Object Oriented Software Design I

Behavioural Patterns

Giuseppe Lipari
http://retis.sssup.it/~lipari

Scuola Superiore Sant'Anna – Pisa

November 20, 2011

Outline

What are Behavioural Patterns

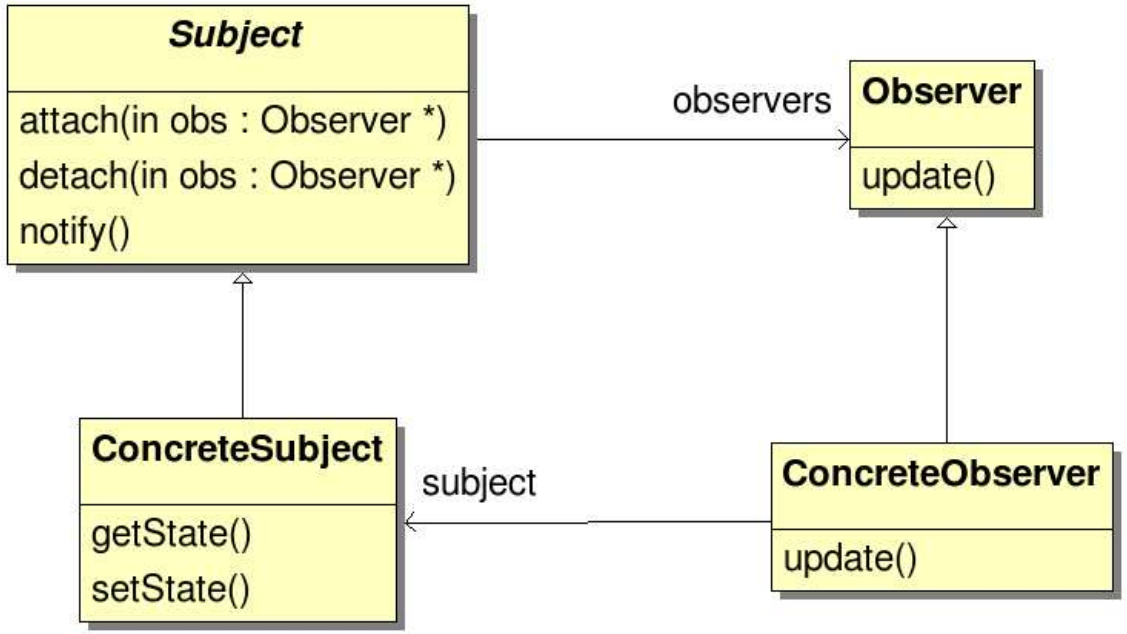
- Behavioural patterns describe patterns of message communications between objects
- Therefore, they are concerned with *algorithms*, rather than with *structures*
  - Of course, behavioural patterns are strictly related to structural patterns
- Key observation: how the objects know about each other?
- Main goal: *decouple* objects from each other to allow a dynamic and flexible configurations of algorithms and methods

Outline

Motivation

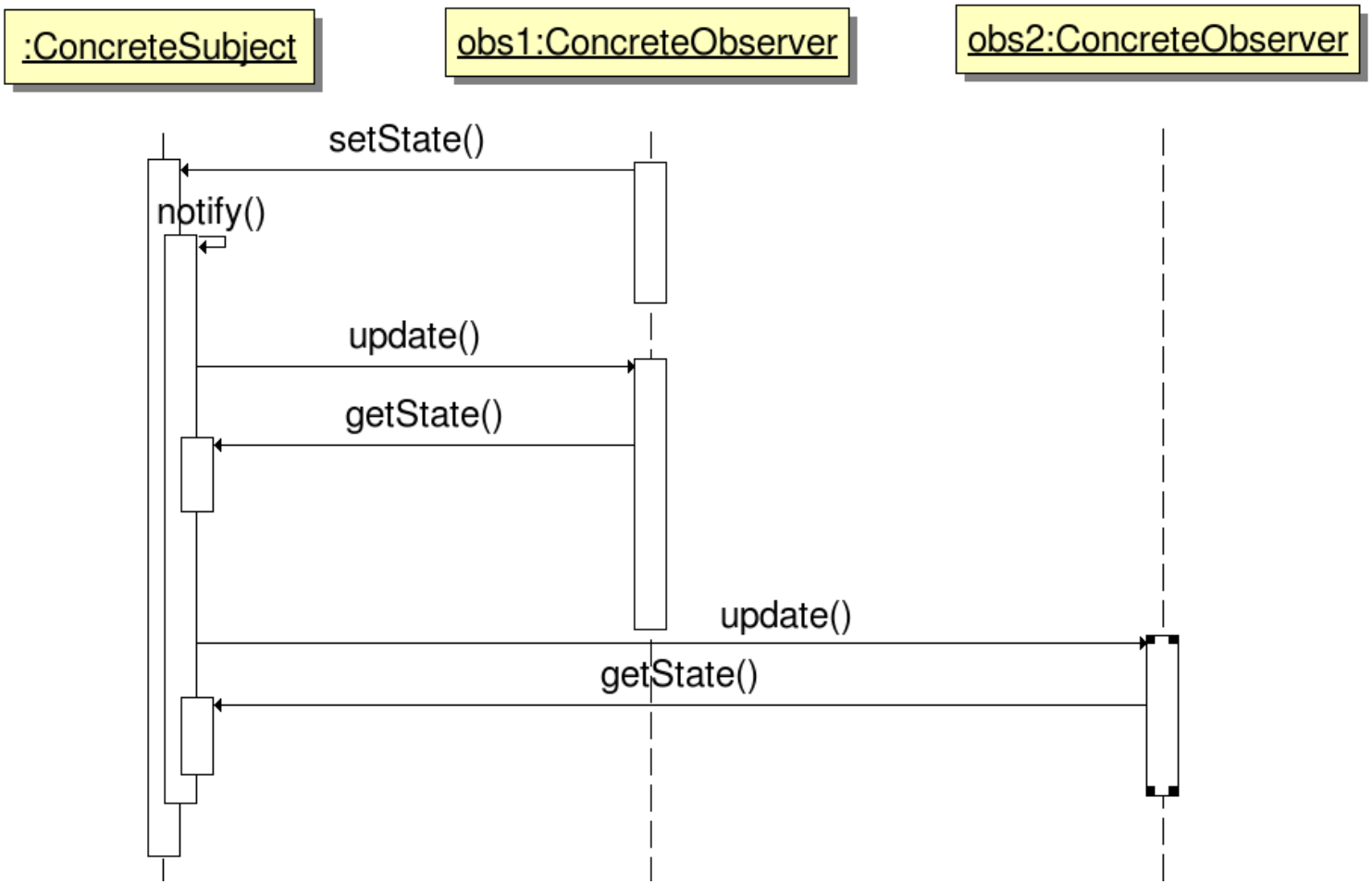
- We need to maintain consistency among (weakly-)related object
  - When something happens to an object, other objects must be informed
- Typical example in GUIs
  - The Document object must be informed when a button “Print” is clicked, so that the print() operation can be invoked
  - The ViewPort object must be informed when the window is resized(), so that it can adjust the visualisation of the objects
  - We have already presented an example when presenting the Adapter pattern: an object can “listen” to other objects changes
- Participants:
  - An object changes its state (subject)
  - Another object wants to be informed (observer)

UML Diagram



- Subject is the interface for something to to be observed
- Observer is thing that observes

Message sequence chart



Example

- The user resizes a window:
  - every component of the window needs to be informed of a resize operation (viewport, scrollbars, toolbars, etc.)
  - in this way, every object can synchronize its state with the new window size
- Solution:
  - The window can install *observers*
  - All components (viewport, scrollbar, etc.) can attach an observer to the main window that is informed when a resize operation is under way
  - The observer asks for the current size of the window, and invoke methods on the objects to adjust their state (size)

Consequences

- **Abstract coupling** between subject and observer
  - all that a subject knows is that there is a list of observers, but it does not know anything about the observers themselves
  - the observer instead must know the subjects
- **Broadcast communication**
  - There can be many independent observers, with different purposes and hierarchies
  - *Example*: resizing a window can affect the viewports inside the window, the scrollbars, etc.

Consequences

- **Abstract coupling** between subject and observer
  - all that a subject knows is that there is a list of observers, but it does not know anything about the observers themselves
  - the observer instead must know the subjects
- **Broadcast communication**
  - There can be many independent observers, with different purposes and hierarchies
  - *Example*: resizing a window can affect the viewports inside the window, the scrollbars, etc.
- **Unexpected updates**
  - A seemingly innocuous operation on the subject may cause a cascade of updates on the observers and their dependent objects, many of them may not care about any update
  - This simple protocol does not tell the observer *what* change happened to the subject (a resize? a move?)

Update problems

- **Pull model**
  - the subject sends nothing
  - the observer asks for details about the changes
  - equivalent to what we have already seen
- **Push model**
  - the subject sends the observer detailed information about the change (whether it wants it or not)
  - the observer can understand if he is interested in the change by analysing this additional parameter
- **Specifying events**
  - By complicating the protocol, it is possible to register to specific aspects
    - onResize()
    - onMove(),
    - ...
  - more efficient, but more complex interface

Outline

Motivation

- Consider a context-sensitive “help” for a GUI
  - the user can click on any part of the interface and obtain help on it
- The help that is actually provided depends on
  - which part of the interface (button, menu, etc.)
  - the context (where the button is)
- Example:
  - a button in a dialog box
  - a button in the main window
- If no help can be found for that part, a more general help page is shown

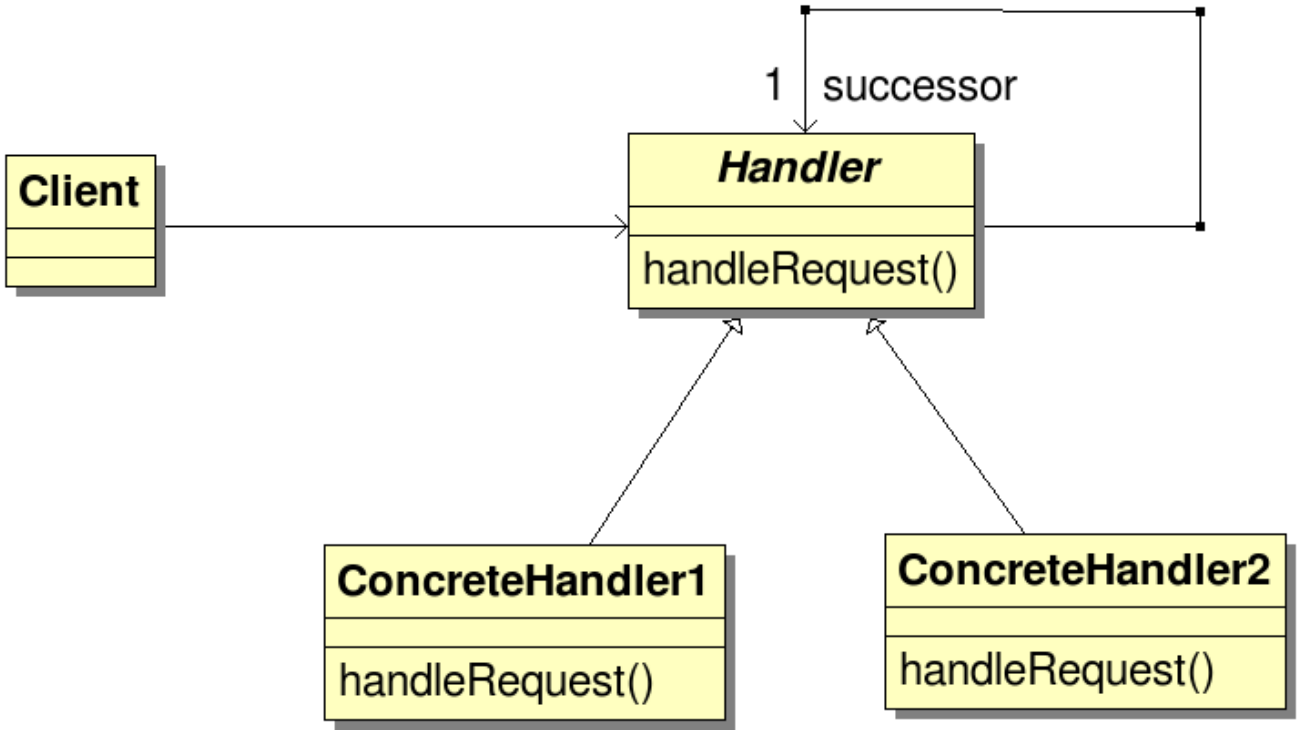
Motivation

- Consider a context-sensitive “help” for a GUI
  - the user can click on any part of the interface and obtain help on it
- The help that is actually provided depends on
  - which part of the interface (button, menu, etc.)
  - the context (where the button is)
- Example:
  - a button in a dialog box
  - a button in the main window
- If no help can be found for that part, a more general help page is shown
- The help should be organised hierarchically
  - From more general to more specific
- The object that provides the help is not known to the object that initiates the request for help
  - the button does not know which help object will handle the request, as this depends on the context

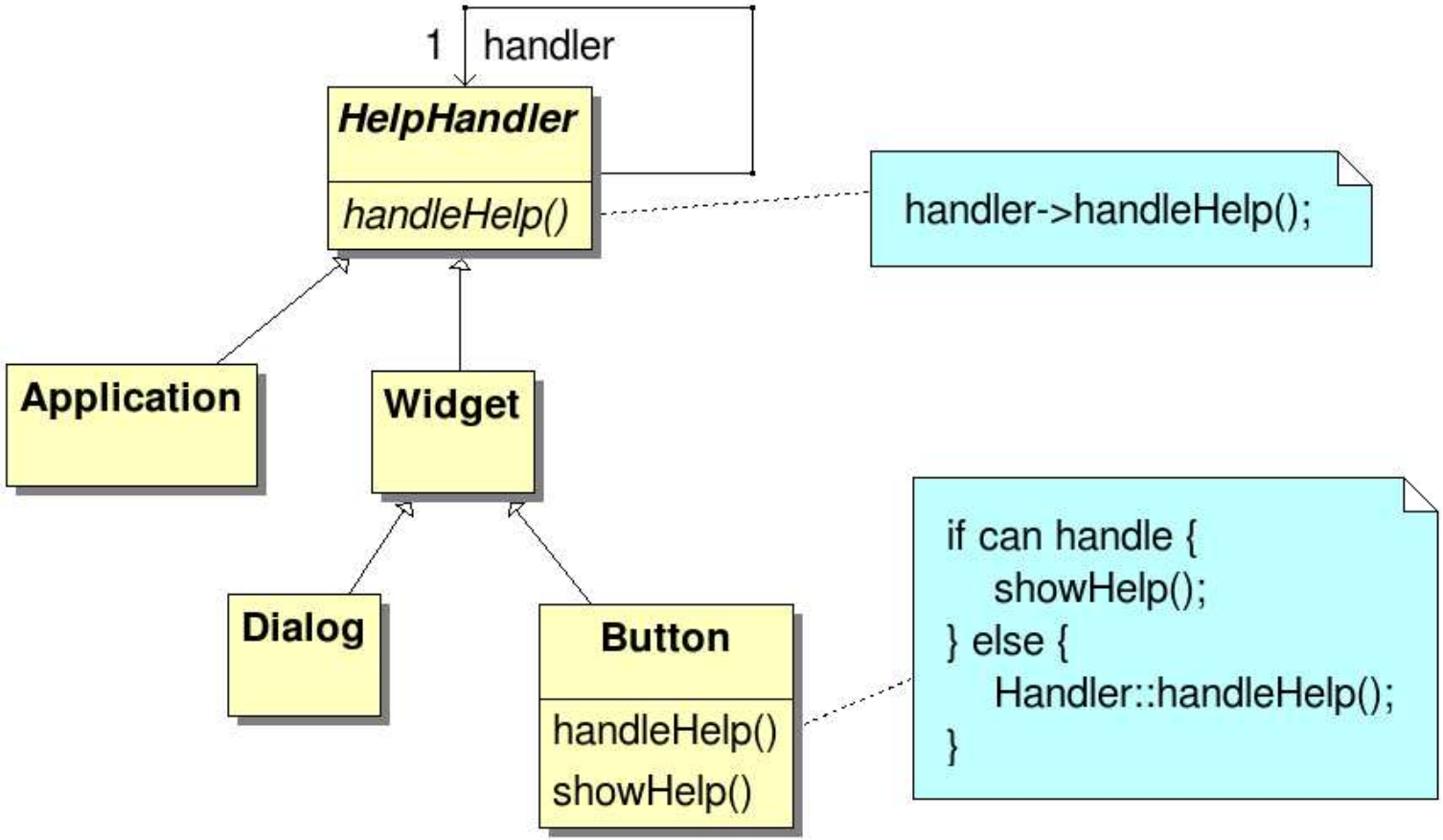
Goals and requirements

- Decouple senders and receivers
- Gives multiple objects a chance to handle the request
- Chain:
  - build a list of receivers
  - pass the request to the first receiver
  - if the request cannot be handled, pass it to the next receiver in the chain
- Consequences
  - *Reduced coupling*: the sender does not care which object handles the request
  - *Added flexibility in assigning responsibility*: different responsibility can be distributed to different objects
  - *Receipt is not guaranteed*: there is not guarantee that eventually some object will handle the request

UML diagram


1
successor
Client
Handler
handleRequest()
ConcreteHandler1
handleRequest()
ConcreteHandler2
handleRequest()

Example Instance



Notes

- Applicability
  - More than one object can handle a request, and the handler is not known a priori
  - 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
- Implementation
  - *Connecting successors*: the `Handler` class itself usually maintains a link to the successor. Also, it automatically forwards allow requests by default if there is a successor.
  - *Representing requests*: usually represented in the method call itself (i.e. `handleHelp()`). However, we could think of one or more parameters to encode the specific request.
  - to simplify the passage of parameters, we could also encode them into an object that is passed along the chain

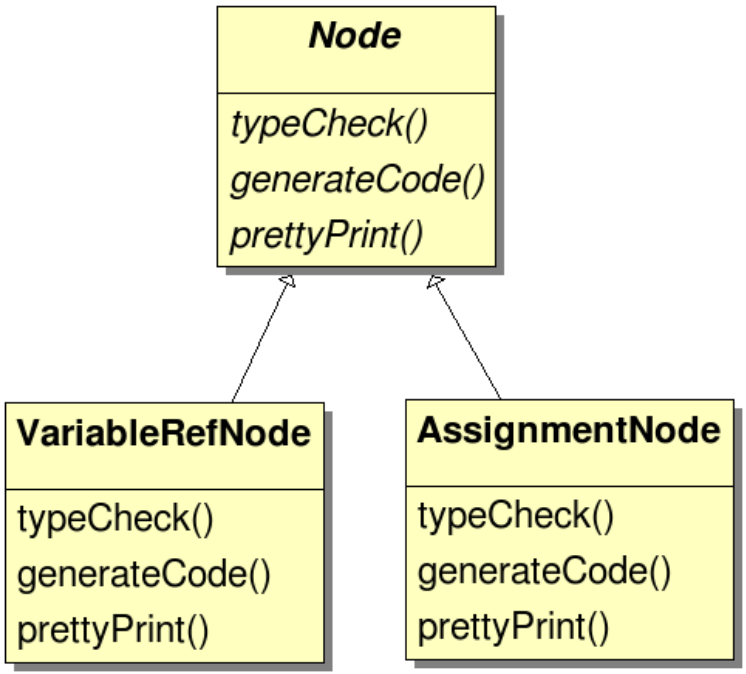
Outline

Motivation

- Consider a compiler that internally represents a program as an abstract syntax tree
  - the compiler will take as input a text file containing the program
  - the *parser* component will read the file and build the syntax tree
  - then it performs syntax checking on the tree
    - for example, it checks that all used variables have actually been defined, and that the type corresponds
  - it will also need to generate code
  - optionally, it can need to print the program in a nice formatted way
- In general, on a complex structure, you may need to define several distinct operations
- The structure may consists of different types of nodes (see the `Composite` pattern)

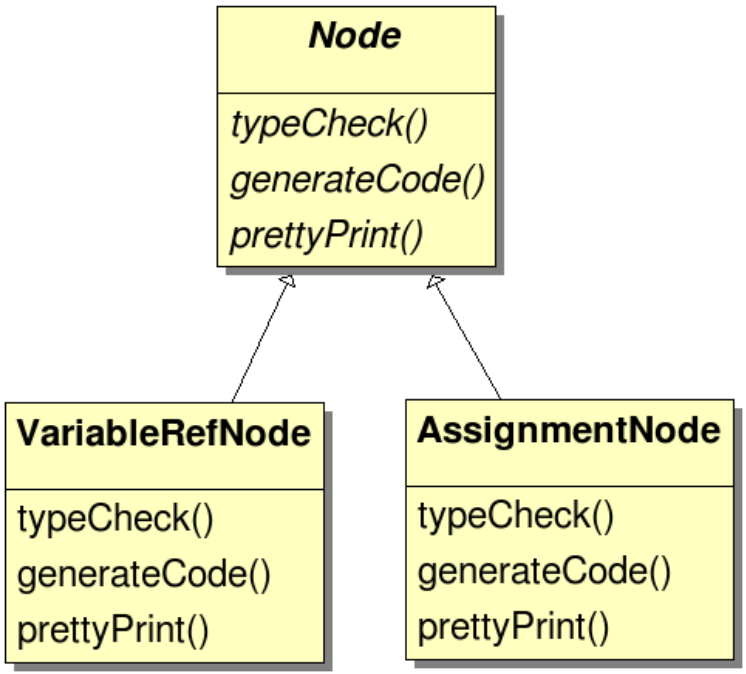
Naive approach

- Let's define a method for each operation in the node itself



Naive approach

- Let's define a method for each operation in the node itself

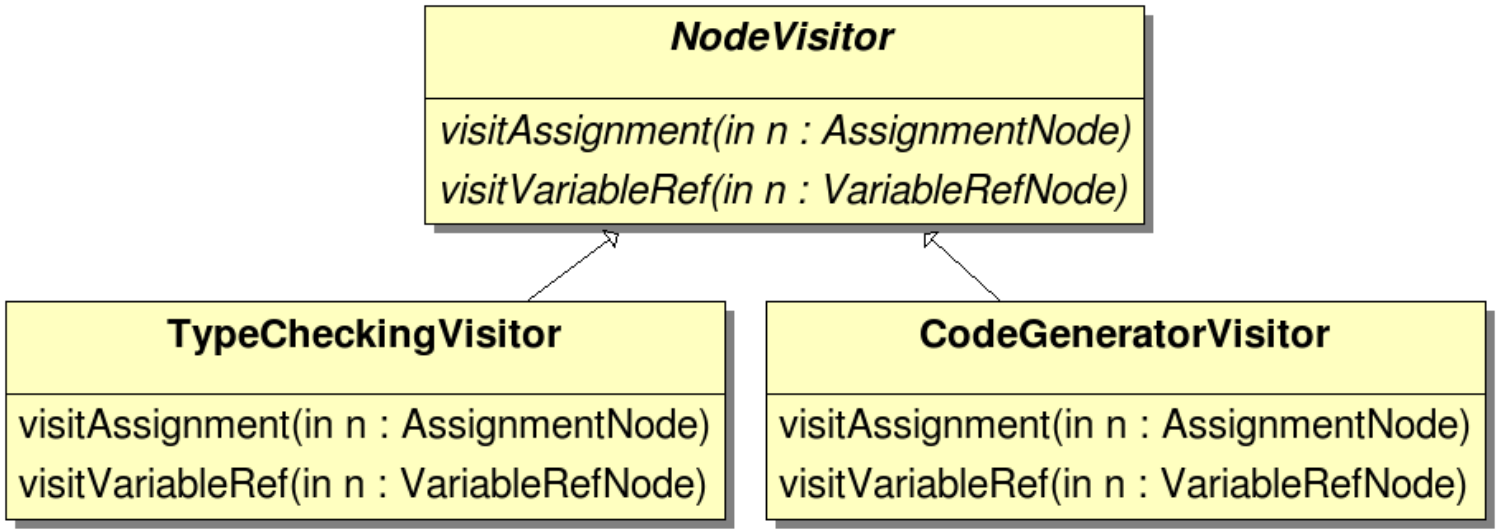


- Not correct. Why?

The problem

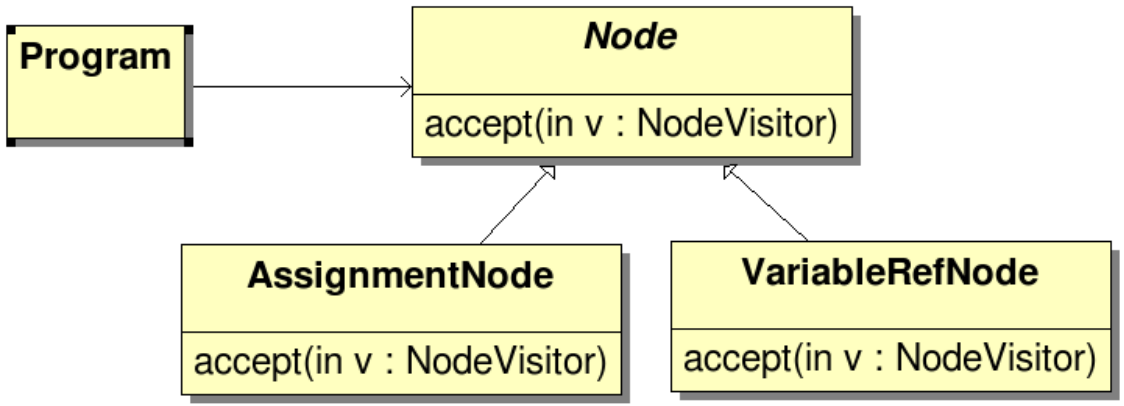
- Does not scale: what if we want to implement one more operation to visit the node?
  - We need to change all the `Node` classes
- Also, we are doing many things to do in a single class
  - The `Node` class should care about the structure, and to provide a generic interface to all types of nodes
  - `Node` typically implements a **Composite** pattern
- What we need to do
  - Decouple visiting from Nodes.
  - Solution: use a different class to encapsulate the various visiting operations

The Visitors



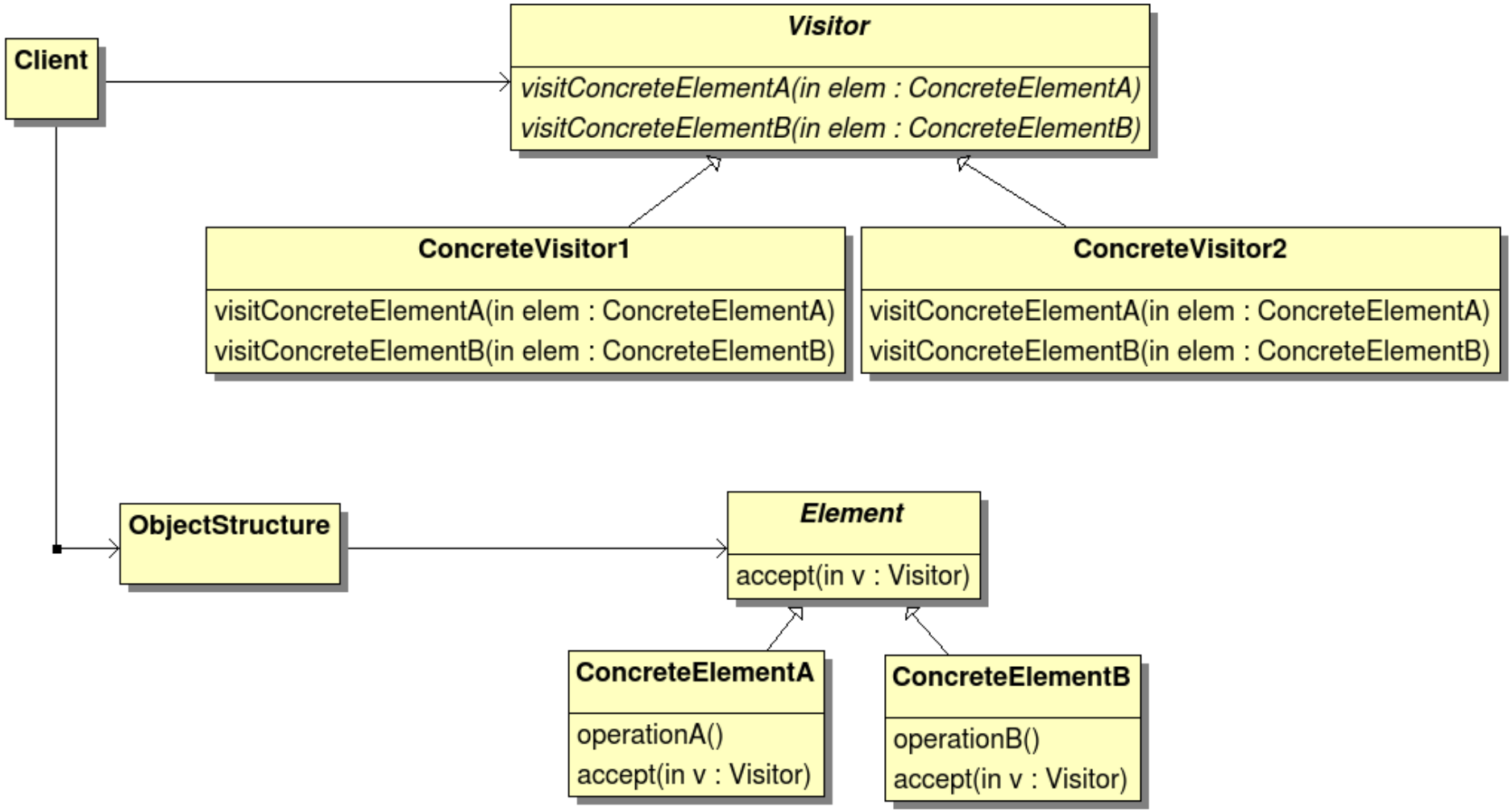
- These classes take care of visiting the Nodes, and doing the appropriate operations
- each concrete visitor implements a different kind of visit

The Nodes to be visited

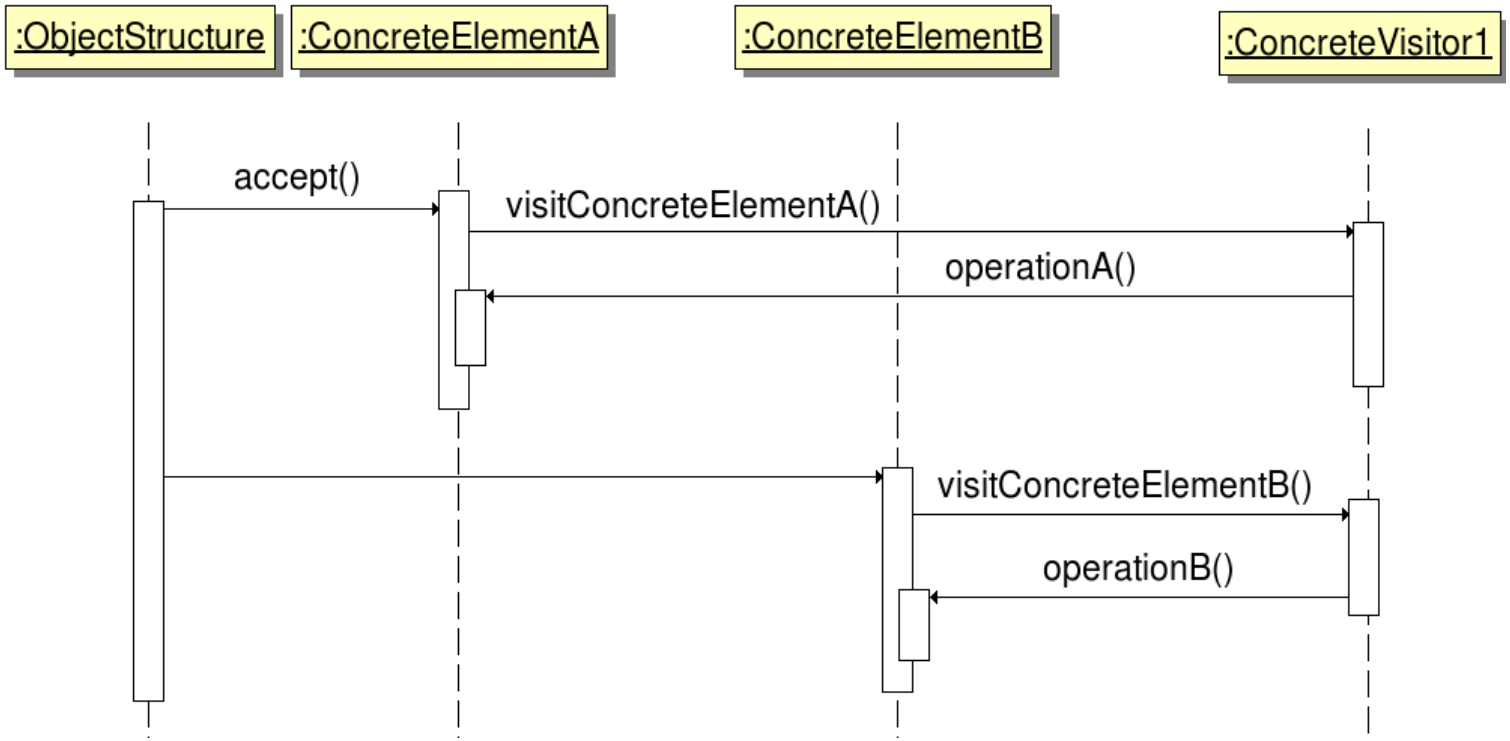


- Now the `Node` class is much simples
- it only need to provide a *hook* for allowing visitors to visit it

Generic UML diagram



Message Sequence Chart



Applicability

- Use the Visitor pattern when
  - an object structure contains many classes of objects with different interfaces, and you want to perform operations on the elements of the structure
  - many different operations needs to be performed on objects in a structure, and you want to avoid putting such operations on the objects (decoupling)
  - the classes defining the object structure rarely or never change

Consequences

- *Visitor makes adding new operations easier*
- *Adding a new `ConcreteElement` is hard*
- similar to an `Iterator`, however the Iterator visits elements of the same type, while visitor traverses structure of objects of different types
- *Accumulating State*: since the visitor is an object, while visiting it can accumulate state, or cross-check the structure

Implementation techniques

- The visitor is able to understand the type of the element it is visiting using the technique called **double dispatch**.
- Single dispatch:
  - the operation to be invoked depends on the type of the object (or of the pointer), and on the parameter list
  - in object oriented slang, we say that it depends on the message type (the method) and on the receiver (the object) type
- Double Dispatch:
  - The operation that is invoked depends on the message type (the method) and on *two* receivers
  - *accept()* is a double-dispatch operation, because the final method that is called depends both on the visitor type and the element type
  - the technique used for the template observer is quite similar: which operation is invoked depends on the message type (update), on the receiver (the observer) and on the subject (parameter of the update)

Who performs the visit?

- Different techniques
  - The object structure
  - the Visitor
  - A separate object (an Iterator)

Outline

Motivation

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

- In many cases it is useful to define a small language to define things that need to be expressed easily
- Examples where a simple language may be useful
  - Configuration files for creating objects
  - List of complex parameters
  - Rules to configure filters, etc.
- If the language is complex (for example, a scripting or programming language), it is better to use classical tools like parser generators
- However, when we want to implement a simple thing, then it may be useful to do it by hand in C
- In the following example, we will assume to build a simple interpreter for regular expressions

Example

- Grammar rules

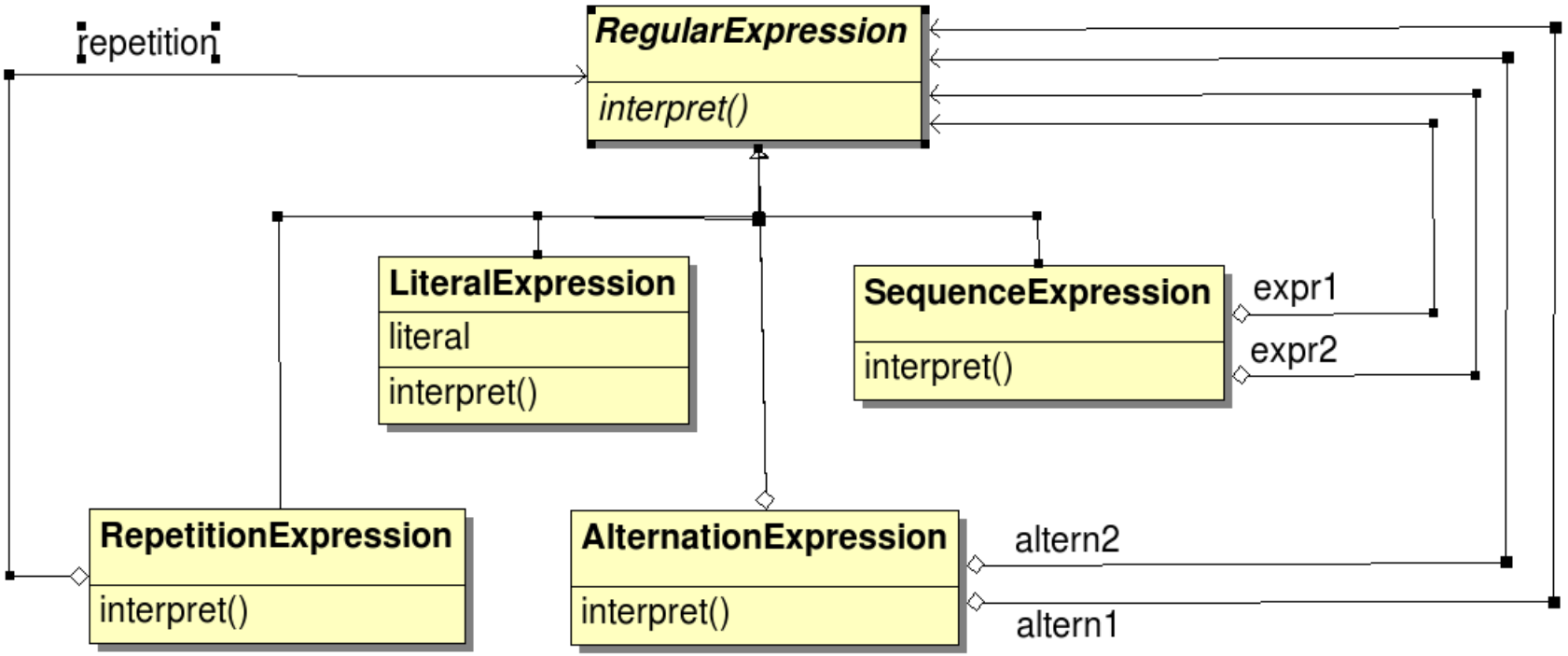
```
expression ::= literal | alternation | sequence | repetition |
    '(' expression ')'
alternation ::= expression '|' expression
sequence ::= expression '&' expression
repetition ::= expression '*'
literal :: = 'a' | 'b' | 'c' | ... {'a' | 'b' | 'c' | ... }*
```

- Expression is the starting rule
- Literal is a *terminal* symbol

Abstract Syntax Tree

- To implement the previous grammar, we prepare a class for each rule
- each class derives from an abstract class
- at the end of the parsing we must obtain an *abstract syntax tree* that will represent the expression

UML representation

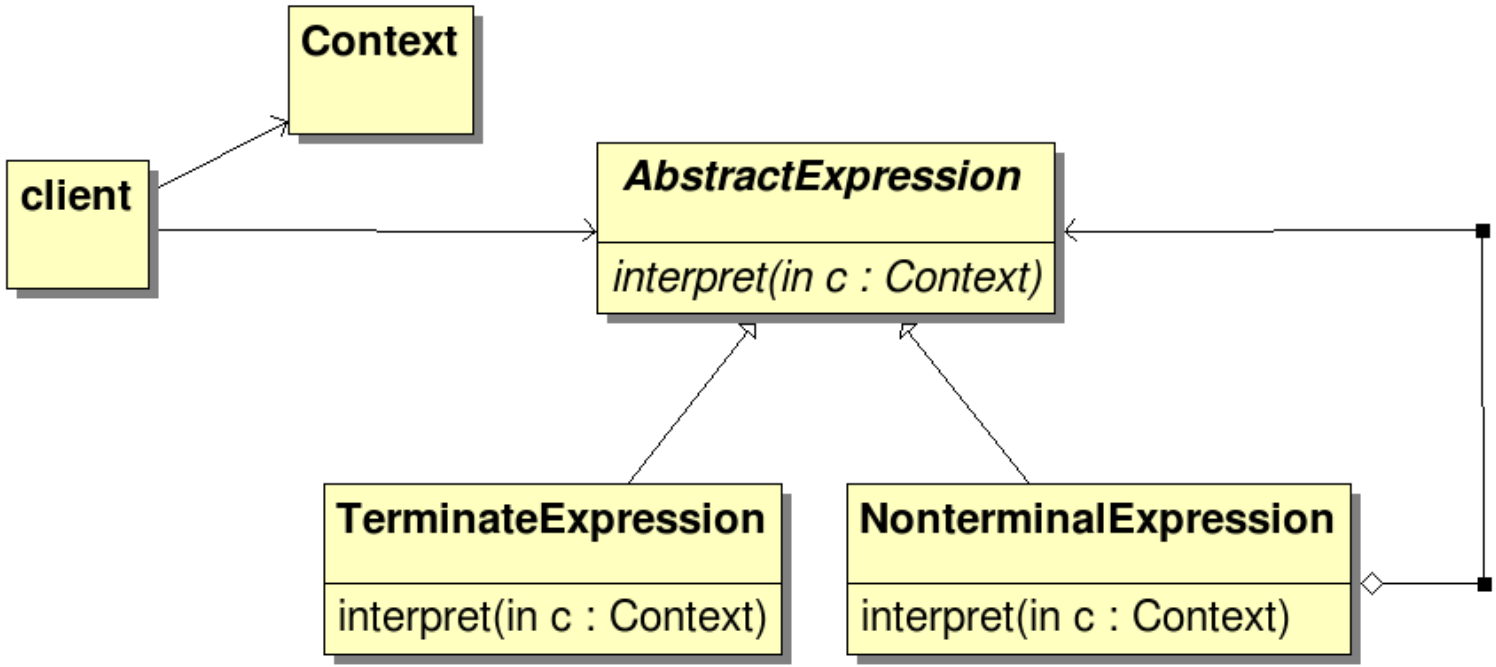


How the interpret works

- The abstract syntax tree must be built by a parser (not part of this pattern)
- once the tree is built, we can use it in our program.
  - For example, we could pass the interpret a sequence of characters, and it will tell us it the sequence respects the regular expression
  - we would also pretty-print the expression, or transform it into another representation (for example a finite state machine)

UML representation

- A general UML representation is the following



Participants

- **AbstractExpression** (RegularExpression)
  - it represents the abstract interface for the node in the tree
- **TerminalExpression** (LiteralExpression)
  - Represents the leaf of the tree, cannot be further expanded
- **NonTerminalExpression** (SequenceExpression, AlternationExpression, etc.)
  - this class represents a rule in the grammar
  - it is also an intermediate node in the tree, can contain children
- **Context**
  - Contains global information useful for the interpret
- **Client**
  - builds the abstract syntax tree via a parser
  - calls the interpreter operation to carry on the interpretation of the language

Consequences

- *It's easy to change and extend the grammar*
  - appropriate classes can be written, existing classes appropriately modified
- *Easy to implement the grammar*
  - Classes are easy to write and often their generation can be automated by a parser generator
- *Complex grammars are hard to maintain*
  - When the number of rules is large, you need a lot of classes
  - also, not very efficient to execute
- Adding new ways to interpret expressions
  - Since you have the tree, you can do many things with it
  - by using a Visitor pattern, you can easily add new operations without modifying the classes

Example of parser

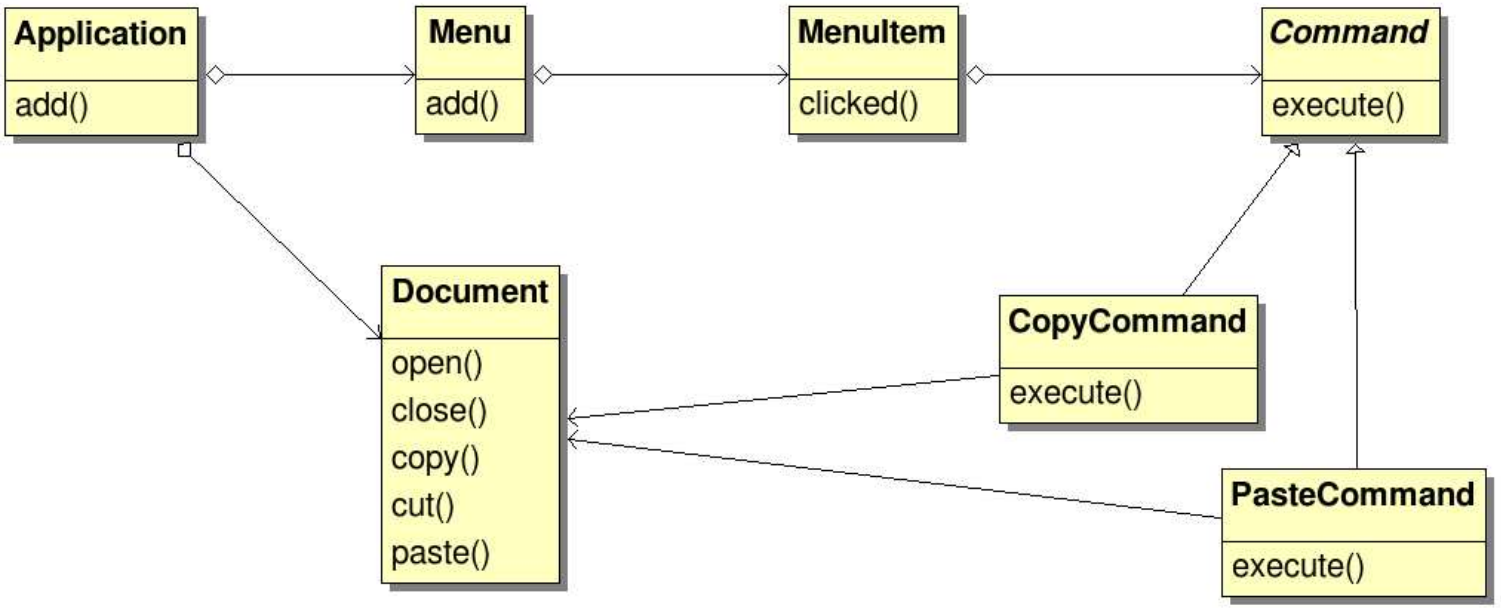
- In the code

Outline

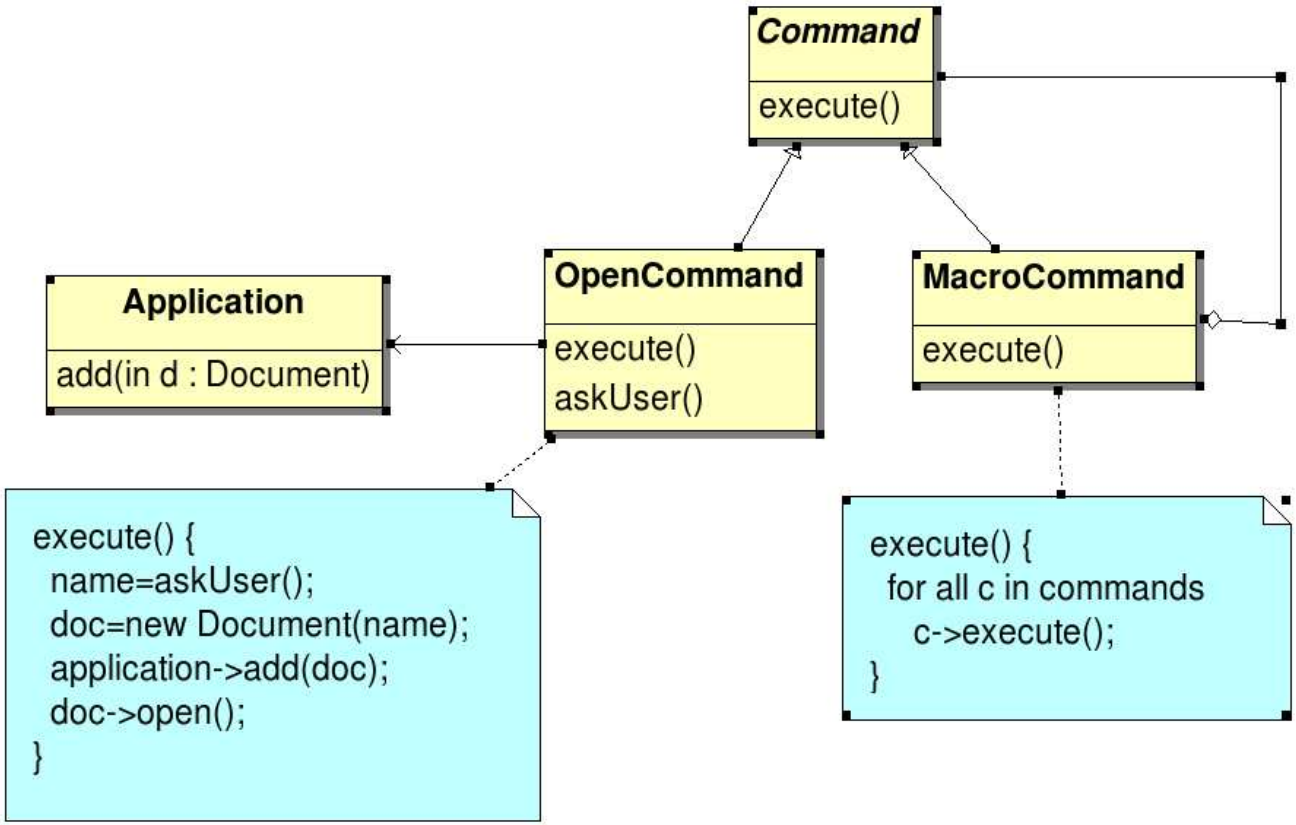
Motivation

- Sometimes it is necessary to issue requests to objects without knowing anything about the operation being requested, or the receiver of the request
  - Example: when pressing a button, something happens that is not related or implemented to the Button class
  - In many cases, exactly the same operation can be performed by a menu item, or by a button in a toolbar
- We want to encapsulate commands into *objects*
- This patterns is the OO equivalent of C *callbacks*
- Other uses
  - Undo/redo of commands
  - Composing commands (macros)

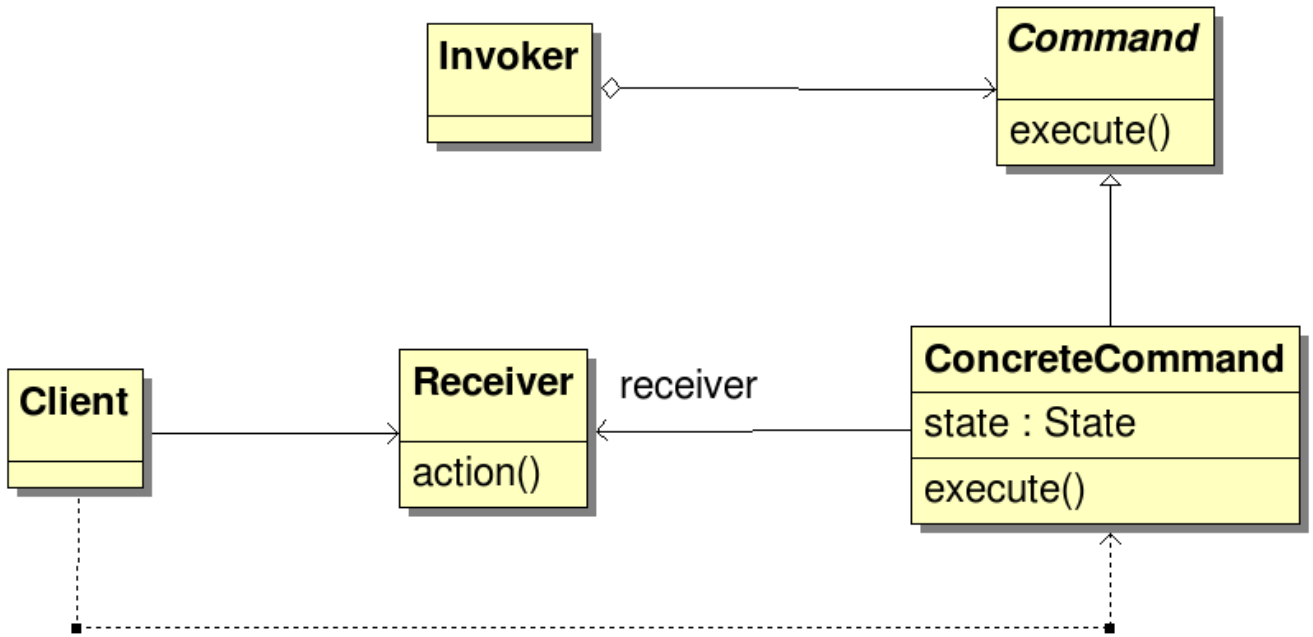
UML example


Application
add()
Menu
add()
MenuItem
clicked()
Command
execute()
Document
open()
close()
copy()
cut()
paste()
CopyCommand
execute()
PasteCommand
execute()

Implementing macros



General UML structure



Undo/Redo

- It is not always possible
  - The operation should be reversible
  - Need to add and undo() operation in the Command abstract class
  - The command may need to carry additional state of the receiver inside
  - We need an history list (how far should we go with the history?)
- Using prototype
  - We could use a Prototype pattern to create copies of commands, customise with the internal state of the receiver, and then save the copy on the history

Differences with callbacks

- Commands are objects, not just functions
  - They can carry state information on the receivers
  - They can carry information on the history itself
- The Invoker only needs to know the general interface of the command (execute()), not the specific internal information (i.e. parameters, etc.) which are decided at creation time

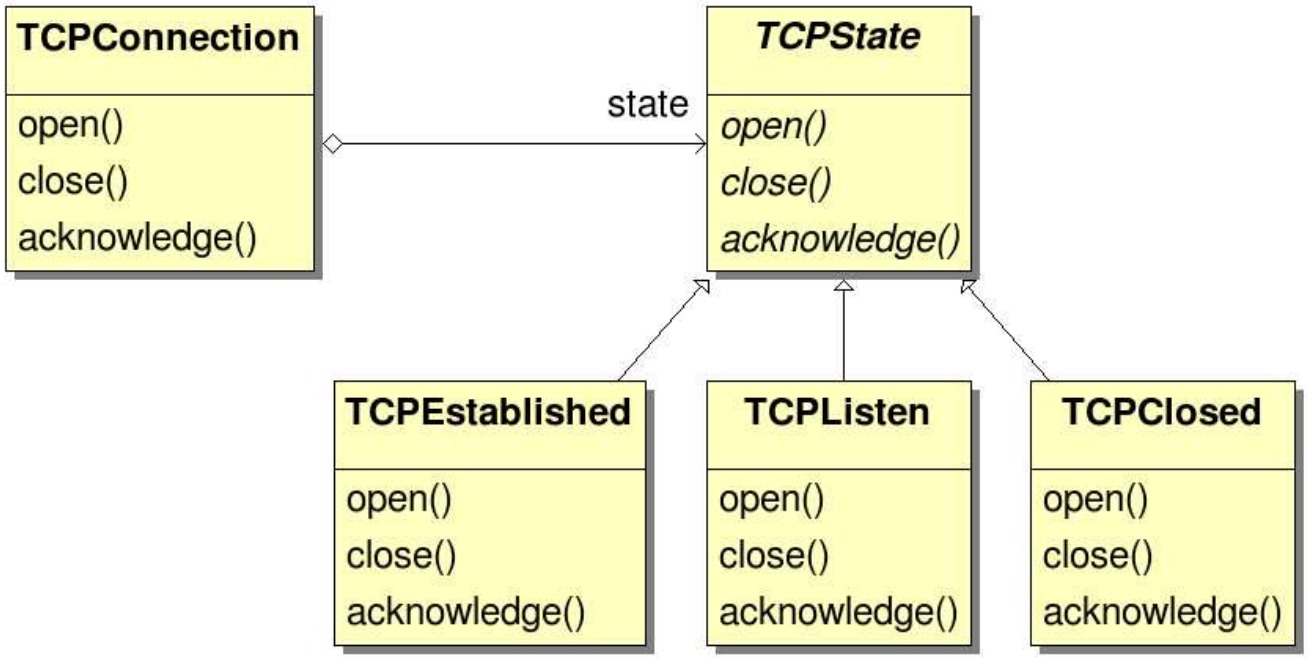
Outline

The State pattern

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

- This pattern is useful to implement simple state machines
- The idea is to implement each state with a different class, and each event with a different method
- Consider a library to implement the TCP protocol
  - A TCPConnection can be in one of several different states
  - For example, the connection can be void, established, closing, etc.
  - the response to a request of open depends on the current state of the connection: only if the connection is not yet established we can open it
- this behaviour can be implemented as follows:
  - An abstract class TCPState that implements a method for each possible request
  - the derived classes represent the possible states
  - only some of them will respond to a certain request

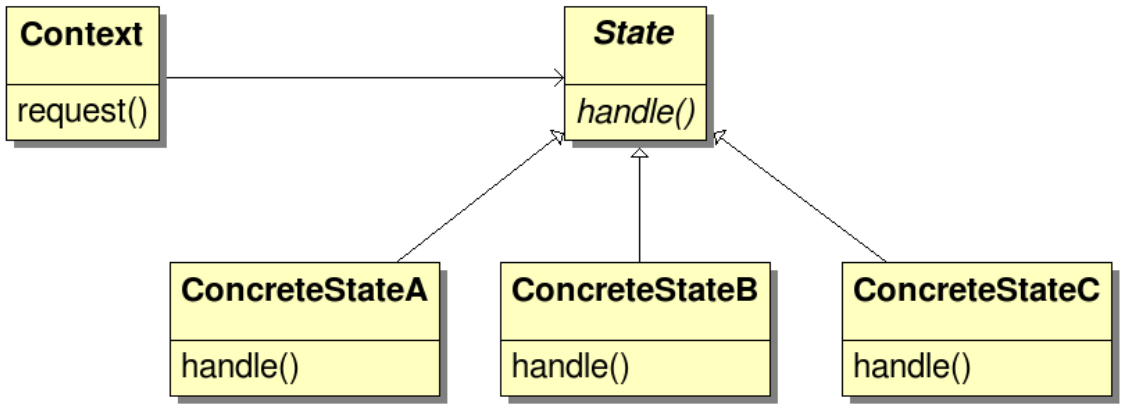
Example


TCPConnection
open()
close()
acknowledge()
state
TCPState
open()
close()
acknowledge()
TCPEstablished
open()
close()
acknowledge()
TCPListen
open()
close()
acknowledge()
TCPClosed
open()
close()
acknowledge()

Applicability

- Use the State pattern in one of the following cases
  - an object behaviour depends on its state, that will change at run-time
  - operations have large, multi-part, conditional statements that depend on the object state. This state is usually represented by one or more enumerated constants

UML diagram


Context
request()
State
handle()
ConcreteStateA
handle()
ConcreteStateB
handle()
ConcreteStateC
handle()

Participants

- **Context** (TCPConnection)
  - defines the interface of interest to clients
  - maintains an instance of a ConcreteState subclass that defines the current state through a pointer to the abstract State class
- **State** (TCPState)
  - defines and interface for encapsulating the behaviour associated with a particular state of the Context
- **ConcreteState** subclasses
  - each subclass implements a behaviour associated with a state of the Context

Consequences

- *Localizes state-specific behaviour and partitions behaviour for different states*.
  - All behaviour associate with a particular state is concentrate into a single class (ConcreteState).
  - new states and transitions can be easily added
  - the pattern then avoid large if/then/else conditional instructions
  - however, distributing information in state classes may not be appropriate for complex behaviour, because it increases the amount of interaction and dependencies between classes
- *it makes state transitions explicit*.
  - a transition is a change in the state object, therefore is quite visible

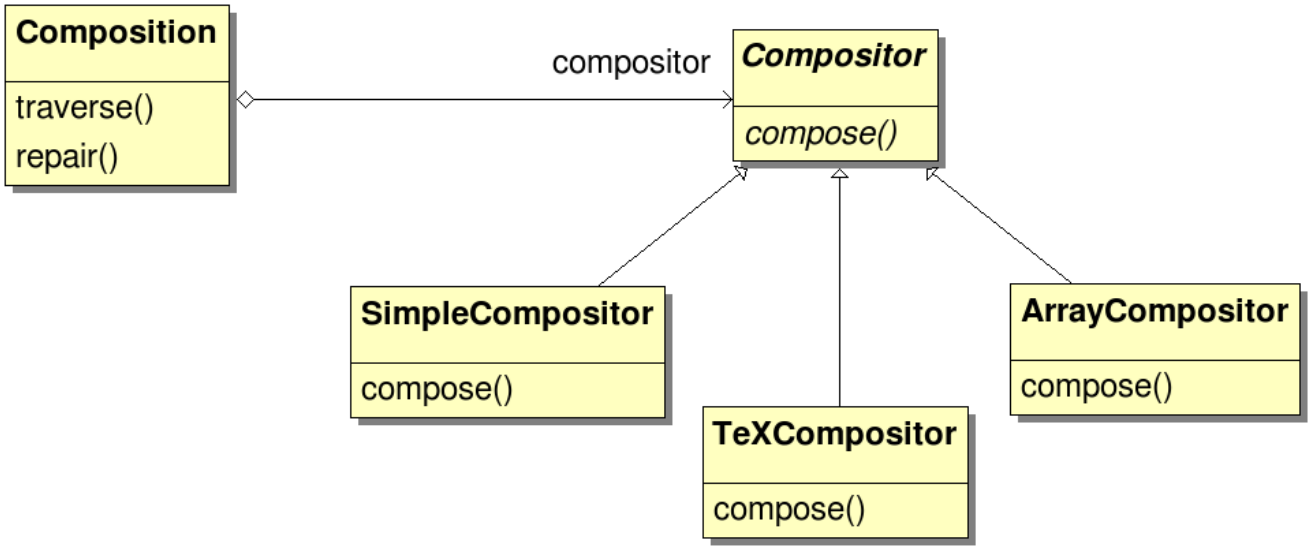
Outline

Strategy

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

- In general, it is useful to delegate an algorithm to a function, instead of embedding it into the normal code
  - we make the algorithm general and reusable
  - we can easily change the algorithm by substituting the function
- In object oriented programming, objects can be used instead of functions
- An example:
  - many algorithms exist for breaking a stream of text into lines
  - hard-wiring them into the class that uses them is undesirable, because it goes against the single-responsibility principle
  - therefore, we could define an hierarchy of “function objects” that behave like functions

UML diagram


Composition
traverse()
repair()
compositor
Compositor
compose()
SimpleCompositor
compose()
TeXCompositor
compose()
ArrayCompositor
compose()

Structure

- **SimpleCompositor** implements a simple strategy that determines line breaks one at a time
- **TeXCompositor** implements the TeX algorithm for finding line breaks. This strategy tries to optimise line breaks globally, that is one paragraph at a time
- **ArrayCompositor** implements a strategy that selects breaks so that each row has a fixed number of items. It's useful for breaking a collection of icons into rows, for example

Structure

- **SimpleCompositor** implements a simple strategy that determines line breaks one at a time
- **TeXCompositor** implements the TeX algorithm for finding line breaks. This strategy tries to optimise line breaks globally, that is one paragraph at a time
- **ArrayCompositor** implements a strategy that selects breaks so that each row has a fixed number of items. It's useful for breaking a collection of icons into rows, for example
- A Composition maintains a reference to a Compositor object
- we can change strategy both at compile time and at run-time
- Why using classes instead of functions?
  - Objects can carry state, while functions can't