Behavioural Patterns

Lect. PhD. Arthur Molnar

Babes-Bolyai University

arthur@cs.ubbcluj.ro

Overview

1 Behavioural Patterns

- Intro
- Chain of Responsibility
- Command
- Iterator
- Mediator
- Memento
- Observer
- State
- Strategy
- Template Method
- Visitor
- Discussion of Behavioural Patterns

Intro

- Concerned with algorithms
- How the responsibility is assigned to objects
- Provides ways for objects to fulfill requirements while loosely coupled

Chain of Responsibility

Chain of Responsibility pattern

Avoid coupling the request sender with the receiver(s).

- Request sender does not know about the receivers
- Receivers can be chained, with the first compatible one handling the request

Chain of Responsibility

Motivating example: context-sensitive help within a GUI application

- GUI's are defined hierarchically, with each component having a parent
- Each component might handle a help request by providing a help handler
- In case a component does not provide context-based help, its parent should do it

Chain of Responsibility

- The widget where help request is issued (**e.g.** *a button*) does not know who exactly will provide it
- Each object must share a common interface for handling help requests (**e.g.** a *HelpHandler*)

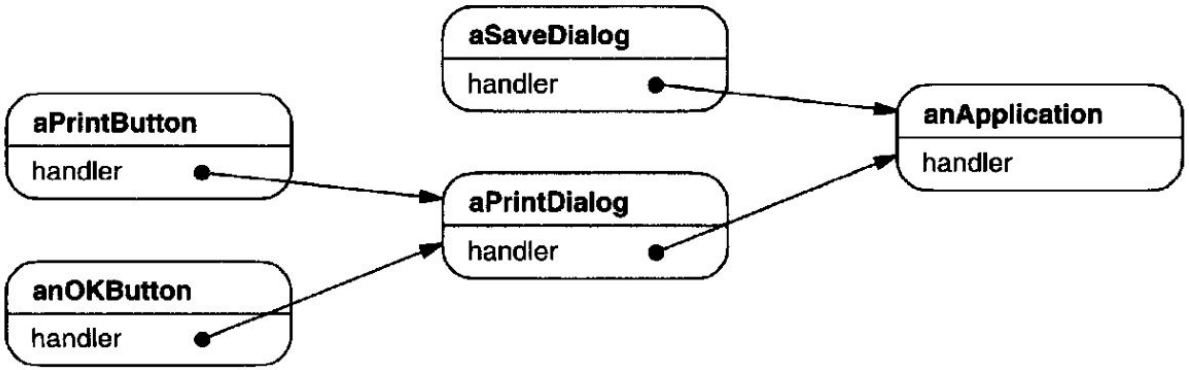

Figure: from [1]

Chain of Responsibility

The **HelpHandler** implements support for the responsibility chain pattern

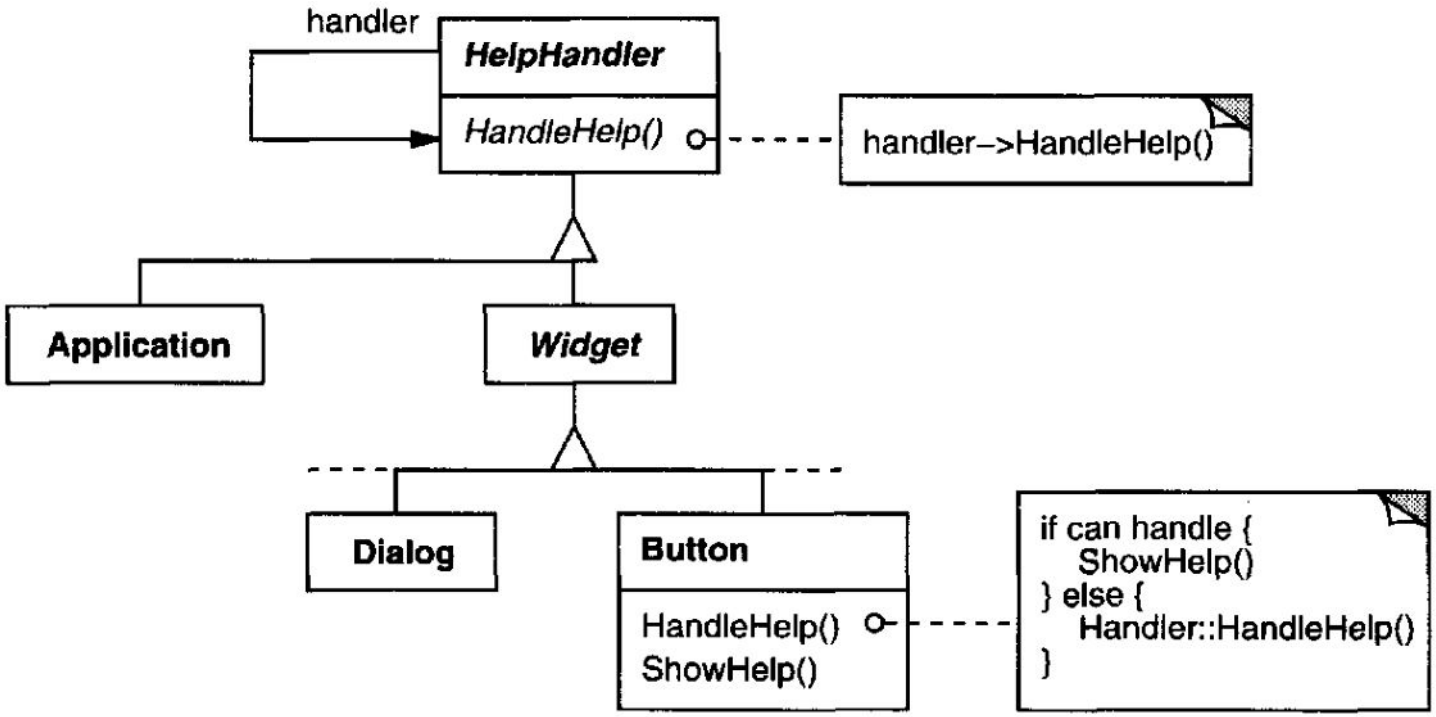

Figure: from [1]

Chain of Responsibility

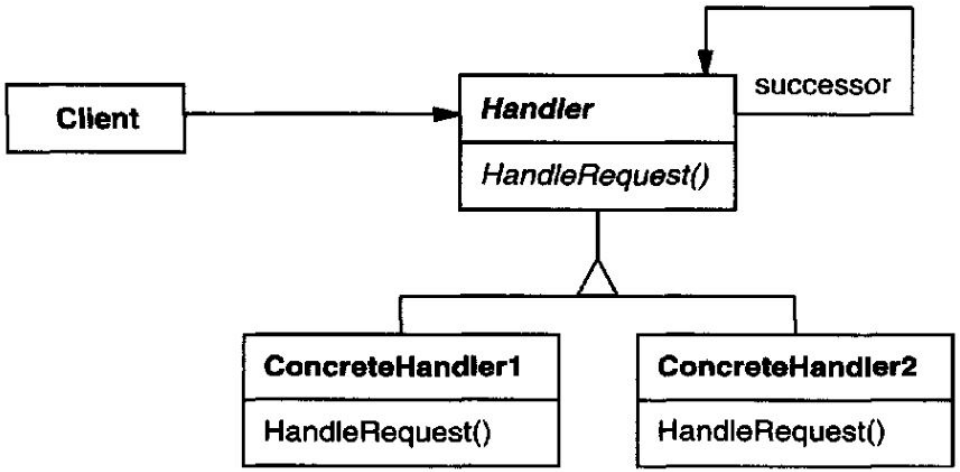

Figure: General case, from [1]

Using the chain of responsibility

+ Decouple the handler(s) from the message source
+ Provide the handler(s) dynamically
– No guarantee that a request will be handled

Chain of Responsibility

Logger example source code

git: /src/ubb/dp/behavioural/ChainOfResponsibilityLogger.java

Computer example source code

git: /src/ubb/dp/structural/CompositeExample.java

Command

Command pattern

Encapsulate a request as an object.

- Allows issuing request without knowing the operation **or** the receiver(s)
- The request can be passed around between application systems, as it is encoded within an object
- The key is an abstract *Command* class that declares an interface for executing operations
- Great for implementing undo/redo ☺

Command

Motivating example - a menu-based GUI

- Each choice in the menu is a *MenuItem* (**e.g.** similar to Java Swing)
- When clicked, each menu item runs the *execute()* method of its command object

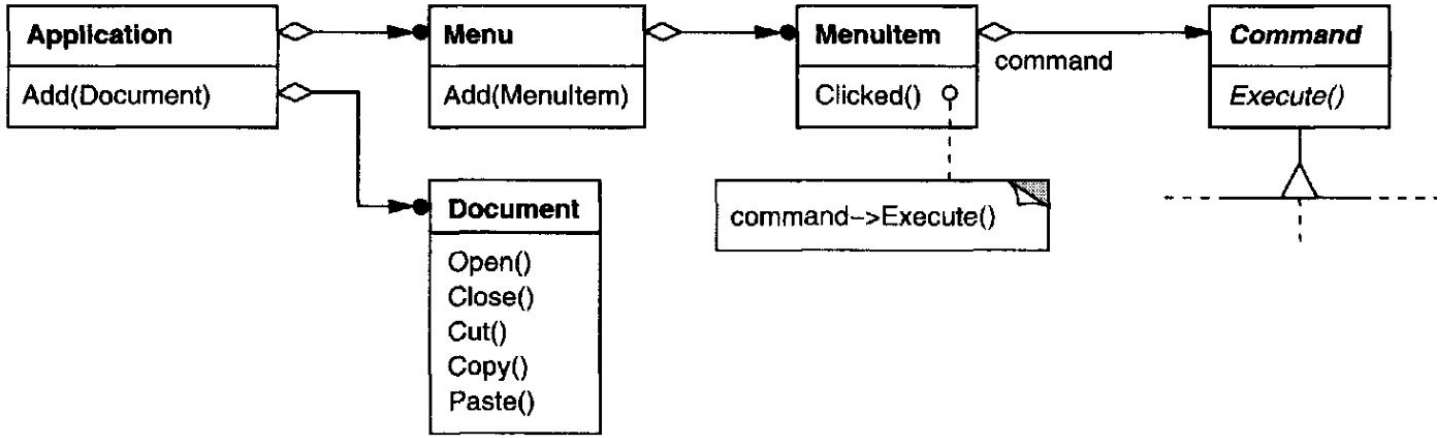

Figure: from [1]

Command

Motivating example - undo/redo

- Some operations are *simple* (**e.g.** delete a rental car that has no rental history)
- Some operations are compounded (**e.g.** if you delete a rental car, you must also clear its rental history; when undoing this, you have to restore the full state)

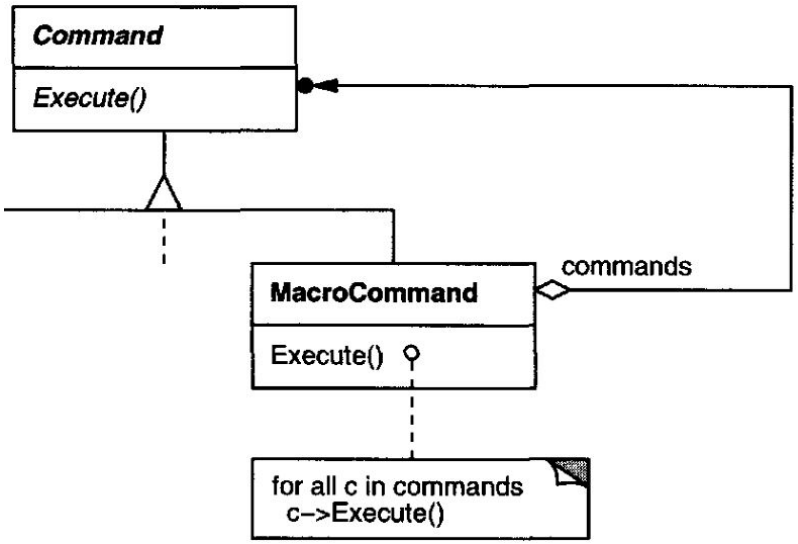

Figure: from [1]

Command

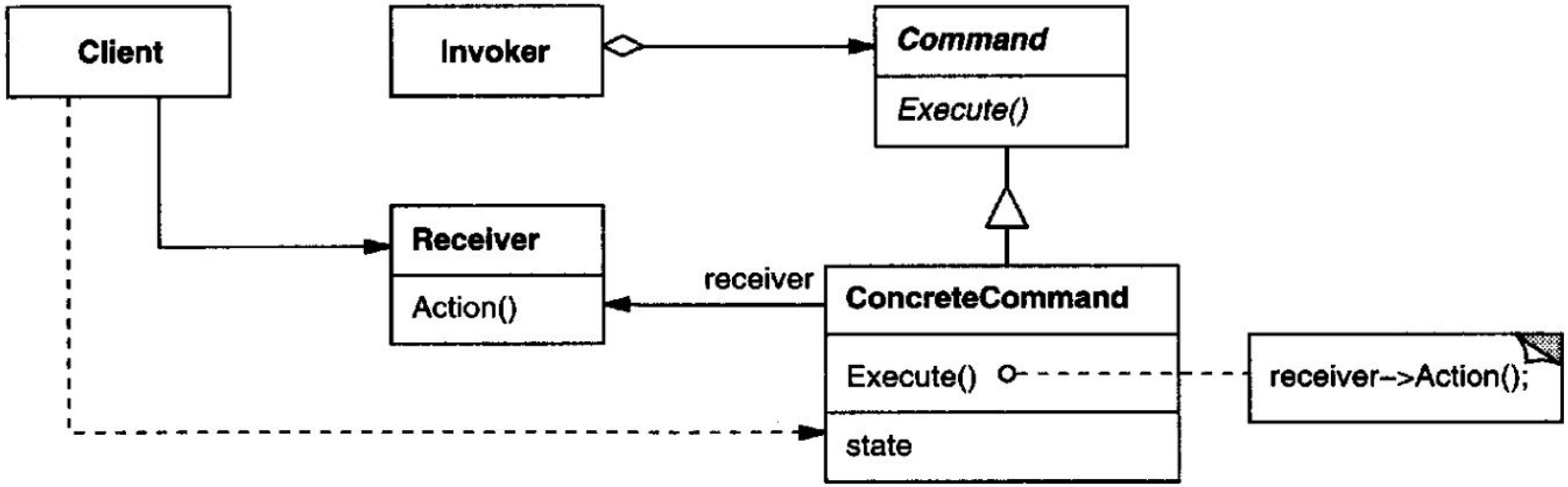

Figure: General case, from [1]

Command

When and how to ... command

- The pattern allows changing request dynamically, as well as reusing them (**e.g.** a *menu item* and *button* might share a command)
- Specify and queue operations for later execution
- In many cases implemented using **callbacks**
- **Commands** are first-class objects (**?**)
- Decoupling concrete commands from issuers and receivers makes it easy to create new ones

Command

Command example source code

git: /src/ubb/dp/behavioural/command

- Command classes are implemented in *UndoController*
- Command objects are created by the *Controller* classes (except the Undo Controller)
- Run the example using the modules *CommandExample**

Command

Command example source code

git: /src/ubb/dp/behavioural/memento

- Command classes are implemented in package **/commands**
- Used as part of the undo/redo mechanism

Iterator

Iterator pattern

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

- You need a way to traverse an aggregate (**e.g.** list, tree, GUI widget structure)
- Separate the interface for accessing elements from the aggregate itself (avoid *interface pollution*)
- Might want to traverse the elements in different ways (**e.g.** preorder, start-end, end-start)

Iterator

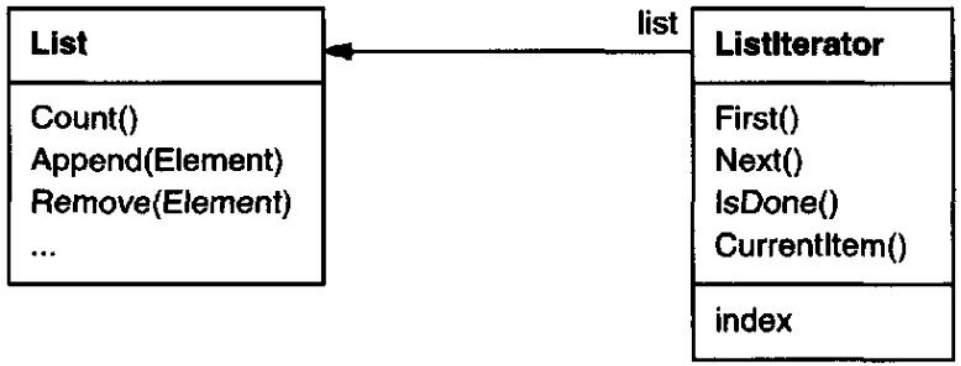

Figure: List iterator, from [1]

- Iterator must be supplied with the aggregate to traverse
- Some of these operations can be unified (**e.g.** have only *next()*, *hasNext()*)
- Separation between aggregate and iterator allows using several iterators over the same structure at once
- The *aggregate* and its *iterator* are coupled

Iterator

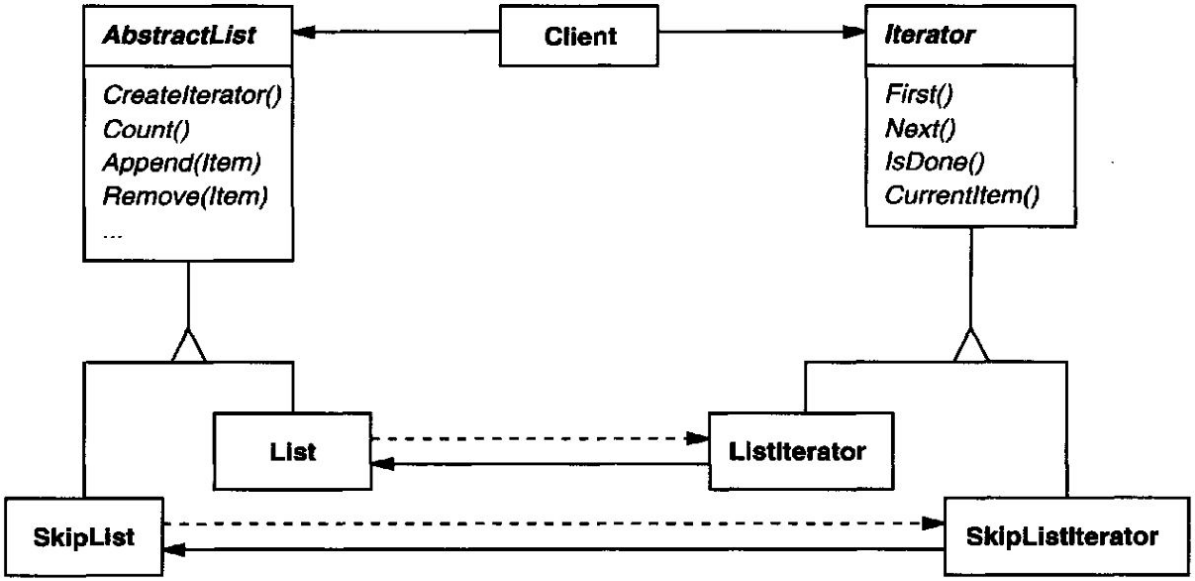

Figure: List iterator, from [1]

- Polymorphic iteration
- Concrete iterator can be obtained directly from the aggregate itself, using a *factory method*

Iterator

Consequences:

- Iterators allow you to implement different traversals
- They simplify the interface of the aggregate
- **You can have multiple traversals at the same time**

Figure: From the Interwebs

Iterator

Internal iterator

- Controlled by the iterator itself
- Client provides the operation to perform, iterator handles it

External iterator

- Controlled by the client
- More flexible
- The one you know ☺

Iterator

Implementation details:

- **Iterator robustness** - what happens if you modify the aggregate during iteration? (creating a copy of the aggregate must be avoided)
- Aggregate might have to share state with its iterators, breaking *encapsulation* (**e.g.** C++ **friend** classes)
- **NullIterator** - one that has always finished the iteration

Iterator

Java

- Java 5 introduced the *Iterable* interface, which the *Collection* interface extends

Python

- Prescribes iterator sequence as part of the language
- We have *__iter__*, *next()* and *StopIteration*

Iterator

NullIterator source code

git: /src/ubb/dp/structural/CompositeExample.java, the *SimpleEquipment* is a leaf node in the composite and returns a NullIterator

Java Iterator source code

git: /src/ubb/dp/structural/CompositeExample.java, check the iteration when computing the desktop's total power consumption and price

Python Iterator source code

git: /src/ubb/dp/behavioural/IteratorExample.py, an iterator built for a *List* adapter wannabe ☺

Mediator

Mediator pattern

Define an object that encapsulates how a set of objects interact.

- Promotes low coupling, as objects do not refer each other directly
- You can vary their interaction using the mediator (or by implementing another mediator type)
- Partitioning a system into interacting objects improves software characteristics (**e.g.** reusability, understandability, maintainability)
- Having many interconnections reduces these desirable characteristics

Mediator

Provides a centralized location for these widgets to interact while loosely coupled

Figure: from http://www.java2s.com/Code/Java/

Mediator

The *FontDialogDirector* object acts as the mediator between the widget objects

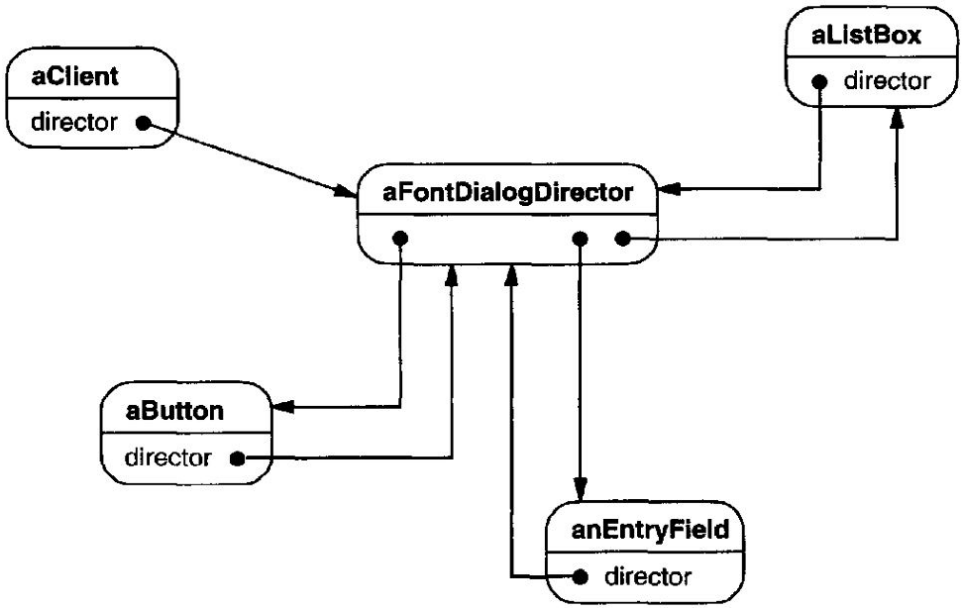

Figure: from [1]

Mediator

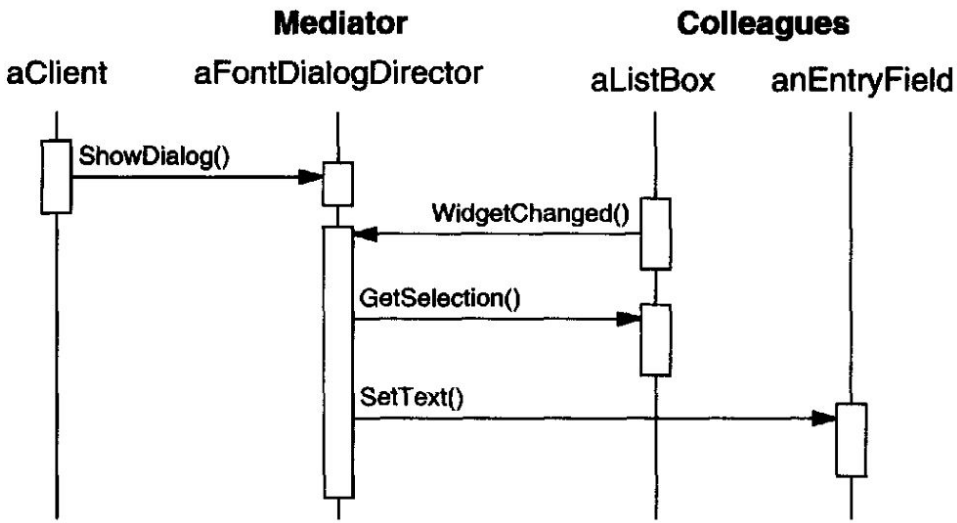

Figure: from [1]

- List box tells the director it's changed
- Director gets the selection from the list box and passes it to entry field
- Director enables corresponding buttons

Mediator

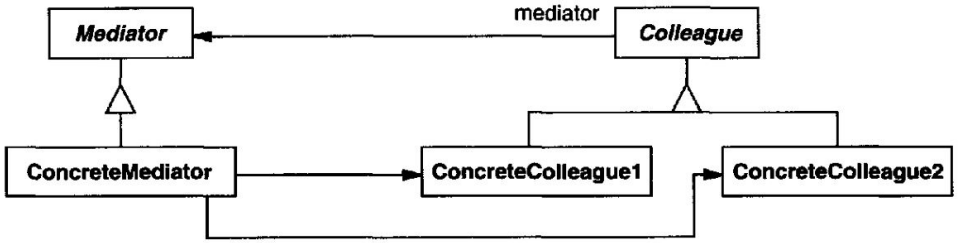

Figure: from [1]

- **Mediator** defines the interface for communicating
- **ConcreteMediator** implements the behavior
- **Colleague** classes communicate with the mediator

Mediator

When **to use Mediator**

- A set of objects communicate in well-defined but complex ways
- A behaviour distributed between classes should be customized without subclassing

Consequences

- It limits subclassing, by grouping behaviour into a class
- Simplifies communication protocols between objects
- Abstracts and centralizes control
- **Mediator** and **Observer** are related

Mediator

Mediator source code

git:/src/ubb/dp/behavioural/MediatorExample.java

Memento

Memento pattern

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

- Useful for implementing checkpoints, rollback, undo/redo ☺
- Solves the issue of externalizing state without breaking object encapsulation

Memento

Motivating example - a system for unlimited undo/redo

- We want to undo/redo moving shapes in a graphical editor
- Each operation is modelled as an instance of the *Command* pattern
- Each operation must keep the state that needs to be restored, a *Memento*
- The memento object stores the internal state of an *originator*
- The state of *requested from* / *restored to* the originator when needed

Memento

How it works

- At each operation, the *Editor* creates a *Memento* object and adds it to the history
- At *undo* or *redo*, the state is restored from history

Memento

In the general case...

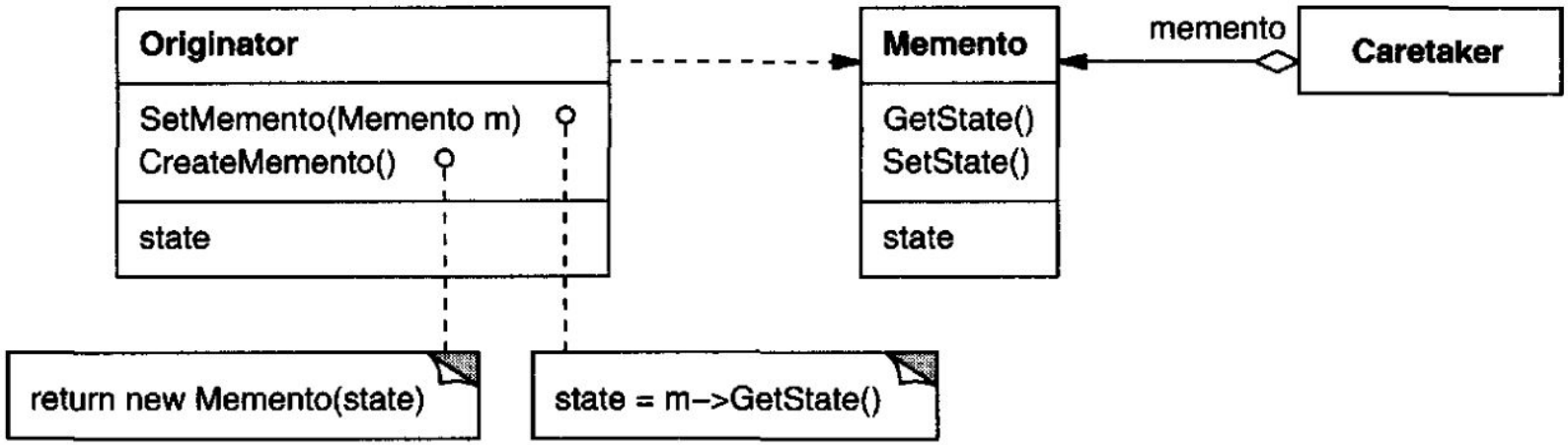

Figure: from [1]

Memento

In the general case...

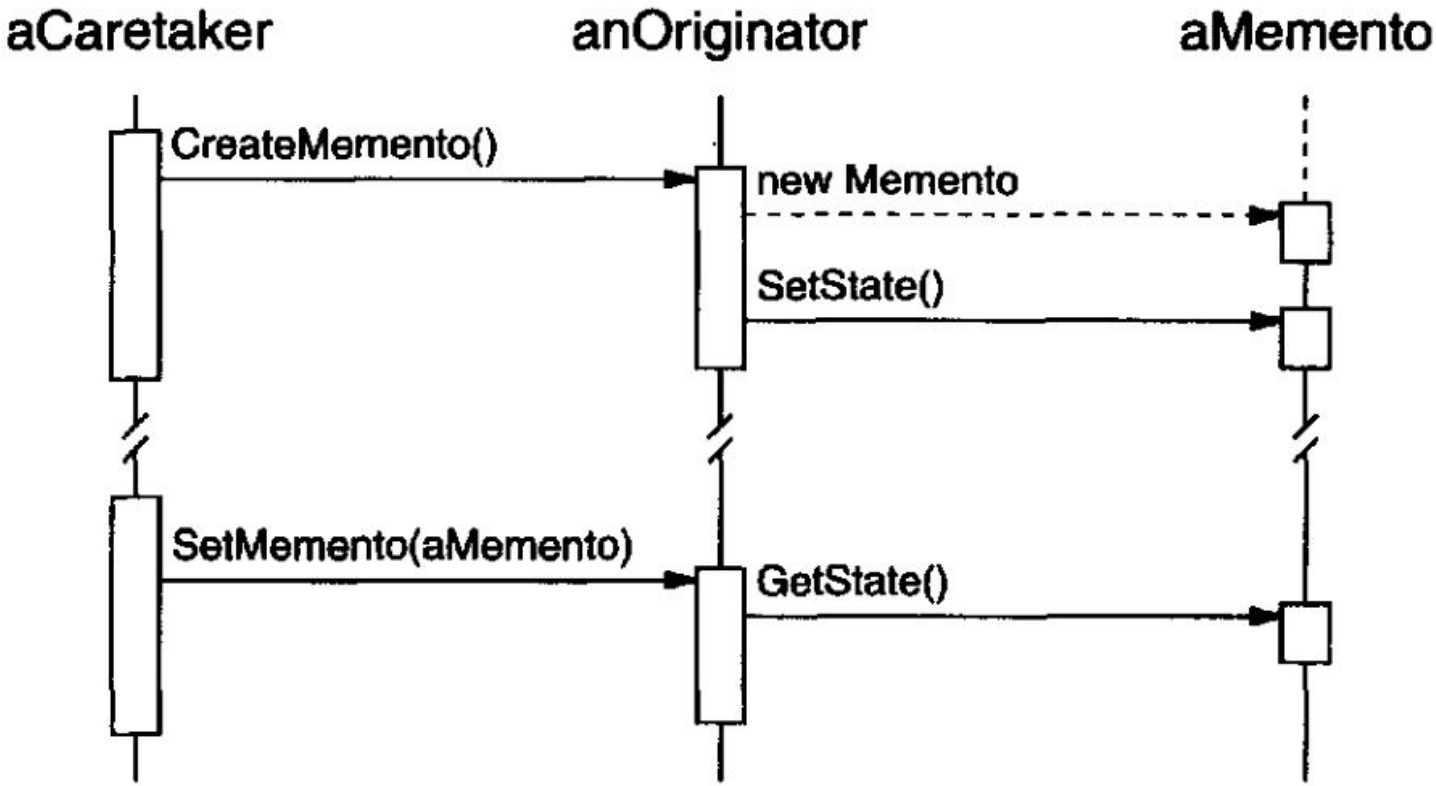

Figure: from [1]

Memento

Consequences...

+ Preserves encapsulation boundaries
+ Simplifies the originator, as it no longer has to "remember" its state
– Using them might be expensive
– Hidden costs for maintaining, especially in non-GC languages

Memento

Memento source code

git:/src/ubb/dp/behavioural/memento

Observer

Observer pattern

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

- Solves the problem of maintaining consistency between objects (problem is similar to *Mediator*)
- Key roles are the subject (*Observable*) and the *observer*
- One subject can have many observers, and one object can observe several subjects
- Implements the *publish-subscribe* interaction

Observer

When to use observer pattern

- Need to encapsulate an abstraction in several objects that depend on each other
- One change requires other changes, but in a flexible way
- The publisher notifies its subscribers, but it does not know exactly **who** or **how many** these subscribers are

Observer

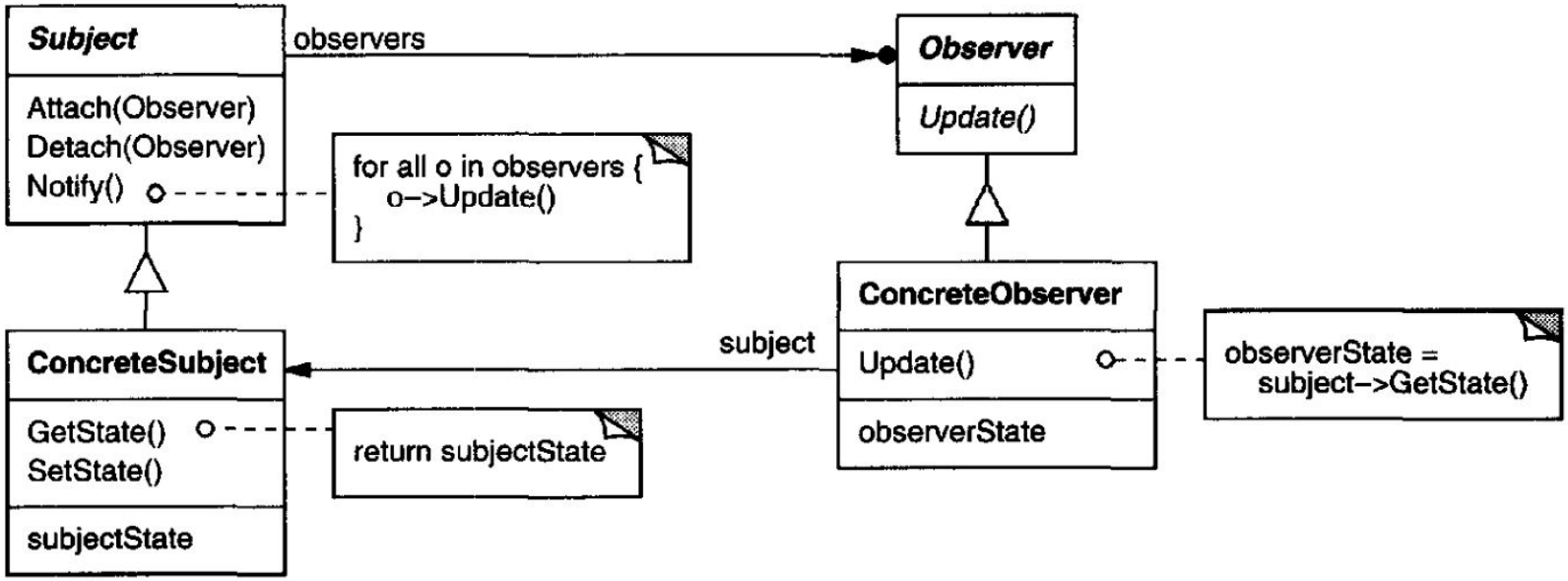

Figure: from [1]

Observer

- Subject notifies its observers when a change occurs
- Observers might further query the subject after being notified

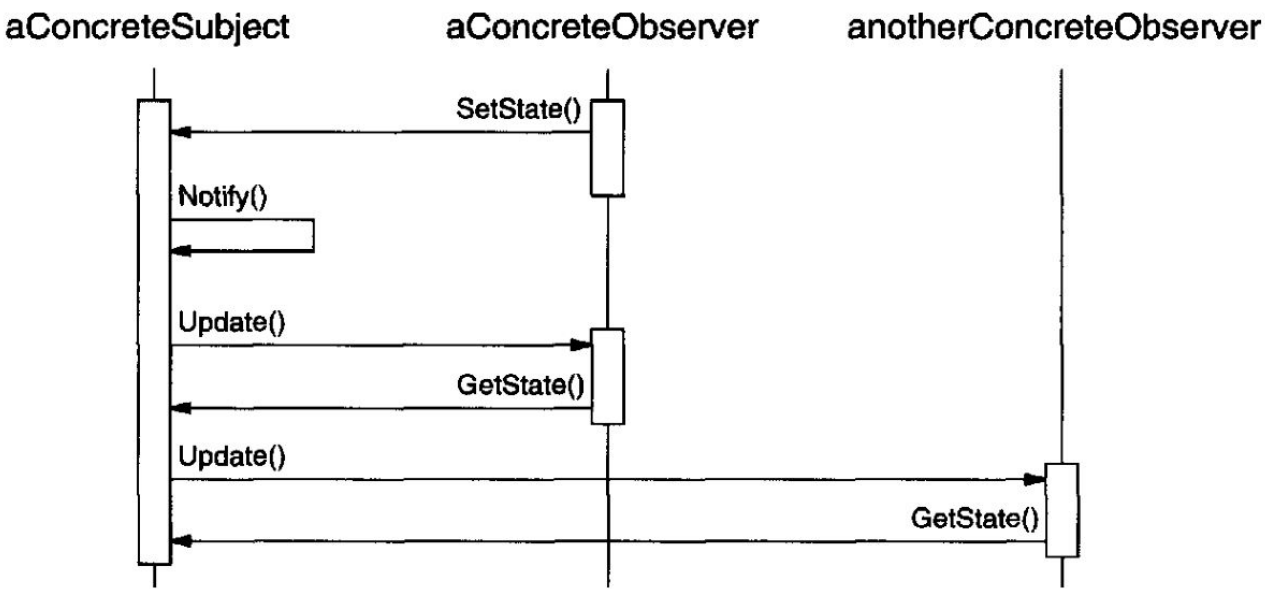

Figure: from [1]

Observer

Consequences

- \+ Main idea is that **subjects** and **observers** can vary independently
- \+ Supports broadcast communication (where senders do not have knowledge of receivers)
- – Observers can change the subject in a matter that results in a lot of further updates

Observer

Implementation details

- Observers can track more than one subject, raising the issue that notifications must provide information about the originator (**e.g.** Java's *ActionEvent.getSource()*)
- Dangling references to subjects in non-GC languages!
- **Push** (subjects sends a lot of info to subscribers) versus **pull** (subscribers further query the subject) implementations
- Subjects can provide finer-grained control regarding the actions subscribers are interested in (**e.g.** our code example)

Observer

Observer source code

MouseAdapter example **git:**/src/ubb/dp/behavioural/memento

Observer source code

git:/src/ubb/dp/behavioural/observer

State

State pattern

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

Motivating example

- A *TCPConnection* class that manages a TCP network connection
- Connection can be in one of several **states**: established, listening, closed
- When the connection object receives requests, it responds differently depending on current **state**.
- Behaviour depends on the current state, represented using an *abstract base class* (ABC).

State

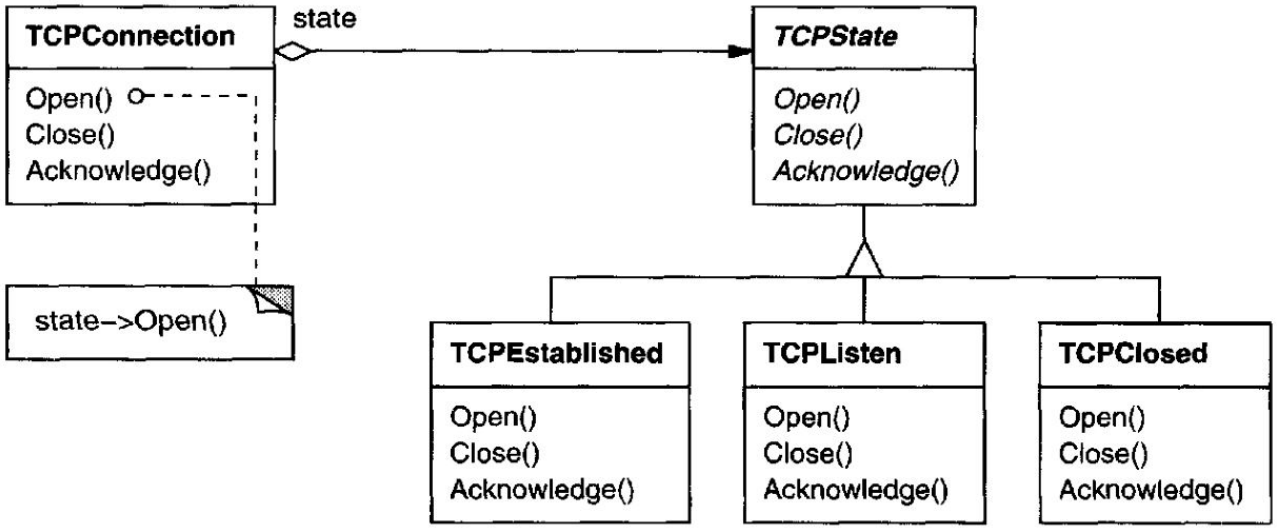

Figure: from [1]

- When the state changes, the *TCPConnection* instance changes the state object used

State

When to use the state pattern...

- An object's behaviour depends on its state, which changes at run-time
- Can be used to replace large conditional statements modeling behaviour

General structure...

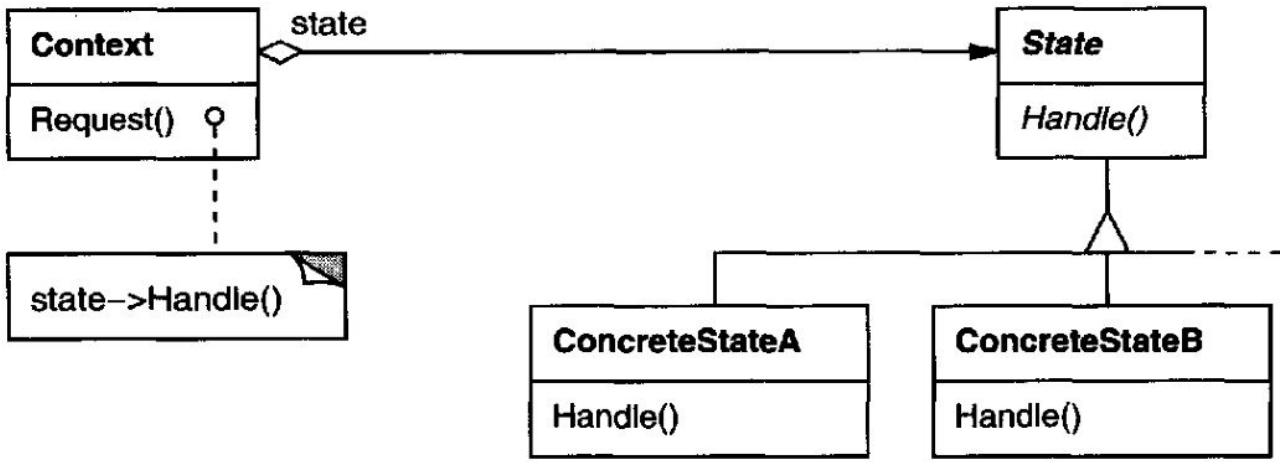

Figure: from [1]

State

Consequences

+ Localizes state-specific behaviour and partitions it for different states
+ Makes state transitions **explicit** (changes object type), instead of **implicit** (change in object internal state)
+ State objects can be shared (using the Flyweight pattern)
– Increases the number of classes, solution is less compact

State

Implementation

- Who is responsible for defining state transitions?
  - **Context** leads to centralization
  - **State** means that state classes know about each other, but new states could be easily added

State

State source code

git:/src/ubb/dp/behavioural/state

Strategy

Strategy pattern

Define a family of algorithms, encapsulate them, and make them interchangeable.

- Strategy allows algorithms to vary independently from the clients using them

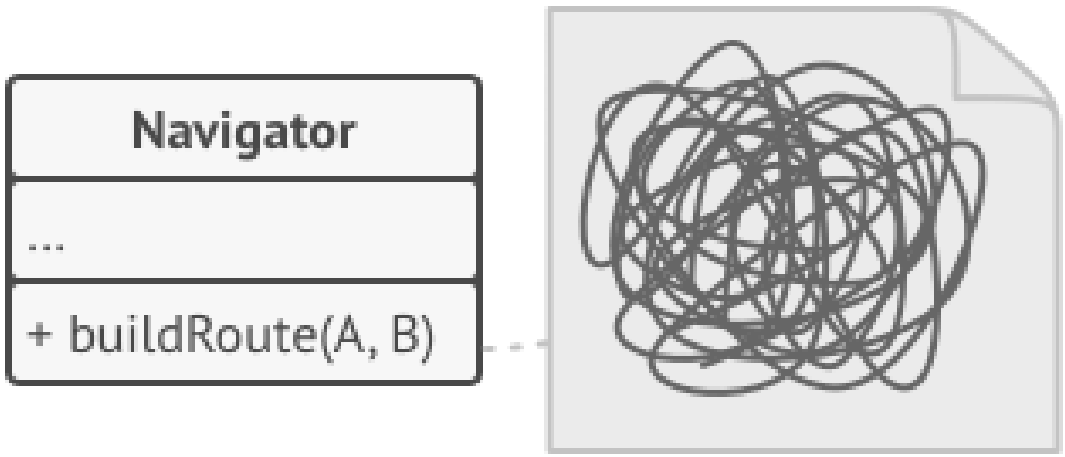

Figure: No strategy leads to complicated code that is difficult to maintain

Strategy

Motivating example[1]

- Implement a navigation app
- You can navigate between two points using a *car*, *public transportation*, or *on foot*
- Different **strategy** required for optimal path calculation in every case

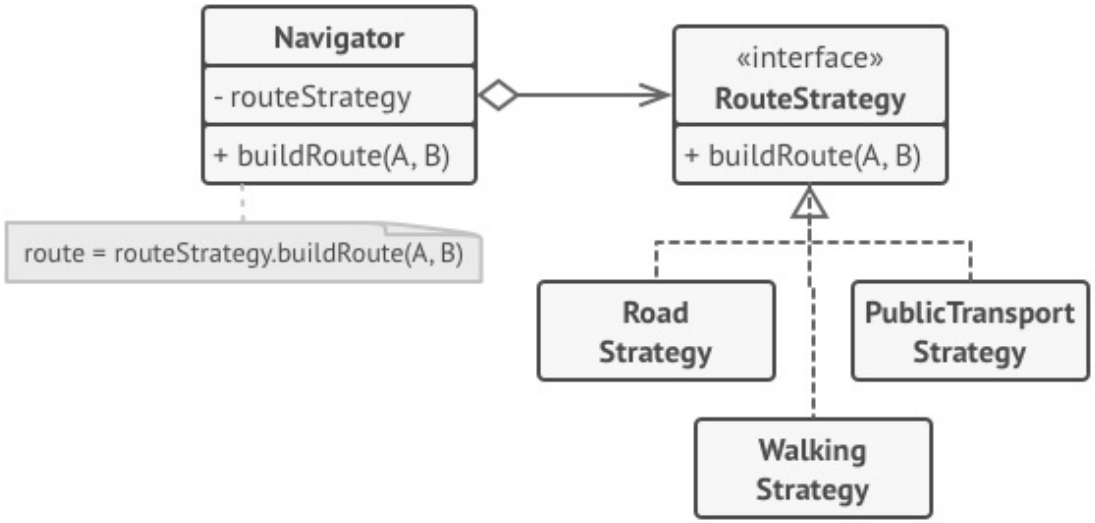

Figure: `https://refactoring.guru/design-patterns/strategy`

Strategy

- Use strategy when several object must differ only in behaviour
- Avoid exposing complex implementation details
- Avoid complicated conditional statements

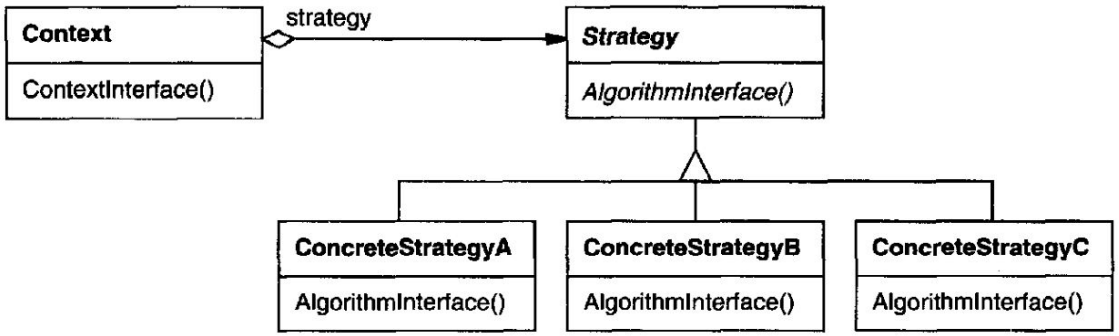

Figure: from [1]

Strategy

Conseqences

- Used to define a family of related algorithms
- You can implement strategies via subclassing too, but that hardwires the implementation into the class (remember *class* vs. *object patterns*); you also cannot vary the implementation dynamically
- Clients have to be aware of differences between strategies (context must be aware of another class's implementation)

Strategy

Strategy source code

git:/src/ubb/dp/behavioural/StrategyExample.java

Template Method

Template method

Define the skeleton of an algorithm in an operation, deferring some steps to subclasses.

Motivating example:[2]

- Create an application that analyzes all kinds of documents
- At first, you only support *.doc* files
- Later, you also add support for other file types (e.g. *.pdf*, *.csv*, *.whatever*)

[2] https://refactoring.guru/design-patterns/template-method

Template Method

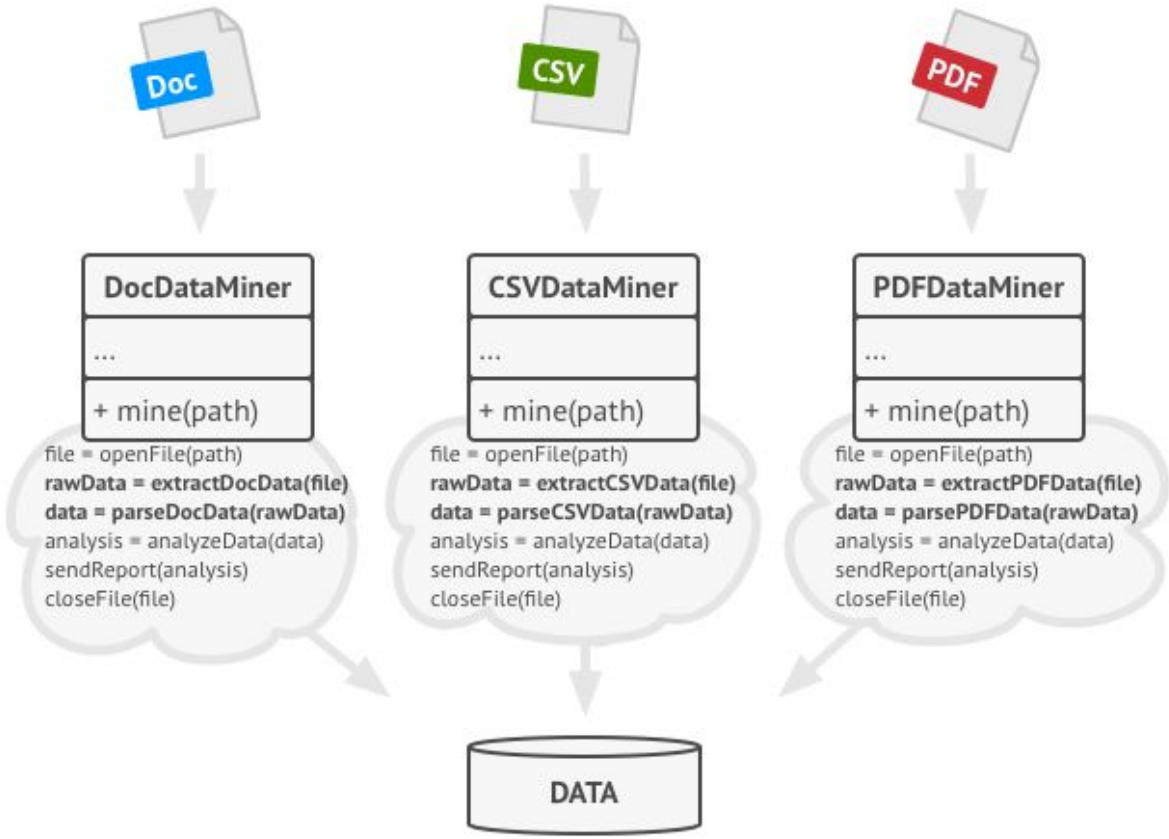

Figure: from `https://refactoring.guru/design-patterns/template-method`

Template Method

- Most algorithm steps are common across file types
- The *extract...* and *parse...* methods are particular to the document format
- We **template them** - provide an abstract implementation that subclasses override

Template Method

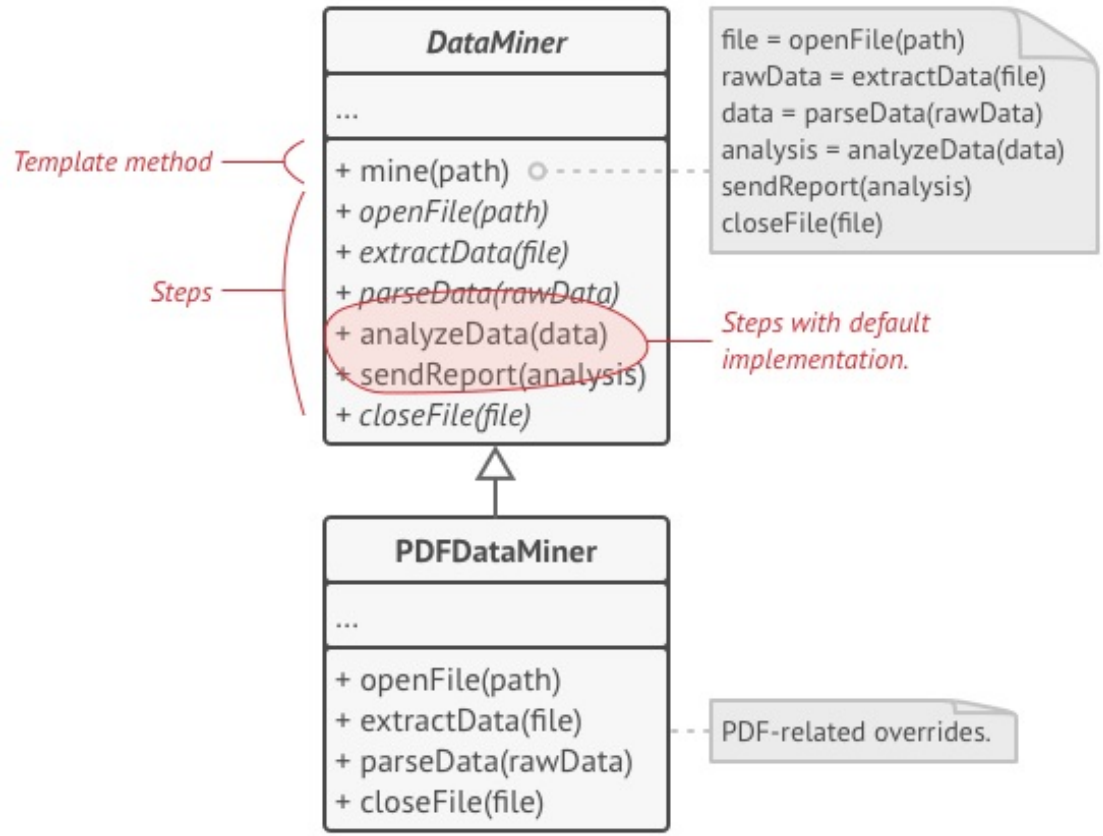

Figure: from `https://refactoring.guru/design-patterns/template-method`

Template Method

When to use

- Implement invariant parts of an algorithm
- Separate differences between algorithms into new classes, and avoid code duplication
- Allow extensions to your code

Template Method

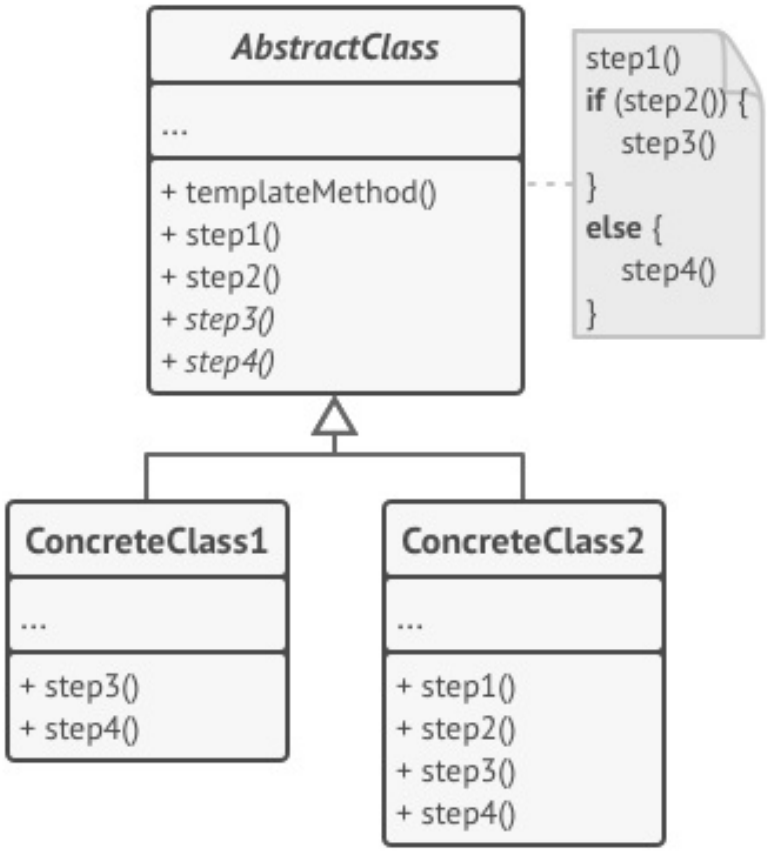

Figure: General case (from `https://refactoring.guru/design-patterns/template-method`)

Template Method

Consequences

- Templates methods are fundamental for code reuse
- Lead to the Hollywood principle[3]

Similarities

- With *Factory method* pattern
- With *Strategy* pattern

[3] "Don't call us, we'll call you" ☺

Template Method

Template Method source code

git:/src/ubb/dp/behavioural/template

Visitor

Visitor pattern

Represent an operation to be performed on the elements of a structure. Define a new operation without changing the elements it operates on.

Visitor

Motivating example[4]

- Develop an app that works with geographical data
- You create a large graph of available objectives
- At some point, you need the graph saved to XML

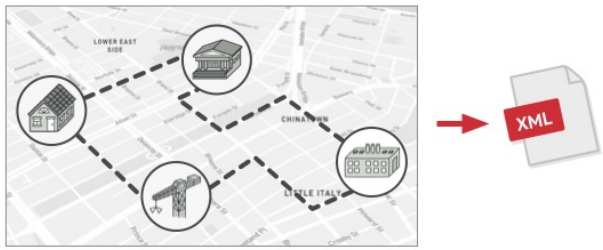

Figure: General case (from `https://refactoring.guru/design-patterns/visitor`)

[4] from `https://refactoring.guru/design-patterns/visitor`

Visitor

Motivating example - add a *saveToXML()* method to all node types

- Nodes model business entities, and have nothing to do with XML
- Limited access to node classes
- What is you also need *saveToJSON()*?

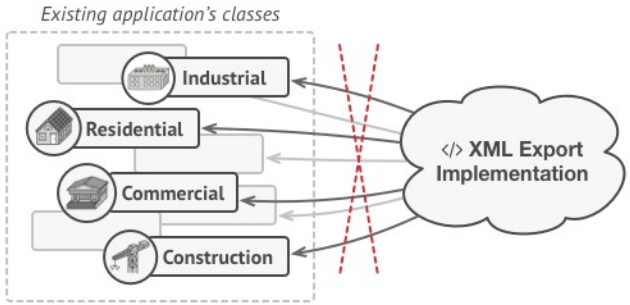

Figure: General case (from `https://refactoring.guru/design-patterns/visitor`)

Visitor

Proposed solution

- Group the functionality into a new, *visitor* class
- Separate the data structure from the algorithm processing it
- Results in two hierarchies - one for the elements being operated on, and one for the visitors
- Elements only need a new operation to *accept(Visitor v)* a visitor, regardless of its type

When to use

- Perform the same operation on many different types
- Operations to carry out change more frequently than the structure elements

Visitor

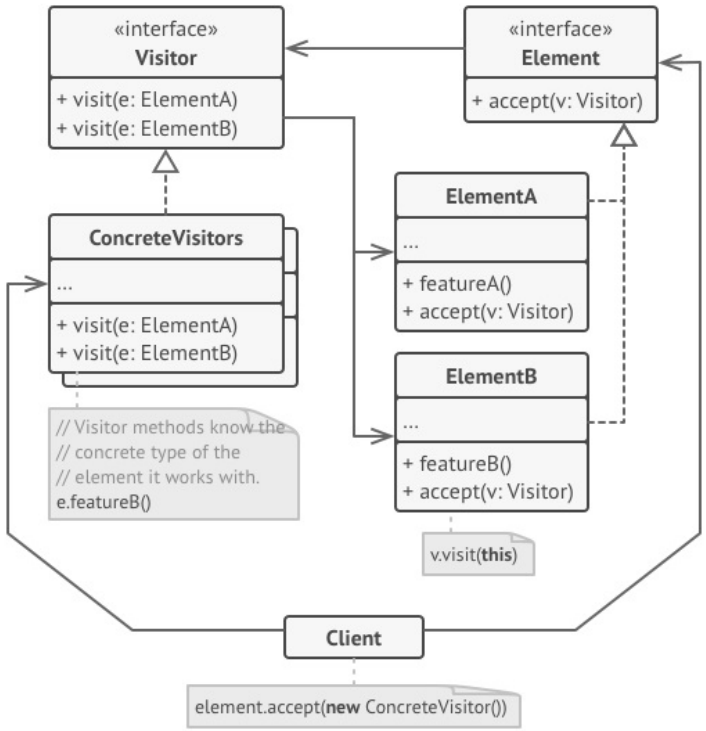

Figure: General case (from `https://refactoring.guru/design-patterns/visitor`)

Visitor

Consequences

- Related behaviour is grouped in the visitor class (single responsibility principle)
- Implementing new behaviour is easy (open for extension)
- Element classes are not polluted with operations
- Adding new elements is difficult, as we have to update the visitors
- Elements must expose enough info to allow visitors to do their job without breaking encapsulation

Visitor

Related patterns

- In many cases visitors are used to apply an operation over a *composite* structure
- An iterator (*internal* or *external*) can be used to visit each element in the structure

Visitor

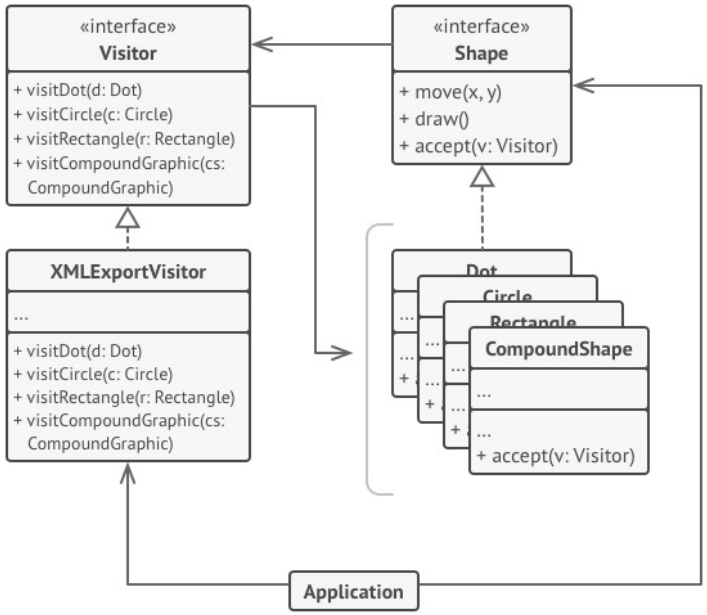

Figure: `https://refactoring.guru/design-patterns/visitor`

Visitor source code

git:/src/ubb/dp/behavioural/visitor

Visitor

Exercise

Implement a visitor-based price and power calculation for the computer assembled in the *Composite* example

Discussion of Behavioural Patterns

Encapsulate Variation - describe aspects that are likely to change

- *Strategy* encapsulates an algorithm
- *State* encapsulates a behaviour that depends on a small number of states
- *Mediator* encapsulates the communication between other objects
- *Iterator* encapsulates the traversal of a data structure

Discussion of Behavioural Patterns

Objects as arguments

- **Visitor** receives as argument the currently visited object
- **Command** encapsulates an operation to carry out in the future as an object
- **Memento** uses an object to "remember" an object state

Discussion of Behavioural Patterns

Decoupling Senders and Receivers

- *Observer* distributes communication between a *Subject* and its *Observers*
- *Mediator* encapsulates the communication - less flexible, tightly encapsulated, easier to comprehend