motivation:
  - Example: Class TCPConnection
  - Represent possible states of network connection as objects
  - Abstract base class TCPState
    - Subclasses implement state-specific behavior
  - TCPConnection maintains a state object
  - State-specific requests are handled directly by the according state object



# STATE
# Motivation



# STATE
# Structure






# STATE
# Collaborations

- *Context* delegates state-specific requests to current *ConcreteState* object
- *Context* may pass itself as an argument to the *State* object
- *Context* is the primary interface for clients
  - *Clients* can configure a *Context* with *State* objects
  - Once configured, *Clients* don´t have to deal any longer directly with *State* objects

# STATE
# Applicability

Use the State pattern when:

- Object's behavior depends on its state
- Object must change behavior at runtime, reasoned by state information

Benefits:

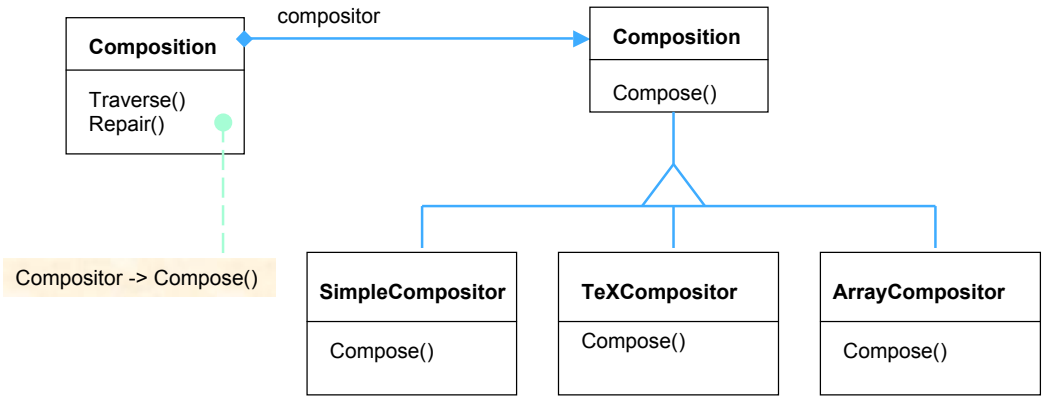
- Localize state-specific behavior
- Makes state transitions explicit
- State objects can be shared



# STRATEGY
# (Object Behavioral)

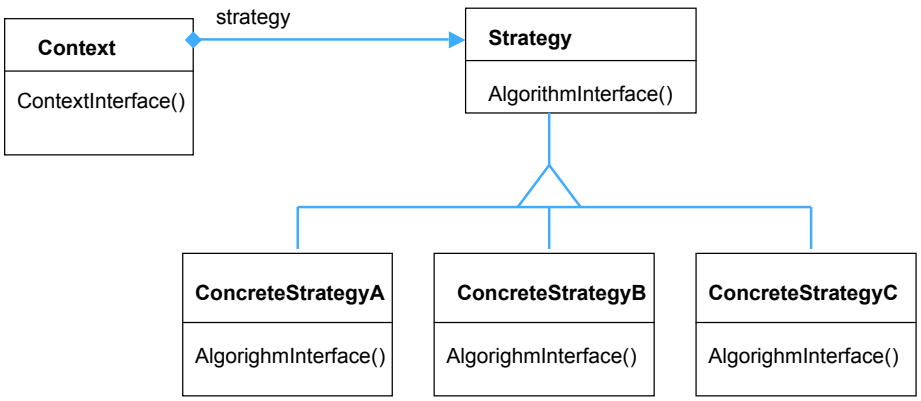
- Intent:
  - Define family of algorithms
  - Encapsulate each one, make them interchangeable
  - Vary algorithm independent from clients
- Motivation:
  - Example: Break composed text stream into lines
    - Simple strategy (AsciiParser)
    - Paragraph optimization (TexParser)
    - Array composition (fixed number of columns)
  - Don't integrate different algorithms directly in client
- Solution:
  - Definition of encapsulating classes, algorithm is a *Strategy*

# STRATEGY
# Motivation






# STRATEGY
# Structure



# STRATEGY
# Applicability

Use the Strategy pattern when

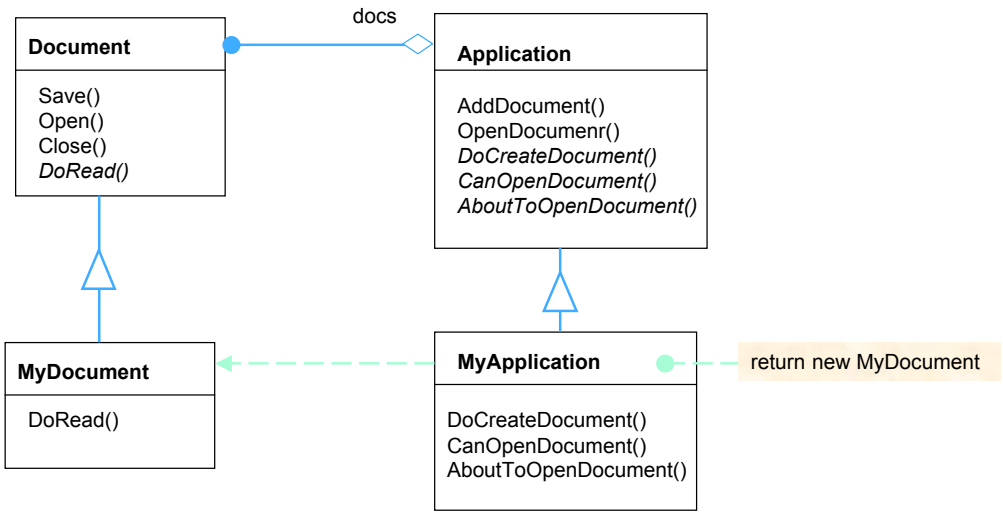
- Related classes differ only in their behaviour
- You need different variants of an algorithm
  - Should be implementable as hierarchy
- To avoid exposing algorithm-specific data structures
- To replace conditional statements for behaviour
  - Move conditional branches to according Strategy classes



# TEMPLATE METHOD
# (Class Behavioral)

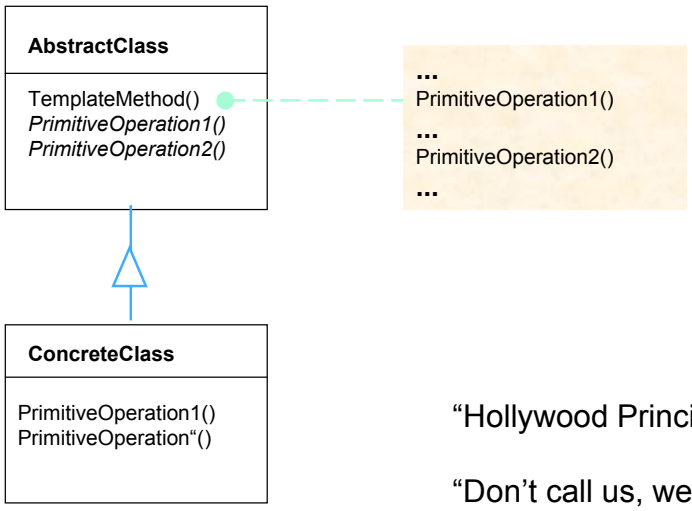
- Intent:
  - Define algorithm skeleton in an operation
  - Subclass might redefine steps of the algorithm
- Motivation:
  - Example: Framework with Application / Document classes
  - Applications can subclass for specific needs
  - Common algorithm for opening a document
    - Check of the document can be opened ← app.-specific
    - Create app-specific Document object ← app.-specific
    - Add new Document to the set of documents
    - Read document data from a file ← doc.-specific
- Solution:
  - Define some of the algorithm steps using abstract operations

# TEMPLATE METHOD
# Motivation






# TEMPLATE METHOD
# Structure




"Hollywood Principle":

"Don't call us, we'll call you"

# TEMPLATE METHOD
# Applicability

The Template Method pattern should be used

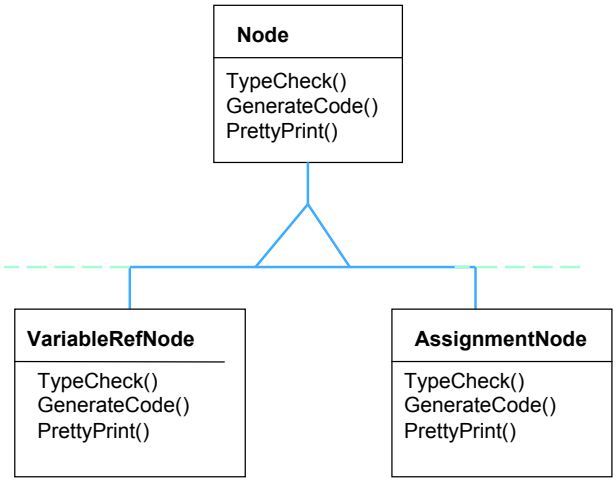
- To implement the invariant parts of an algorithm once
- To leave it up to subclasses to implement varying behaviour
- To avoid code duplication
  - Centralize common behaviour of subclasses
  - First identify the differences in the existing code
  - Then separate the differences into new operation
  - Replace the differing code with a template method
- To control subclasses extensions
  - Template method that calls “hook” operations
  - permitting extensions only at those points.



# VISITOR
# (Object Behavioral)

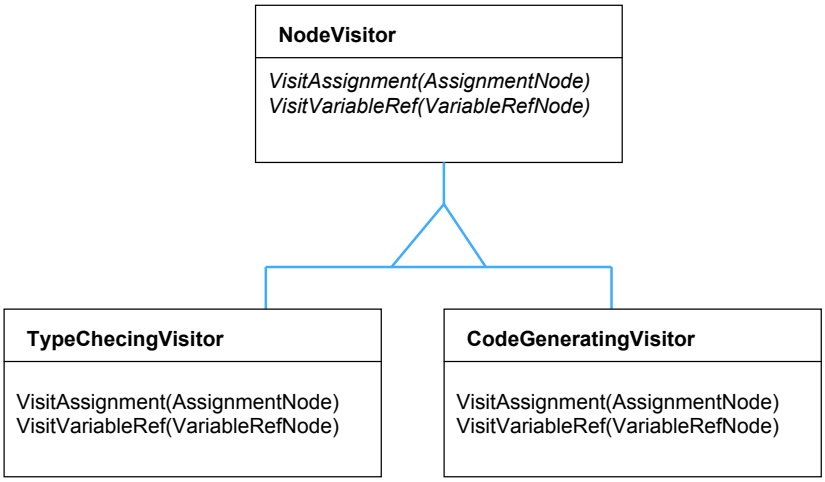
- Intent:
  - Represent operation for object structure elements
  - Define new operations without changing the element classes
- Motivation:
  - Example: Compiler with internal AST code representation
    - Operations on AST (type checking, code optimization, …)
    - Different AST node types (assignment node, variable node, …)
  - Providing operations on each node type is hard
    - New operations
- Solution:
  - Package related operations in a separate object → *Visitor*
  - Pass visitor to element objects
  - Node classes become independent of operations applied to them

# VISITOR
# Motivation

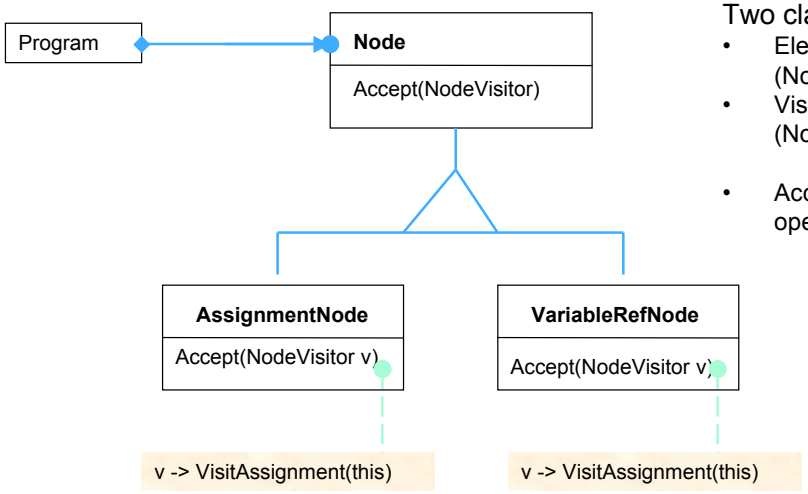





# VISITOR
# Motivation



# VISITOR
# Motivation

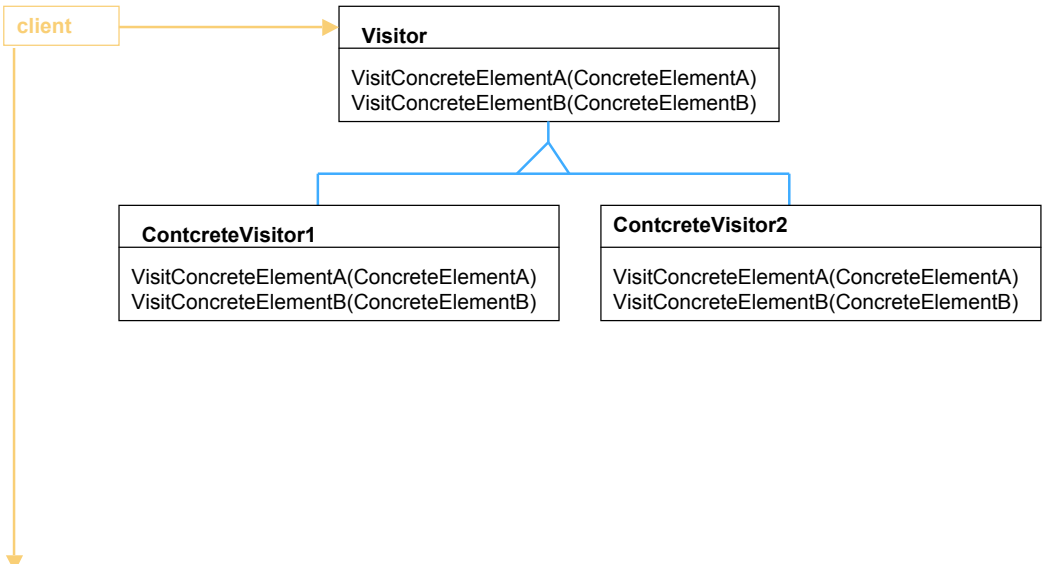



Two class hierarchies:

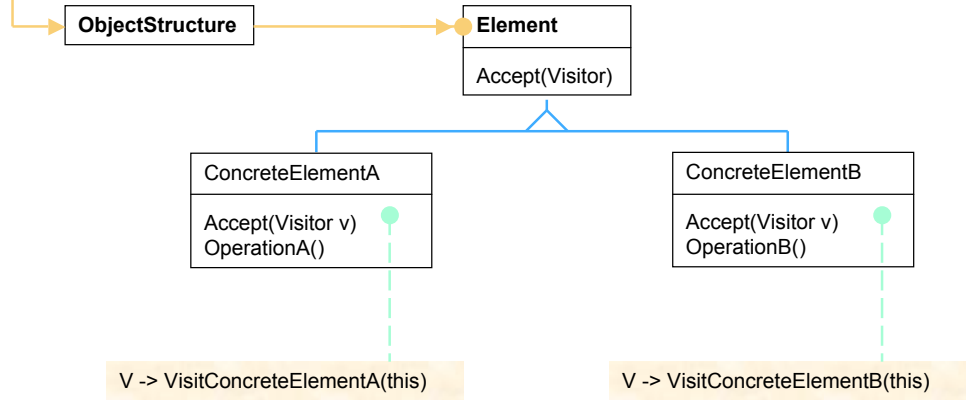
- Elements being operated on (Node hierarchy)
- Visitors with node operations (NodeVisitor hierarchy)
- Acceptance of visitor through operation call



# VISITOR
# Structure (I)



# VISITOR
# Structure (II)






# VISITOR
# Collaborations

- Client must create a *ConcreteVisitor* object
- Client traverses the object structure, visiting each element
- Visited *Element* calls class corresponding *Visitor* operation
  - Element supplies itself as an argument to operation
  - Visitor can access state information

# VISITOR
# Applicability

Use the Visitor pattern when

- Need to perform operations on differing objects, depending on concrete classes
- Many distinct and unrelated operations need to be performed on objects
  - Avoid 'pollution' of object interface
  - Visitor pattern keeps related operations together
  - Only needed application operations with shared object structures
- Rarely change of element object structure, frequent change of operation set
  - Changes of object structure classes might require costly visitor changes



# Design Pattern Space

Defer object creation to another class

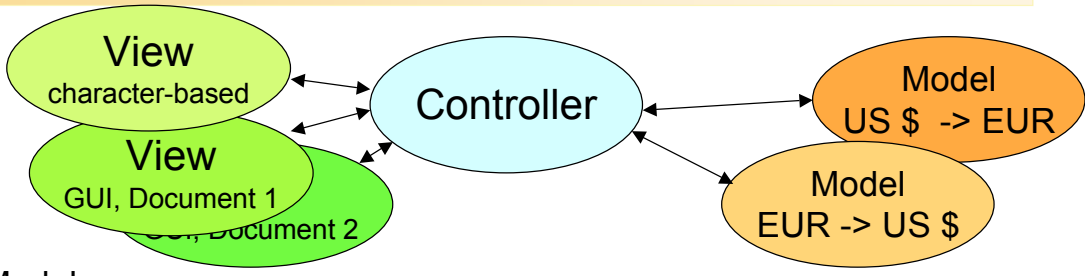
| | | Purpose | | |
|---|---|---|---|---|
| | | Creational | Structural | Behavioral |
| Scope | Class | Factory Method | Adapter (class) | Interpreter<br>Template Method |
| | Object | Abstract Factory<br>Builder<br>Prototype<br>Singleton | Adapter (object)<br>Bridge<br>Composite<br>Decorator<br>Facade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

Defer object creation to another object

Describe ways to assemble objects

Describe algorithms and flow control

# Design Patterns in Smalltalk MVC




- Model
  - Implements algorithms (business logic)
  - Independent of environment
- View:
  - Communicates with environment
  - Implements I/O interface for model
- Controller:
  - Controls data exchange (notification protocol) between model and view



# Model/View/Controller (contd.)

- MVC decouples views from models – more general:
  - Decoupling objects so that changes to one can affect any number of others
  - without requiring the object to know details of the others
  - **Observer pattern** solves the more general problem
- MVC allows view to be nested:
  - CompositeView objects act just as View objects
  - **Composite pattern** describes the more general problem of grouping primitive and composite objects into new objects with identical interfaces
- MVC controls appearance of view by controller:
  - Example of the more general **Strategy pattern**
- MVC uses **Factory** and **Decorator patterns** as well