1 Introduction

1.1 Pattern Definition

- Descriptions of successful engineering stories
- Address recurring problem
- Descripe generic solution that worked
- Tell about the forces of the problem (why is the problem hard)
- Tell about the engineering trade-offs to take (Benefits / Liabilities)
- Solution (Implementation)

1.2 Type of Patterns

- Architecture Patterns (Waiting Room)
- Software Patterns
  - Design Pattern (Elements of Reusable Object-Oriented Sofware)
  - Pattern-oriented Software Architecture (POSA)
- Organizational Patterns
- Learning and Teaching Patterns
- Documentation Patterns

1.3 Pattern Formats

1.3.1 POSA

- Pattern name
- Intent
- Problem
- Solution
- Benefits / Liabilities

1.3.2 Fault Tolerance

- Name
- Intent
- Solution
- Benefits / Liabilities

1.3.3 MAPI

- Name
- Intent
- Consequences

1.3.4 Game Programming Patterns

- Name
- Problem
- Engineering Story that worked
- Benefits / Liabilities
- Solution

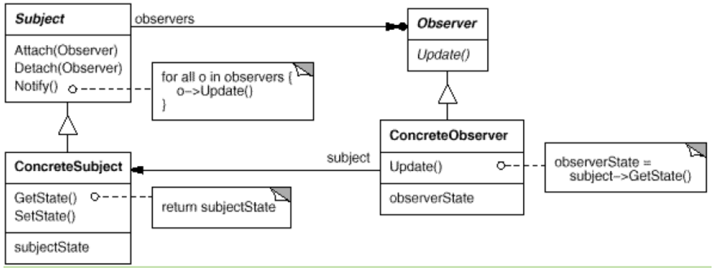
1.4 What are Patterns not?

- Silver bullet
- Novices Tool
- Ready Made Components
- Means to turn off your brain

2 GoF Patterns

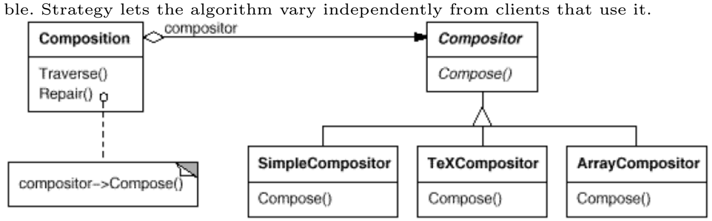
2.1 Observer

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



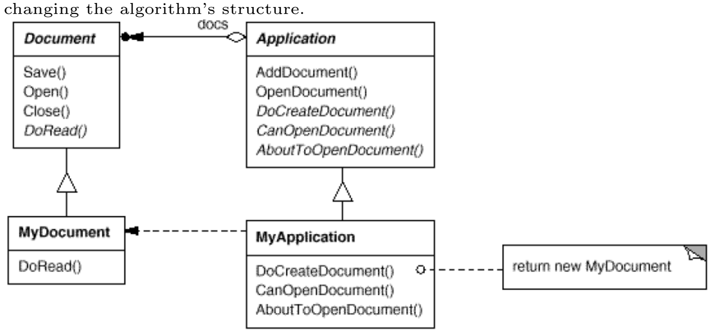
2.2 Strategy

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



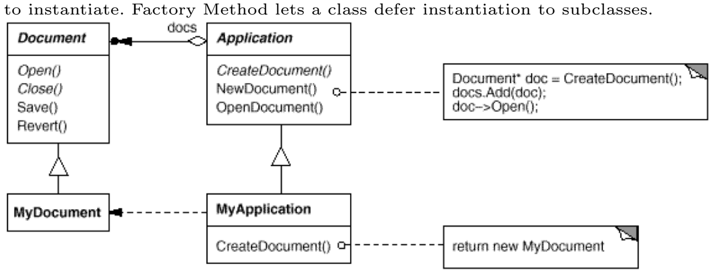
2.3 Template Method

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.



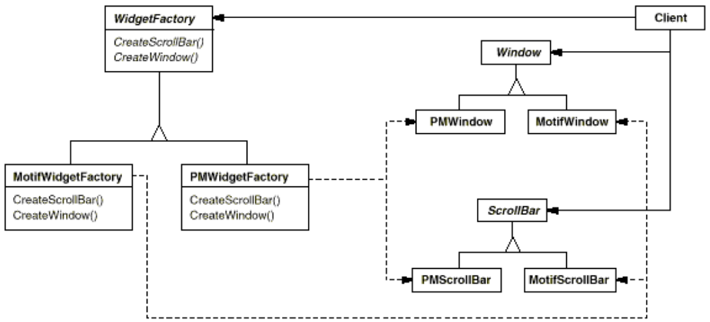
2.4 Factory Method

Define an interface for creating an object, but let the subclass decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.



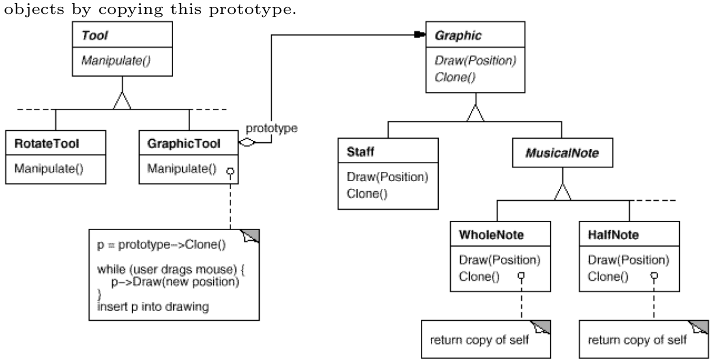
2.5 Abstract Factory

Provide an interface for creating families of related or dependant objects without specifying their concrete classes.



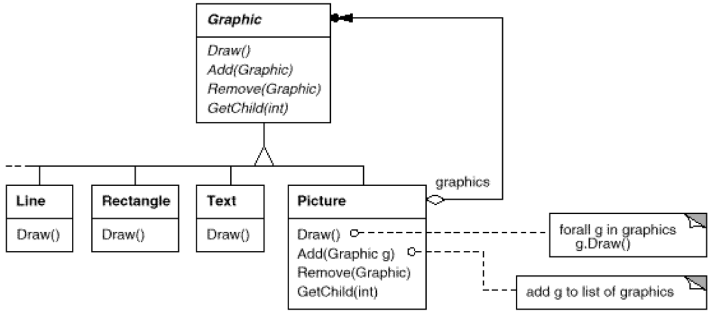
2.6 Prototype

Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.



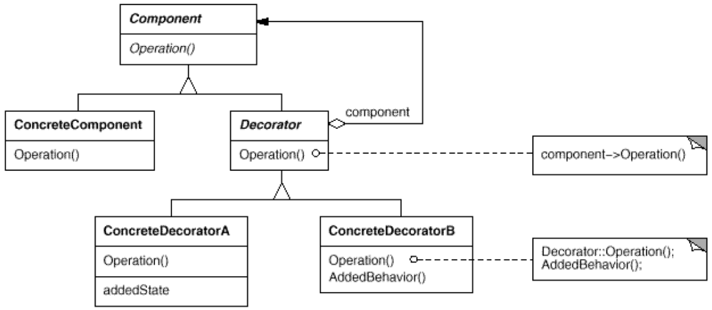
2.7 Composite

Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.



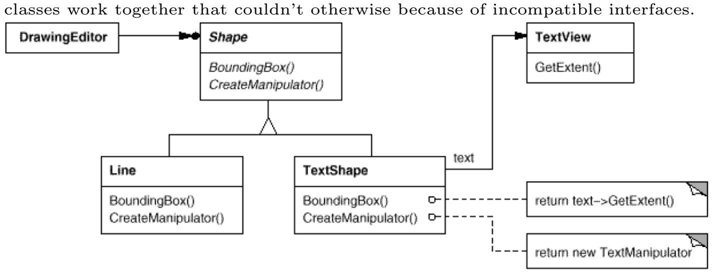
2.8 Decorator

Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.



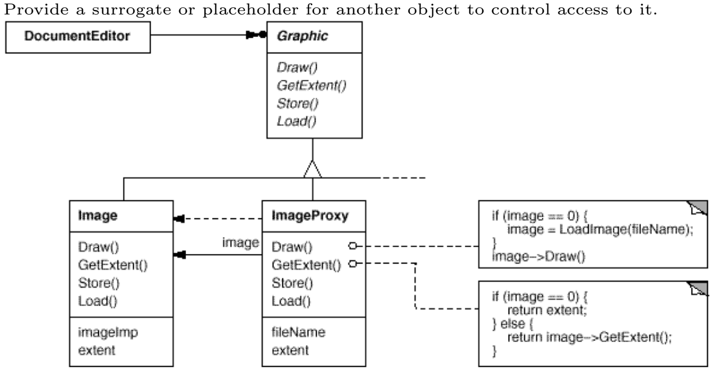
2.9 Adapter

Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.



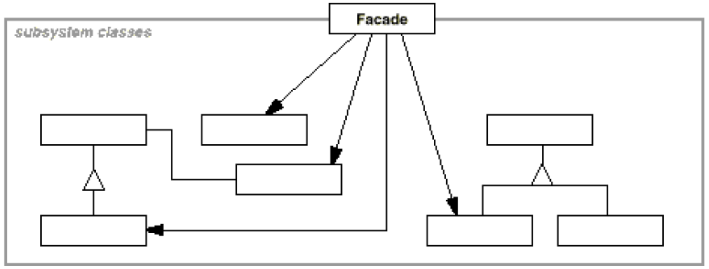
2.10 Proxy

Provide a surrogate or placeholder for another object to control access to it.



2.11 Facade

Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.



2.12 Mediator

2.12.1 Problem

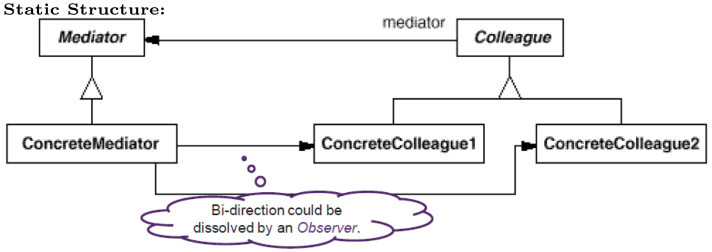
- Object Structures may result in many connections between objects
- In the worst case, every object ends up knowing about every other

Intent:

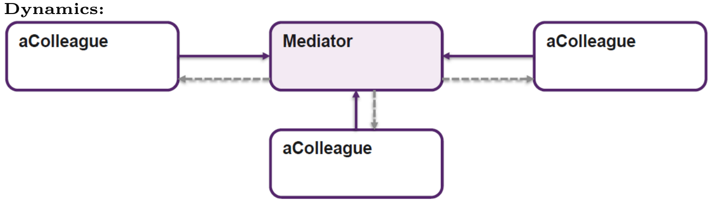
- How can strong coupling between multiple objects be avoided and communication simplified?

2.12.2 Solution

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 lets you vary their interaction independently.
Mediator: Encapsulates how a set of objects interact
Colleaues: Refer to Mediator; this promotes loose coupling
Static Structure:



Dynamics:



2.12.3 Implementation

- Mediator as an Observer
- Colleagues act as Subject

Known Uses:

- Message Bus Systems
- Redux Dispatcher

2.12.4 Summary

Benefits:

- Colleague classes may become more reusable, low coupling
- Centralizes control of communication between objects
- Encapsulates protocols

Liabilities:

- Adds complexity
- Single point of failure
- Limits subclassing (of mediator class)
- May result in hard maintainable monoliths

2.13 Memento

2.13.1 Problem

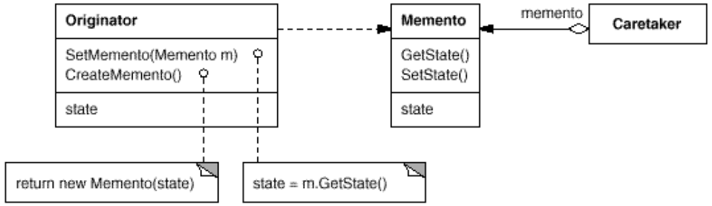
- Sometimes it's necessary to record the internal state of an object
- Objects normally encapsulate their state, making it inaccessible

Intent:

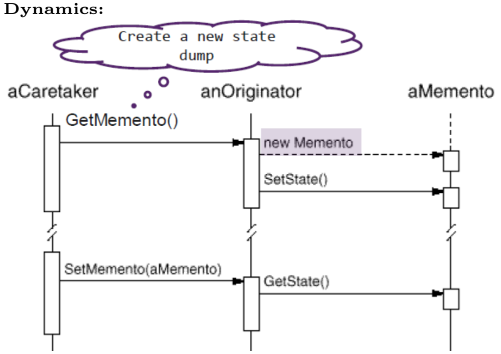
- How can the state of an object be externalized without violating its encapsulation?

2.13.2 Solution

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



Dynamics:



2.13.3 Participants

Memento

- Stores some / all the internal state of the Originator
- Allows only the originator to access its internal information

Originator

- Can create Memento objects to store its internal state at strategic points
- Can restore own state to what the Memento object dictates

Caretaker

- Stores the Memento objects
- Cannot explore / operate the contents

2.13.4 Implementation

- Originator creates memento and passes over its internal state
- Can be combined with Factory Method
- Declare Originator as *friend* of Memento, so Originator can read out its properties

2.13.5 Summary

Benefits

- Internal State of an object can be saved and restored at any time
- Encapsulation of attributes is not harmed
- State of objects can be restored later

Liabilities

- Creates a complete copy of the object every time, no diffs (memory usage)
- No direct access to saved state, it must be restored first

2.14 Command

2.14.1 Problem

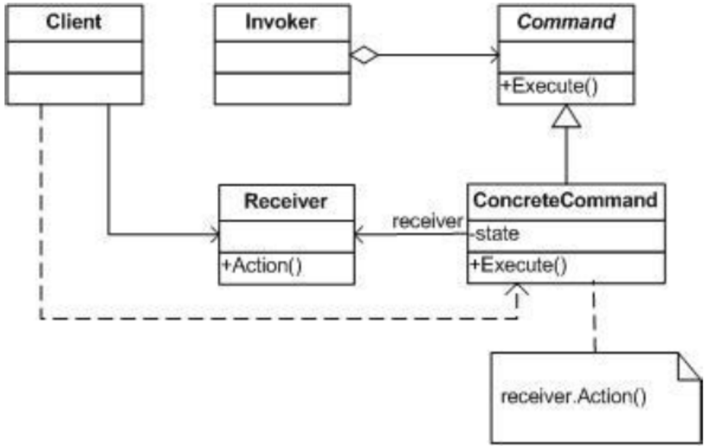
- Decouple the decision of what to execute from the decision of when to execute
- The execution needs an additional parametrization context

Intent:

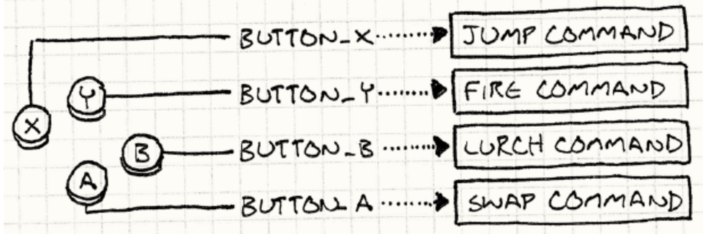
- How can commands be encapsulated, so that they can be parameterized, scheduled, logged and/or undone?

2.14.2 Solution

Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operation.



Dynamics:



2.14.3 Summary

Benefits:

- The same command can be activated from different objects
- New commands can be introduced quickly and easily
- Command objects can be saved in a command history
- Provides inversion of control, encourages decoupling in both time and space

Liabilities:

- Large designs with many commands can introduce many small command classes mauling the design

2.15 Command Processor

2.15.1 Problem

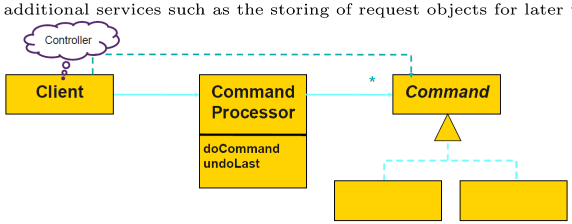
- Common UI applications support do and multiple undo steps
- Steps forward and backward are accessible in a history

Intent:

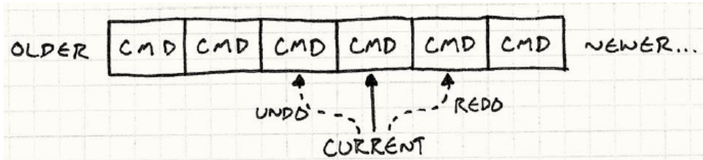
- How could we manage command objects, so the execution is seperated from the request and the execution can be undone later?

2.15.2 Solution

Separate the request for a service from its execution. A command processor component manages requests as separate objects, schedules their execution, and provides additional services such as the storing of request objects for later undo.



Dynamics:



2.15.3 Participants

Command Processor

- A Separate processor object can handle the responsibility for multiple Command objects

Command

- A uniform interface to execute functions

Controller

- Translates requests into commands and transfers commands to Command Processor.

2.15.4 Implementation

- Command Processor contains a *Stack* which holds the command history
- Controller creates the Commands and passes them over to Command Processor
- Creation of Commands may be delegated to a *Simple Factory*

2.15.5 Summary

Benefits:

- Flexibility
- Allows addition of services related to command execution
- Enhances testability

Liabilities:

- Efficiency loss due additional indirection

2.16 Visitor

2.16.1 Problem

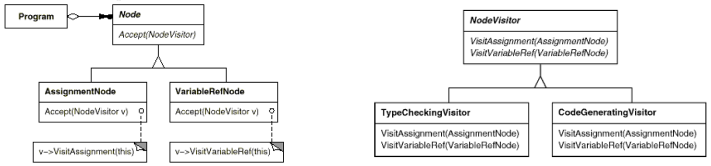
- Operations on specific classes needs to be changed/added without needing to modify these classes
- Different algorithms needed to process an object tree

Intent:

- How can the behaviour on individual elements of a data structure be changed/replaced whout changing the elements?

2.16.2 Solution

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.



2.16.3 Implementation

- 2 Class Hierarchies (Elements / Visitors)
- Visitors iterate though object hierarchy
- Solves Double-Dispatch problem of single dispatched programming languages

Patterns that combine naturally with Vistor:

- Composite
- Interpreter
- Chain of Responsibility

2.16.4 Summary

Benefits:

- Visitor makes adding new operatios easy
- Separates related operations from unrelated ones

Liabilities:

- Adding new node classes is hard
- Visiting sequence fix defined within nodes
- Visitor breaks logic apart

3 Beyond GoF

3.1 External Iterator

3.1.1 Problem

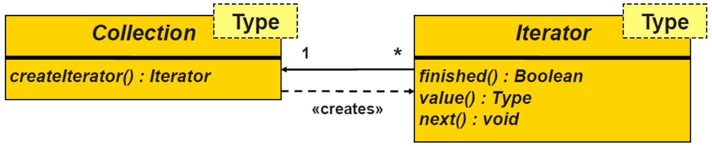
- Iteration through a collection depends on the target implementation
- Separate logic of iteration into an object to allow multiple iteration strategies

Intent:

- How can strong coupling between iteration and collection be avoided, generalized and provided in a collection-optimized manner?

3.1.2 Solution

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



Elementary operations of an Iterator's behaviour:

- Initializing an iteration *new ArrayList().iterator();*
- Checking a completion condition *it.hasNext();*
- Accessing a current target value *var x = it.next();*
- Moving to the next target value *it.next();*

3.1.3 Summary

Benefits:

- Provides a single interface to loop though any kind of collection

Liabilities:

- Multiple iterators may loop through a collection at the same time
- Life-Cycle Management of iterator objects
- Close coupling between Iterator and Collection class
- Indexing might be more intuitive for programmers

3.2 Enumeration Method

3.2.1 Problem:

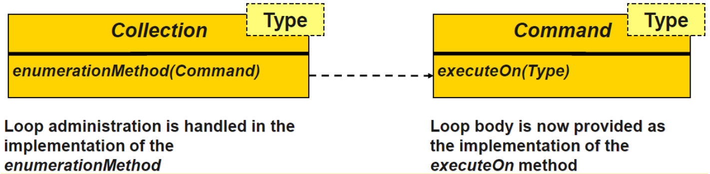
- Iteration management is performed by the collection's user
- Avoid state management between collection and iteration

Intent:

- How can a collection be iterated considering the collection state and furthermore state management be reduced?

3.2.2 Solution

Support encapsulated iteration over a collection by placing responsibility for iteration in a method on the collection. The method takes a Command object that is applied to the elements of the collection.



3.2.3 Summary

Programming languages already implement Enumeration Method as their loop construct. (e.g. *.forEach()*)

Benefits:

- Client is not responsible for loop housekeeping details
- Synchronization can be provided at the level of the whole traversal rather than for each element access

Liabilities:

- Functional approach, more complex syntax needed
- Often considered too abstract for programmers

3.3 Batch Method

3.3.1 Problem

- Collection and client (iterator user) are not on the same machine
- Operation invocations are no longer trivial

Intent:

- How can a collection be iterated over multiple tiers without spending far more time in communication than in computation?

3.3.2 Solution

Group multiple collection accesses together to reduce the cost of multiple individual accesses in a distributed environment.

- Define a data structure which groups interface calls on client side
- Provide an interface on servant to access groups of elements at once

3.4 Objects for State

3.4.1 Problem

- Object's behaviour depends on its state, and it must change its behaviour at run-time
- Operations have large, multipart conditional statements (Flags) that depend on the state

Intent:

- How can an object act according to its state without multipart conditional statements?

3.4.2 Solution

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