

4.4 Design Patterns

- Gamma, Helm, Johnson, Vlissides: Design Patterns, Elements of Reusable Object-Oriented Software, Addison Wesley, 1995. “gang of four”
- recurring patterns of collaborating objects
- goals: flexibility, maintainability, communication, reuse
- each pattern emphasizes certain aspects
flexibility vs. overhead, # objects
- alternative approaches and combinations possible
- task: which (combination of) pattern(s) is best
- class-based ↔ object-based patterns
- inheritance ↔ delegation

Classification of Design Patterns

Purpose:

Creational Patterns deal with object creation

Abstract Factory, Builder, Factory Method, Prototype, Singleton

Structural Patterns composition of classes or objects

Adapter, Bridge, Composite, Decorator, Facade, Flyweight, Proxy

Behavioral Patterns interaction of classes or objects

Command, Interpreter, Iterator, Observer, State, Strategy, Visitor

Scope:

Class static relationships between classes (inheritance)

Object dynamic relationships between objects

Standard Presentation:

Intent

Motivation

Applicability

Structure

Participants

Collaborations

Consequences

Implementation

Sample Code

Known Uses

Related Patterns

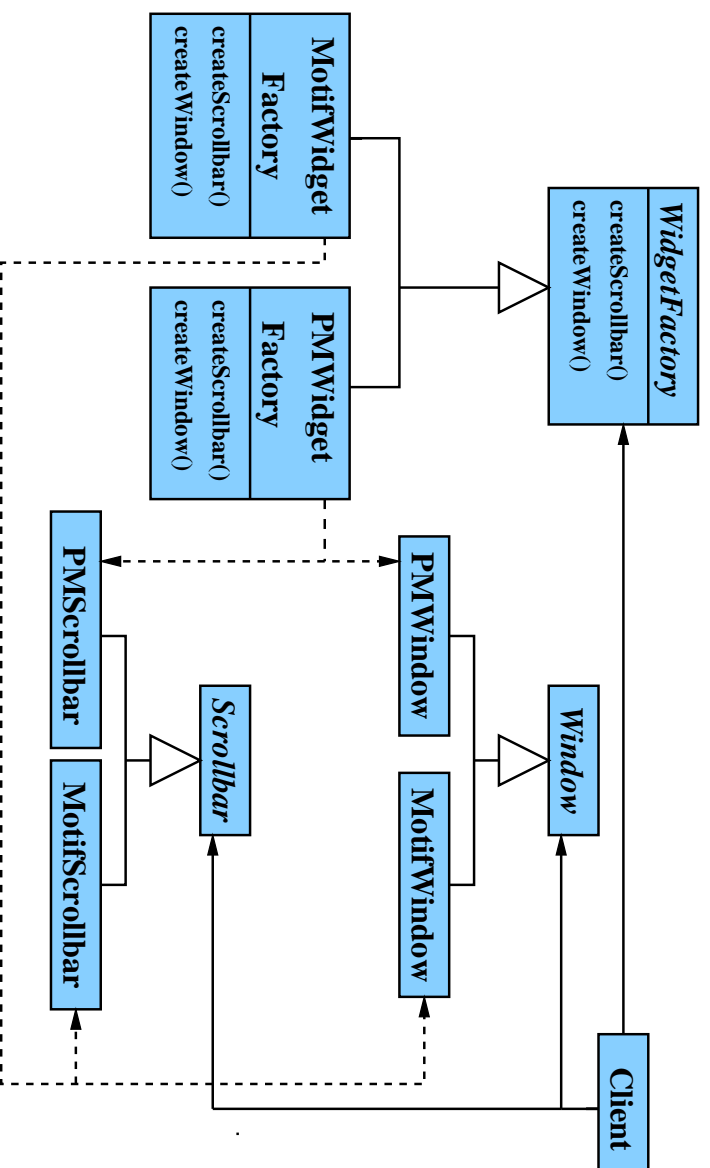
4.4.1 Creational Patterns

Pattern: **Abstract Factory (Kit)** **object, creational**

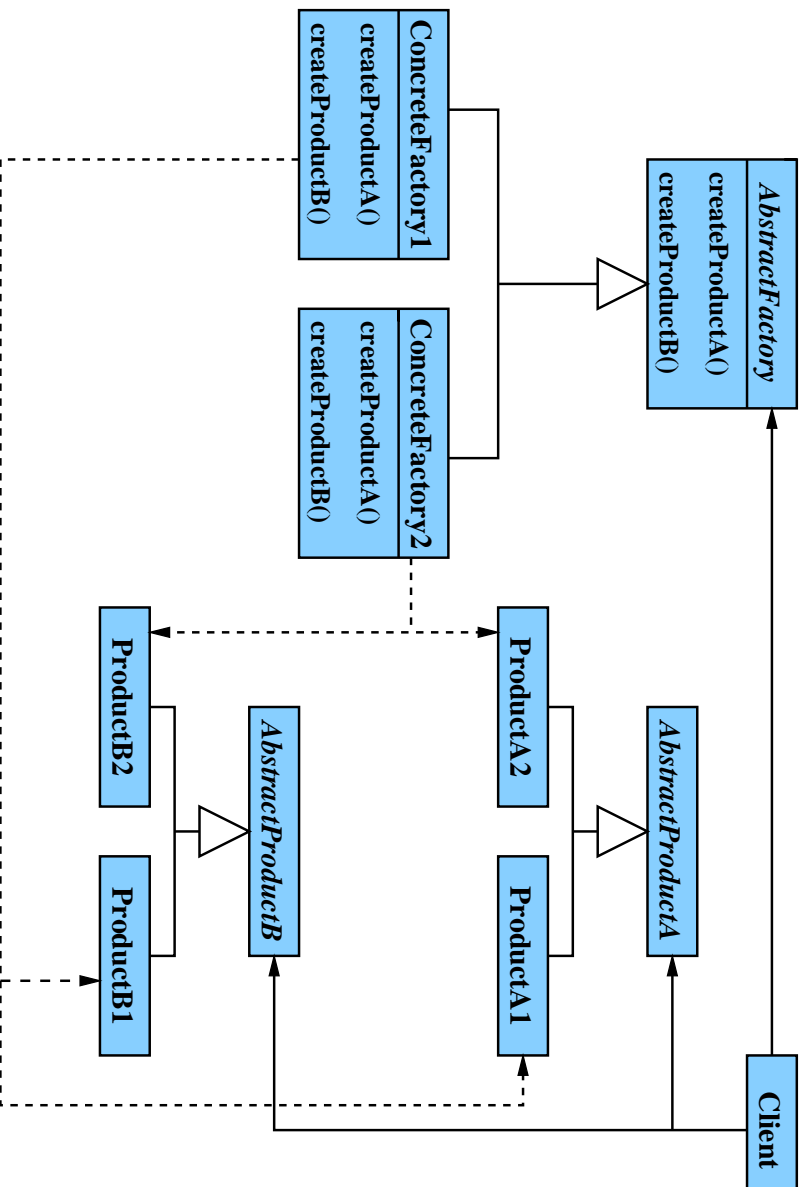
Intent: Provide an interface for creating families of related or dependent objects without specifying their concrete classes

Motivation

- user interface toolkit supporting multiple look-and-feel standards
e.g., Motif, Presentation Manager



Structure



Applicability

- independent of how products are created, composed, and represented
- configuration with one of multiple families of products
- related products must be used together
- reveal only interface, not implementation

Consequences

- product class names do not appear in code
- exchange of product families easy
- requires consistency among products

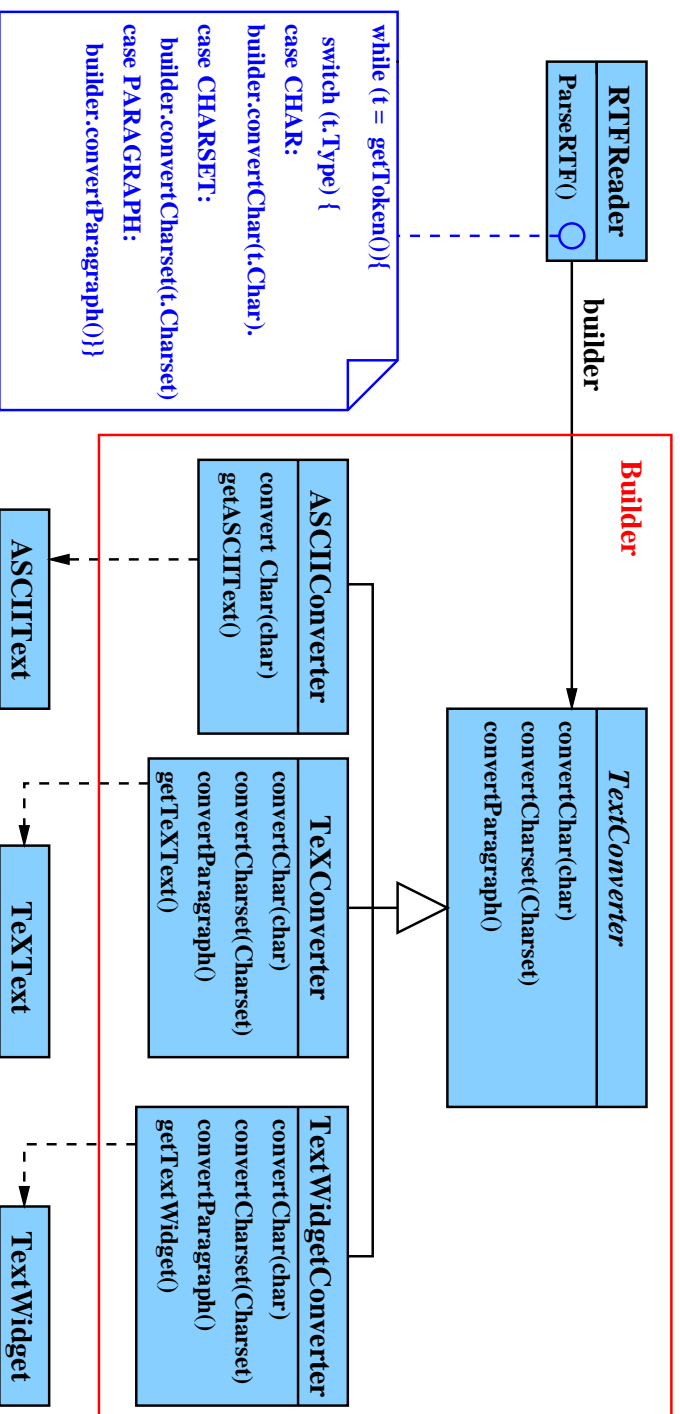
Pattern: **Builder**

object, creational

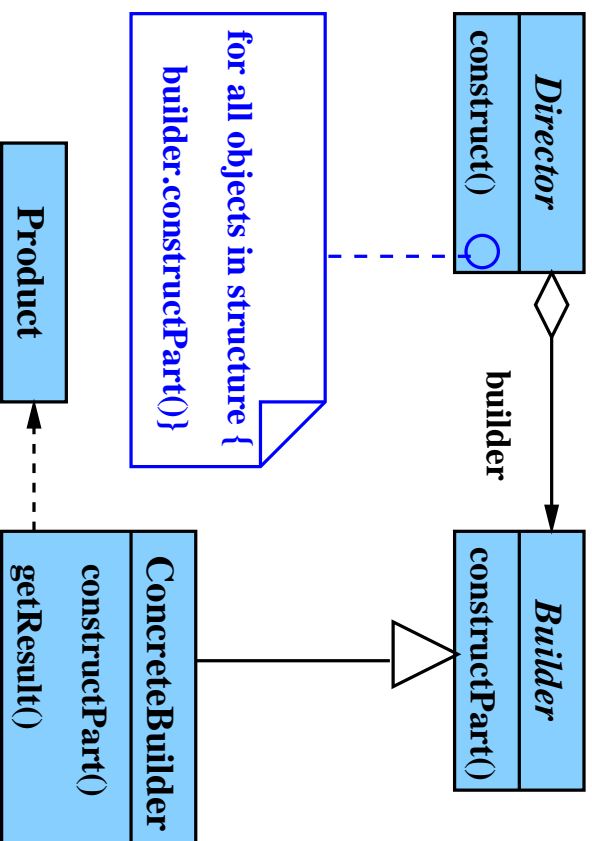
Intent: Separate the construction of a complex object from its representation so that the same construction process can create different representations.

Motivation

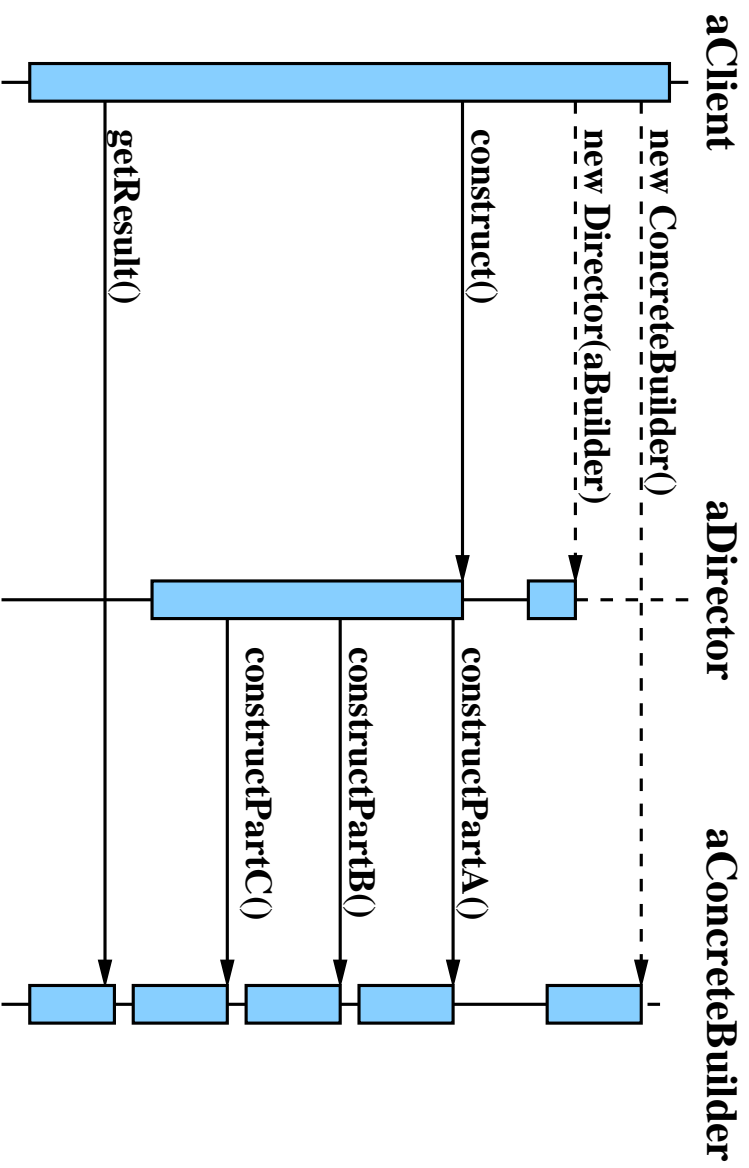
- read RTF and translate in different exchangeable formats



Structure



Interaction Diagram for Builder



Consequences

- reusable for other directors (e.g. XMLReader)

Difference to Abstract Factory

- Builder assembles a product step-by-step (parameterized over assembly steps)
- Abstract Factory returns complete product

Pattern: **Factory Method**

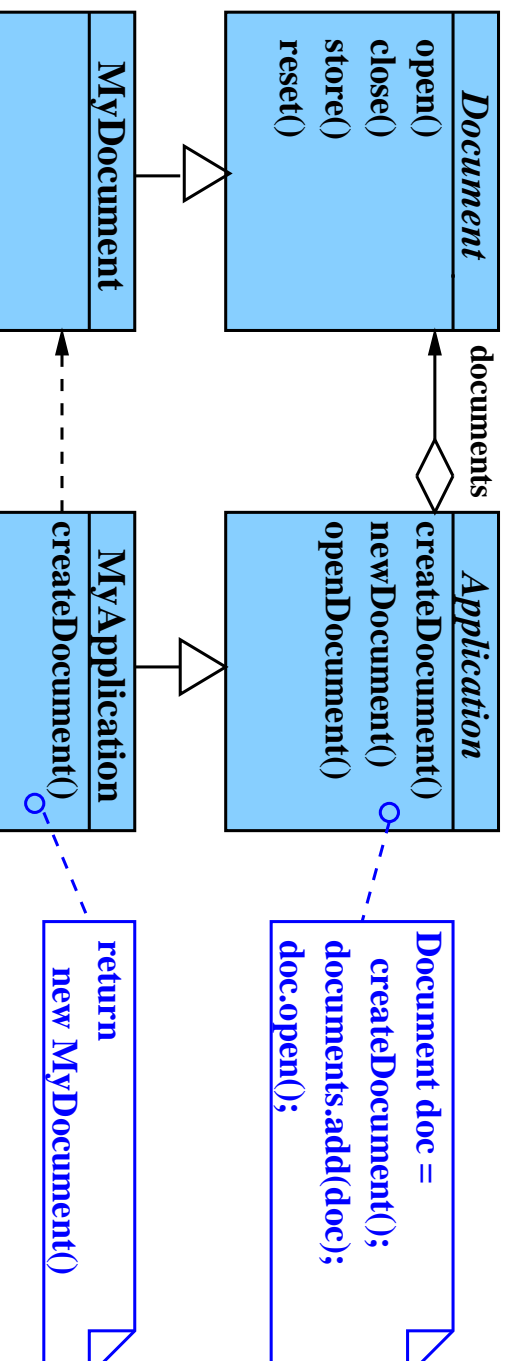
class, creational

Intent

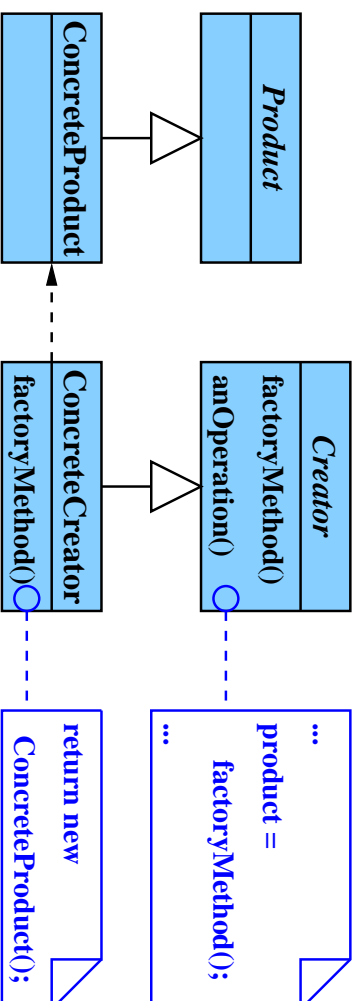
- class must create objects of unknown class (framework)
- interface with operations to create objects
- subclasses determine the class of the created object

Motivation

- framework for applications that can display many (different) documents simultaneously
- abstract classes `Application` and `Document`
- `Application` must create documents of unknown type



Structure



Applicability: class must create objects of unknown class

Consequences

- framework independent of concrete application
- creator class may be concrete
- increases class hierarchy
- default creational pattern; replace by more flexible one if required

Related Patterns: Abstract Factory

Pattern: **Prototype**

object, creational

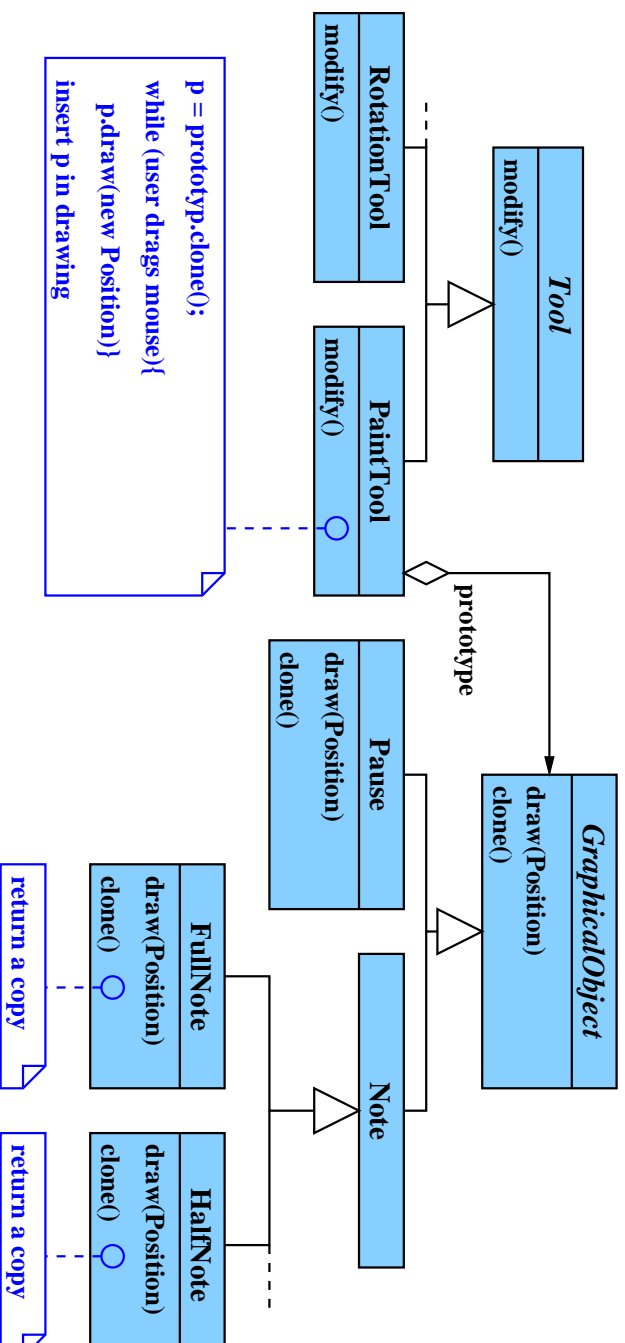
Intent

- create new objects by copying a prototypical object

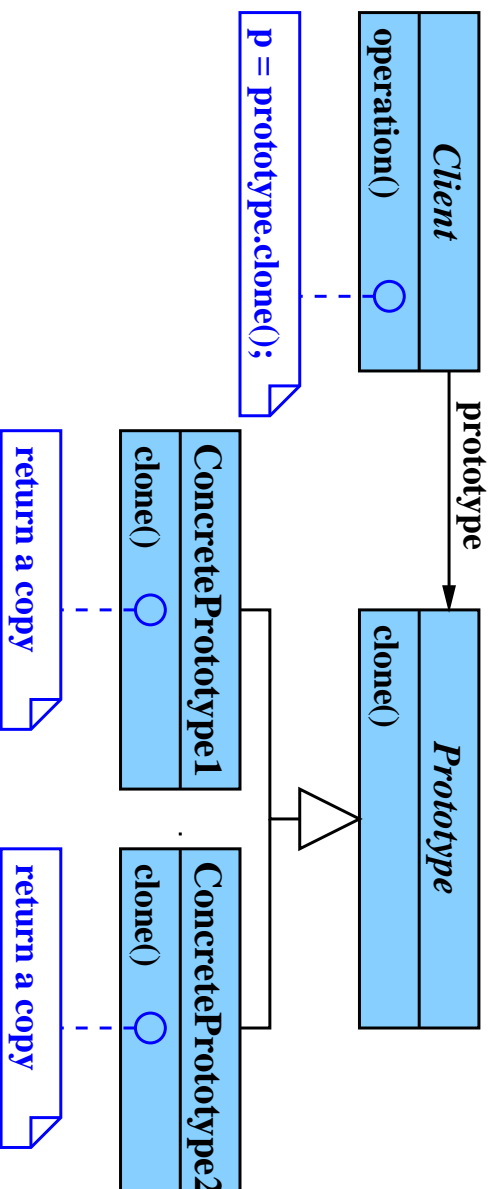
Motivation

- specialize a framework for graph editors to a musical score editor
- class `PaintTool` has to create specialized graphical objects (notes, pauses, keys, ...) two choices
 1. one subclass of `PaintTool` for each kind of musical object
 2. prototype graphical objects for each kind of musical object; `PaintTool` clones the prototype at each use

Motivation (cont'd)



Structure



Applicability

- 1) system must be independent of composition, creation, and representation of its products and
 - 2a) objects that must be created are specified at runtime; or
 - 2b) to avoid parallel class hierarchies Product and Factory; or
 - 2c) instances of a class admit only few different state combinations

Consequences

- similar to Abstract Factory and Builder
- but products may be added and removed at runtime
- fewer classes due to cloning instead of instantiation
- construction of new objects by composition (e.g. user-defined glyphs)
- dynamic configuration
- clone() can be hard to implement (deep copy vs. shallow copy, sharing, cycles)

Related Patterns

- Abstract Factory

Pattern: **Singleton**

object, creational

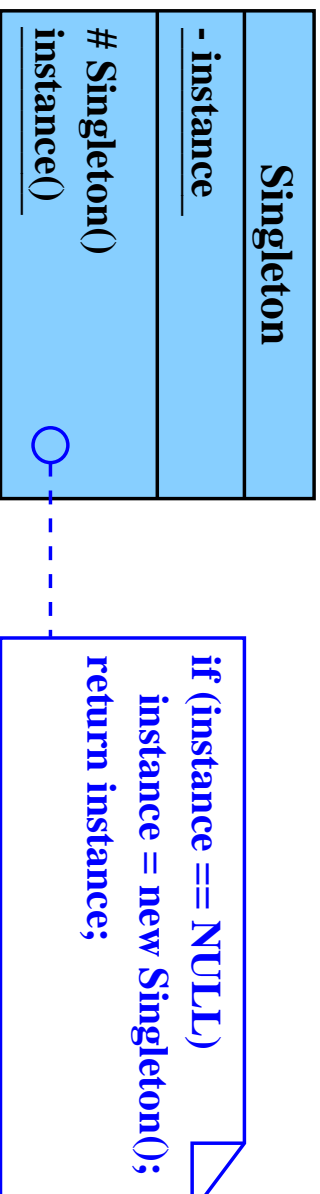
Intent

- class with exactly one object (global variable)
- no further objects are generated
- class provides access methods

Motivation

- to create factories and builders

Structure



Applicability

- exactly one object of a class required
- instance globally accessible

Consequences

- access control on singleton
- structured address space (compared to global variables)

4.4.2 Structural Patterns

- composition of classes and objects

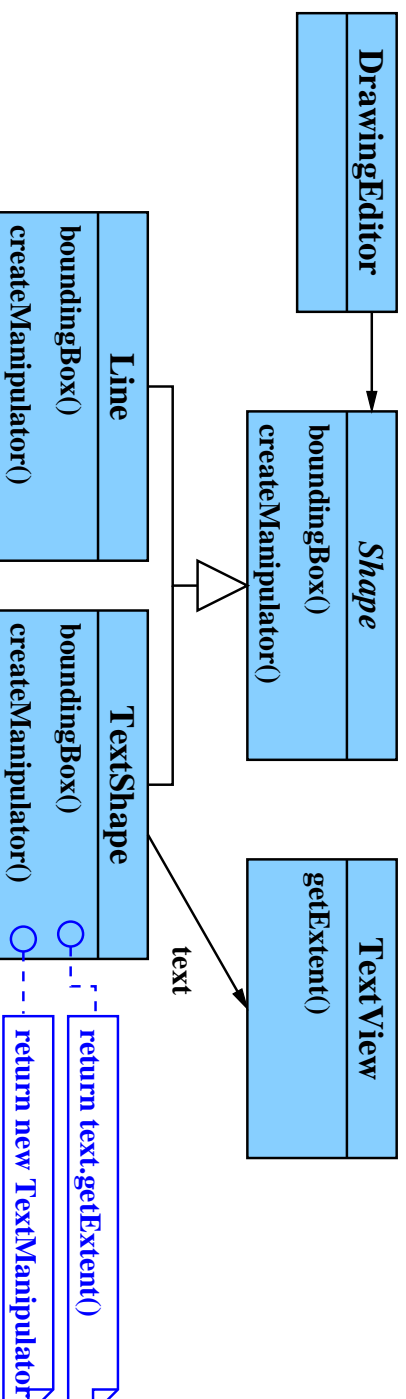
Pattern: **Adapter** (Wrapper)

class/object, structural

Intent

- Adaption of interfaces, e.g. for library classes
- add missing functionality

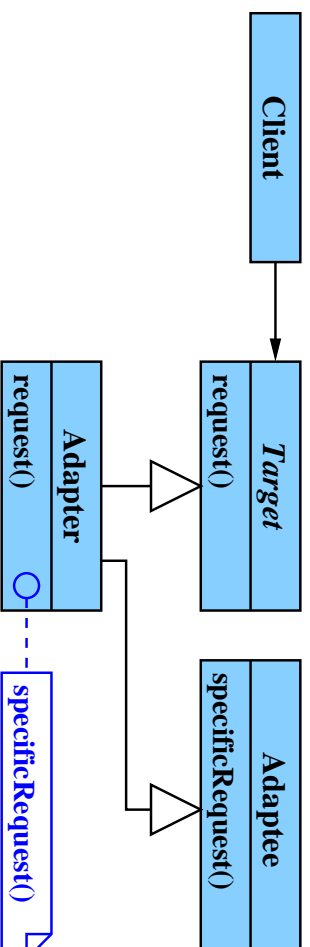
Motivation



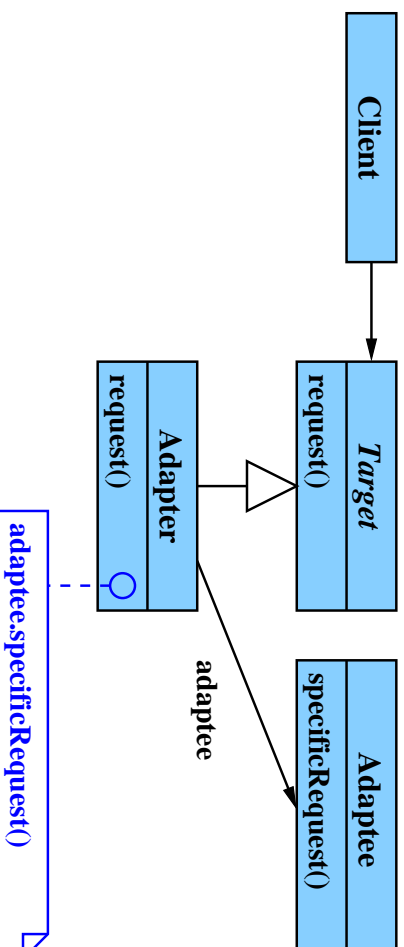
- adapter class `Text` fits `TextDisplay` to interface of class `GraphicalObject`
- similarly: adaption to database interfaces

Structure

1. class adapter



2. object adapter



Applicability: interface fitting

Consequences

1. class adapter

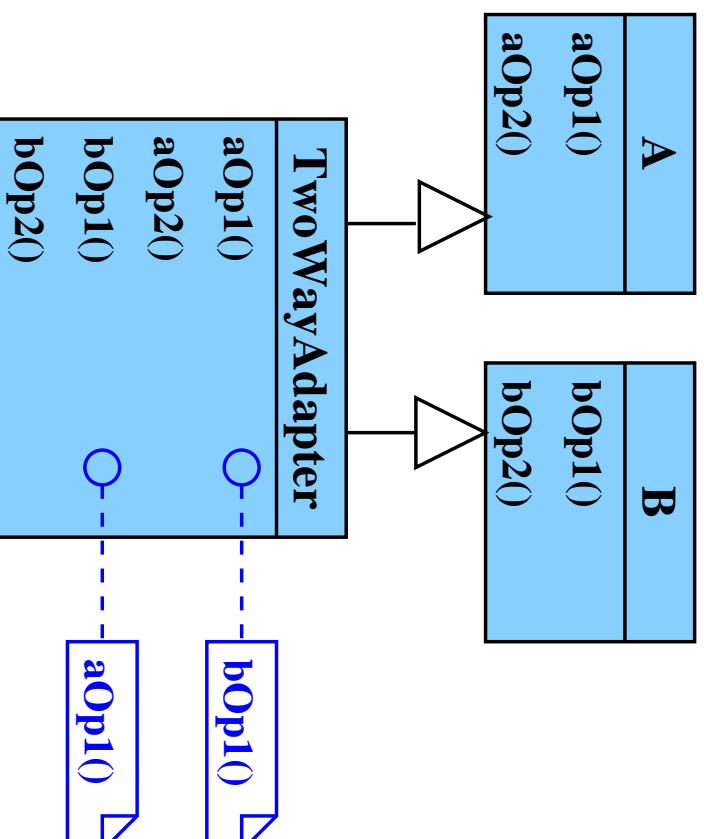
- can only adapt a class, but not its subclasses
- can modify parts of the adapted class (by overriding)
- overhead: one object

2. object adapter

- suitable for a class and all of its subclasses
- no direct overriding of parts of the adapted class
- overhead: two objects

Variant: Two-Way-Adapter

- mutual fitting of interfaces



Pattern: Bridge

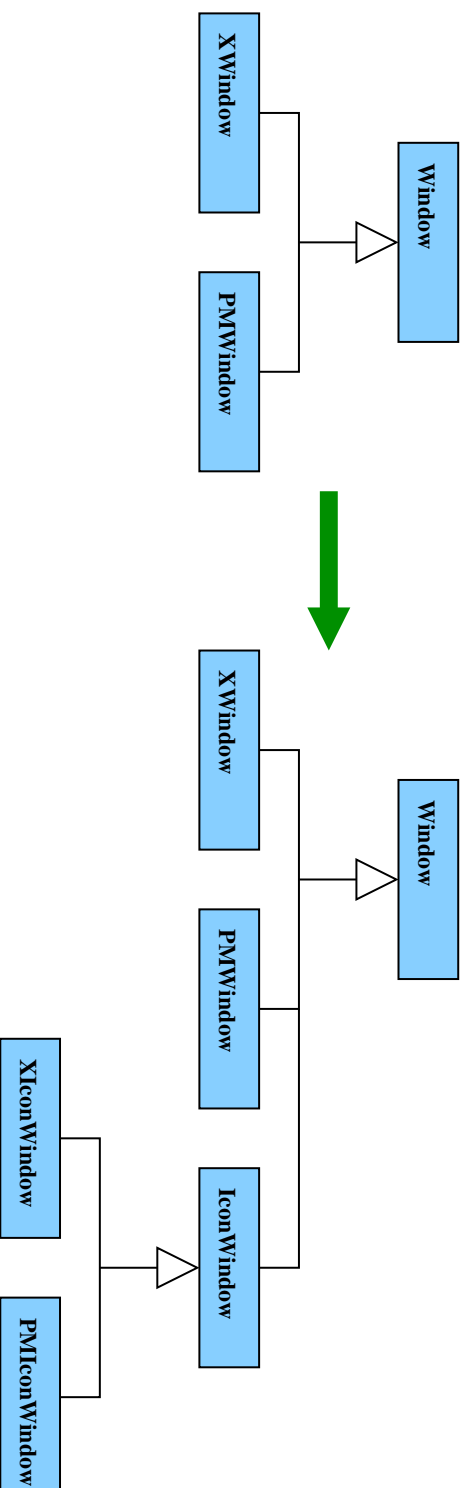
object, structural

Intent

- decouples concept (abstraction) and implementation
- → concept and implementation can evolve independently

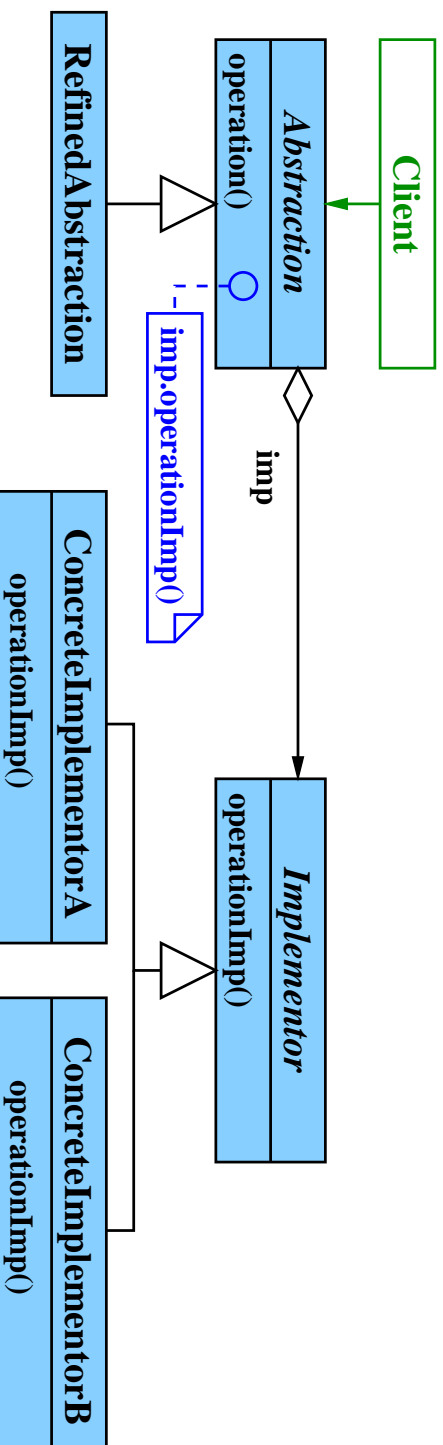
Motivation

- window hierarchy (Window, IconWindow, TransientWindow) with different implementations (X, Presentation Manager)
- naive approach leads to proliferation of classes with platform dependent code:



- solution: bridge

Structure



Applicability

- flexible connection between abstraction and implementation
- dynamic implementation exchange
- abstraction and implementation evolve independently by inheritance
- separation of interface and implementation

Pattern: **Decorator** (Wrapper)

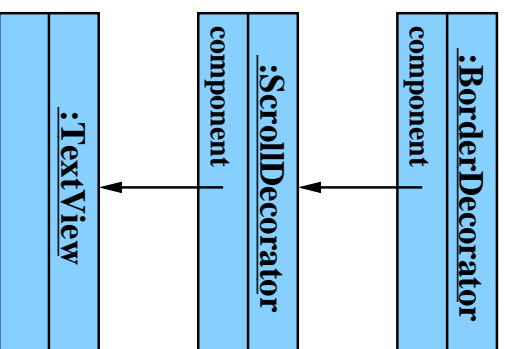
object, structural

Intent

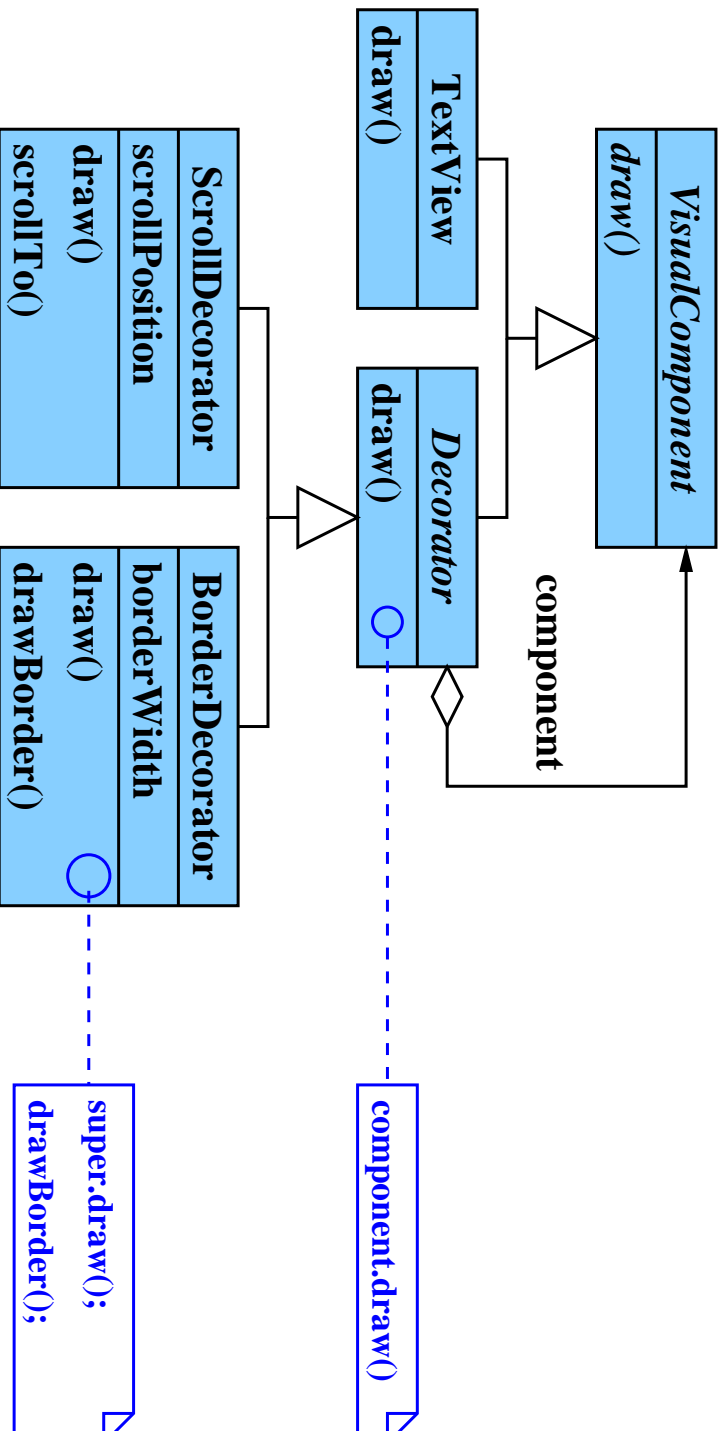
- extend object's functionality dynamically
- more flexible than inheritance

Motivation

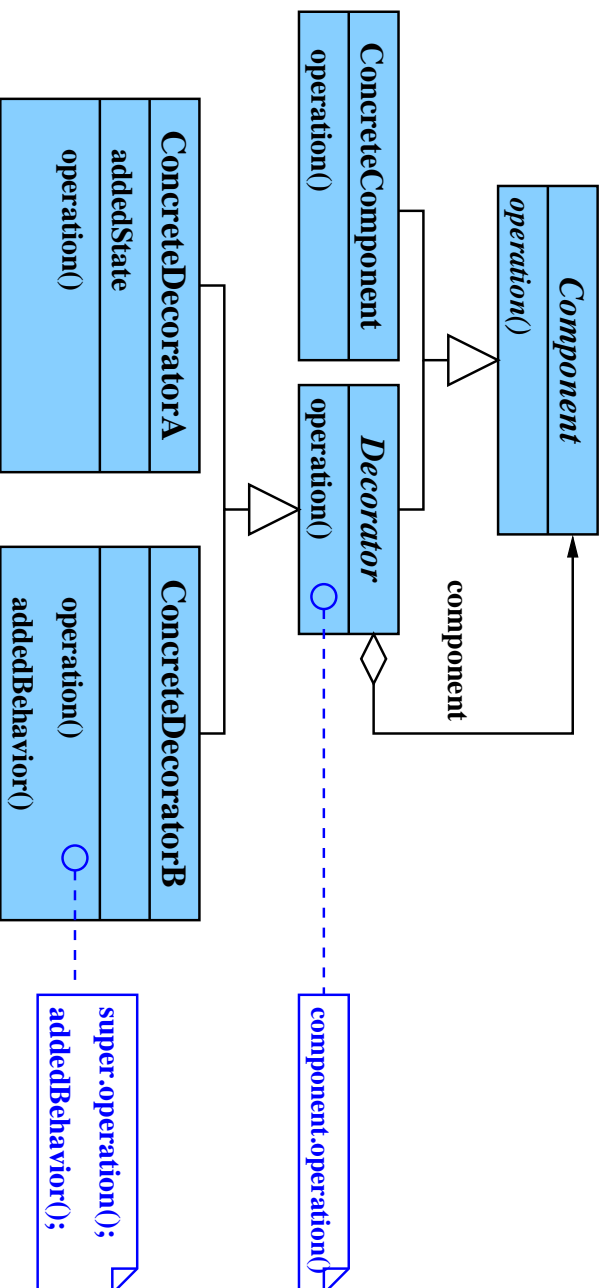
- graphical object can be equipped with border and/or scroll bar
- decorator object has same interface as the decorated object
- decorated forwards requests
- recursive decoration



Motivation (cont)



Structure



Applicability

- dynamically add responsibilities to individual objects
- for withdrawable responsibilities
- when extension by inheritance is impractical

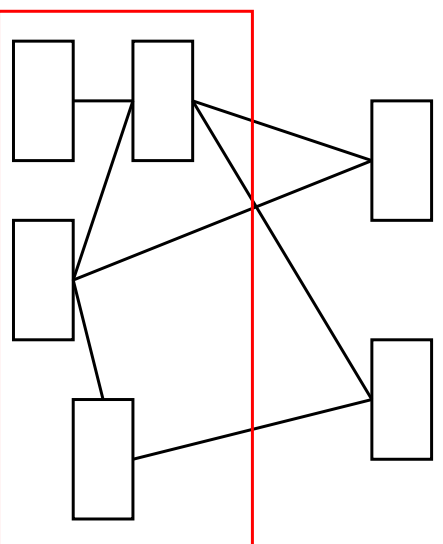
Consequences

- more flexible than inheritance
- avoids feature-laden classes high up in the hierarchy
- decorator \neq component
- lots of little objects \rightarrow hard to learn and debug

Applicability

- simple interface to complex subsystem
- many dependencies between clients and subsystem → Facade reduces coupling
- layering

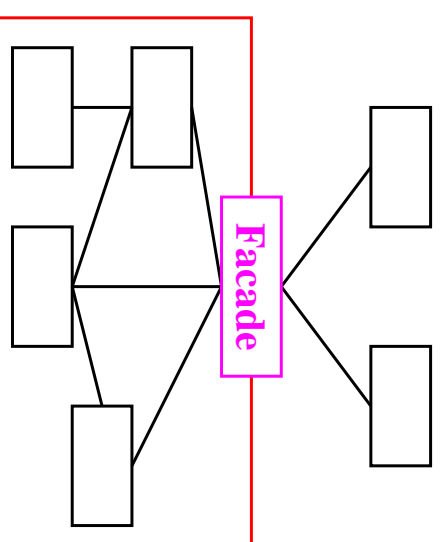
Structure



client classes



subsystem classes



Consequences

- shields clients from subsystem components
- weak coupling: improves flexibility and maintainability
- often combines operations of subsystem to new operation
- with public subsystem classes: access to each interface

Pattern: Flyweight

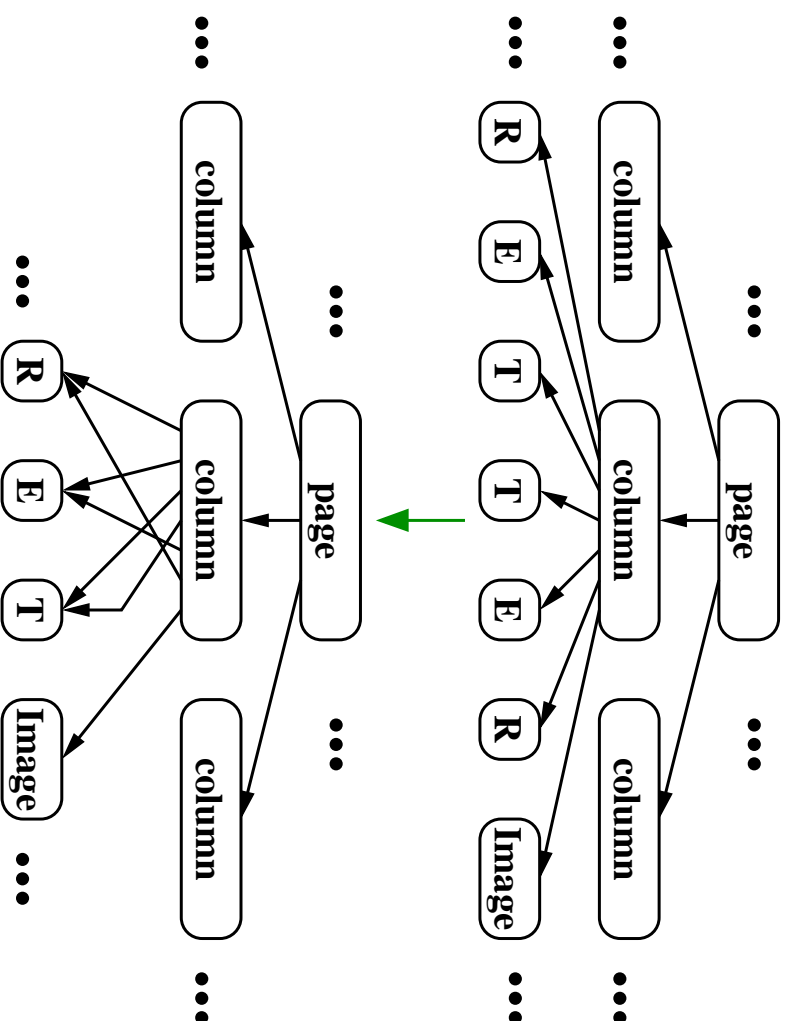
object, structural

Intent: use sharing to support large numbers of fine-grained objects efficiently

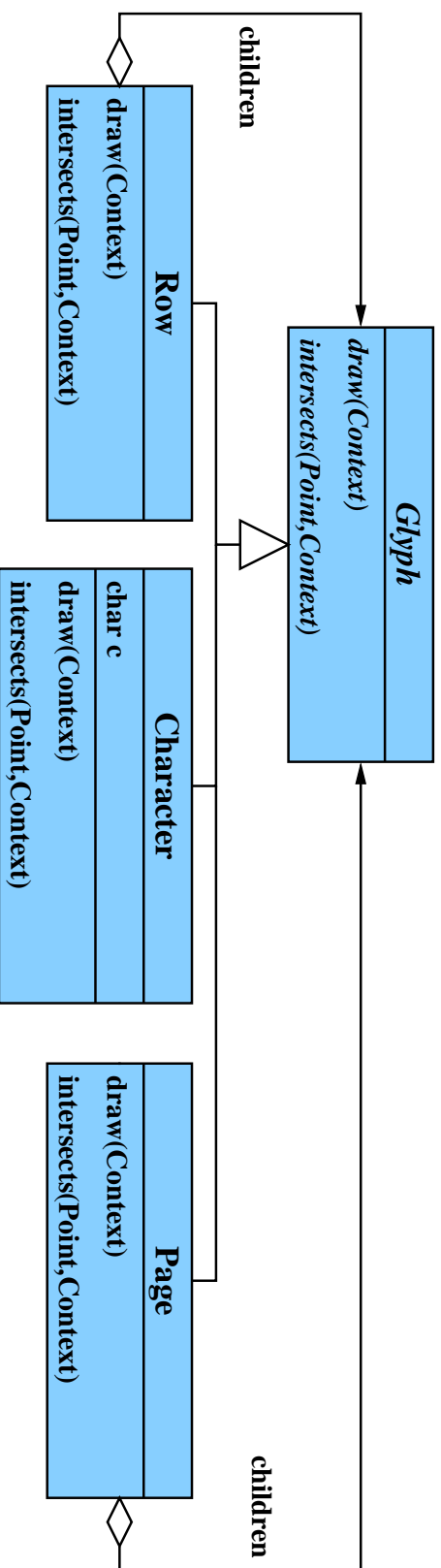
Motivation

- document editor represents images, tables, etc by objects
- but not individual characters!
- reason: high memory consumption
- objects would provide more flexibility and uniform handling of components
- approach: one Flyweight object is shared among many “equal” characters
- for increased useability
 - divide object’s state in
 - * immutable intrinsic state (letter’s identity)
 - attribute of object
 - * mutable extrinsic state (font, size)
 - parameter

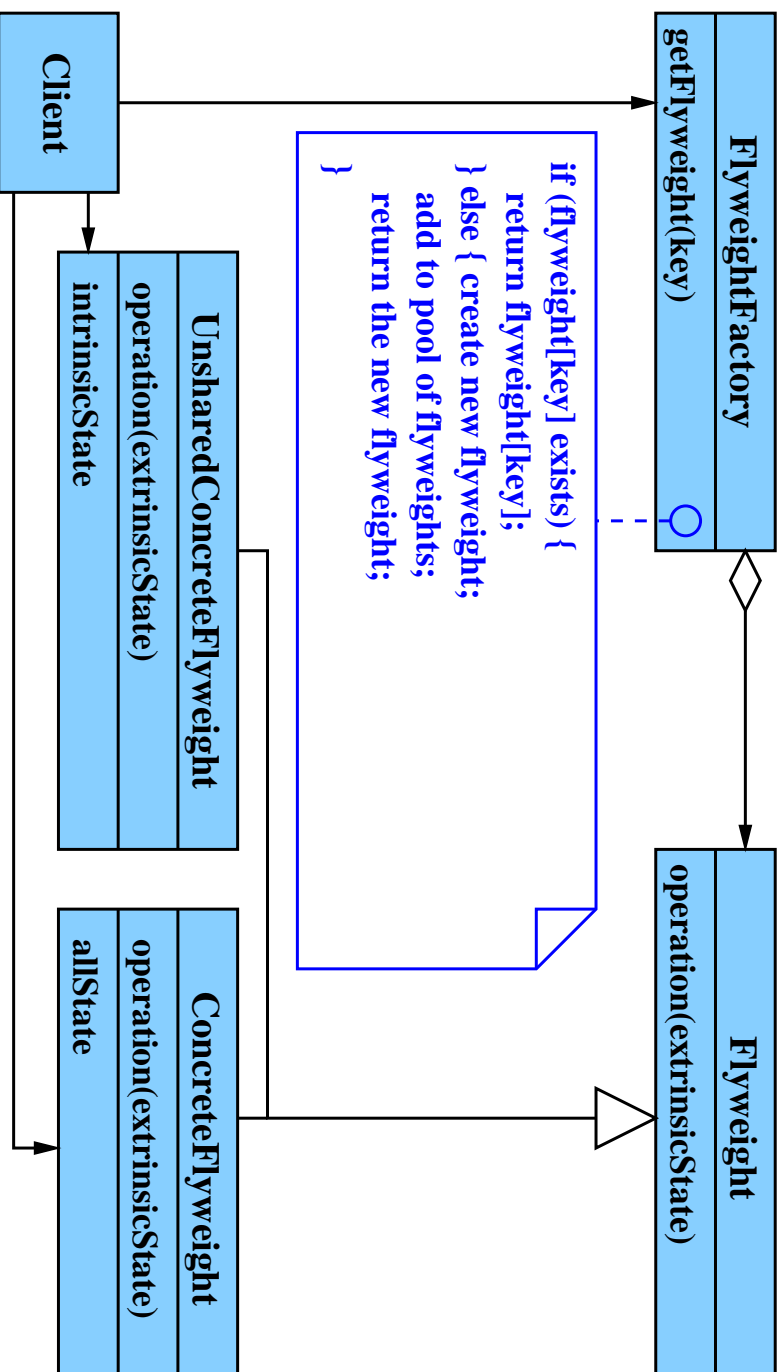
Motivation (2)



Motivation (3)



Structure



Applicability

- many similar objects
- memory consumption to high for “full objects”
- state decomposable in intrinsic and extrinsic state
- identity of objects does not matter

Consequences

- decreased memory consumption
- potentially increased time
due to passing of extrinsic state

Pattern: Composite

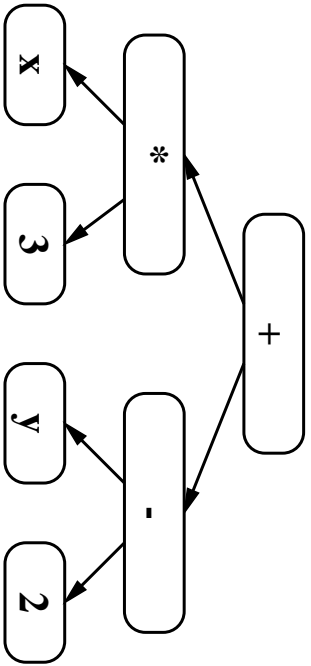
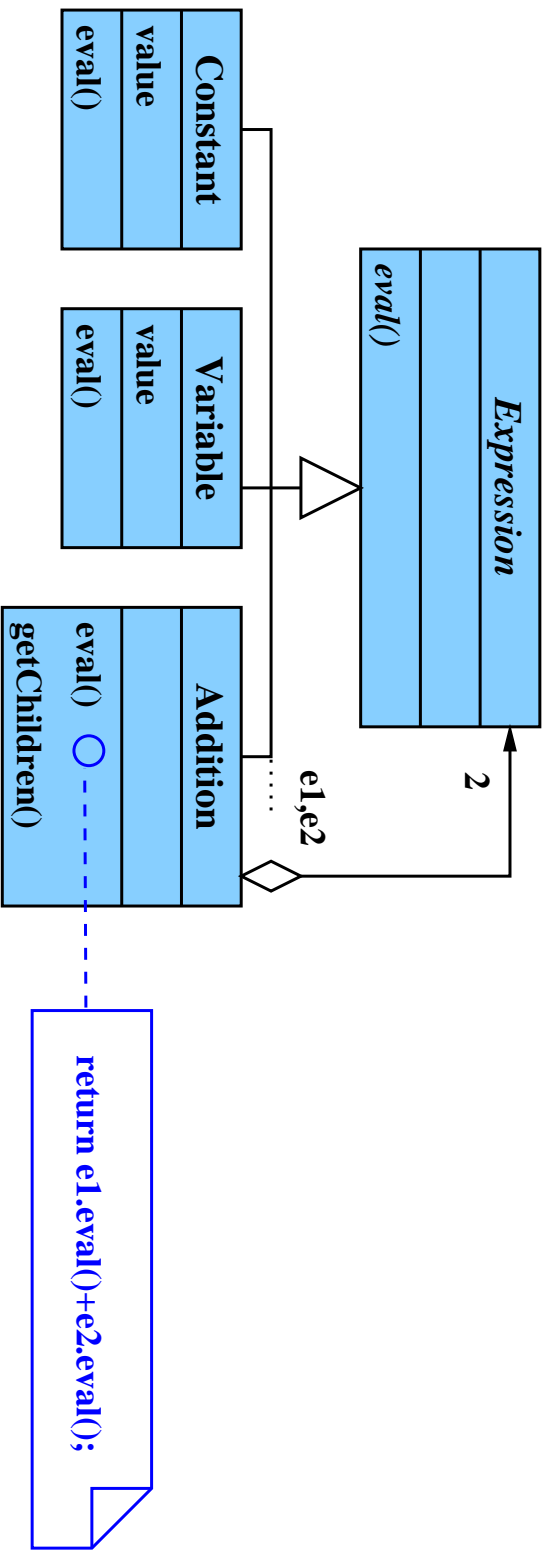
object, structural

Intent

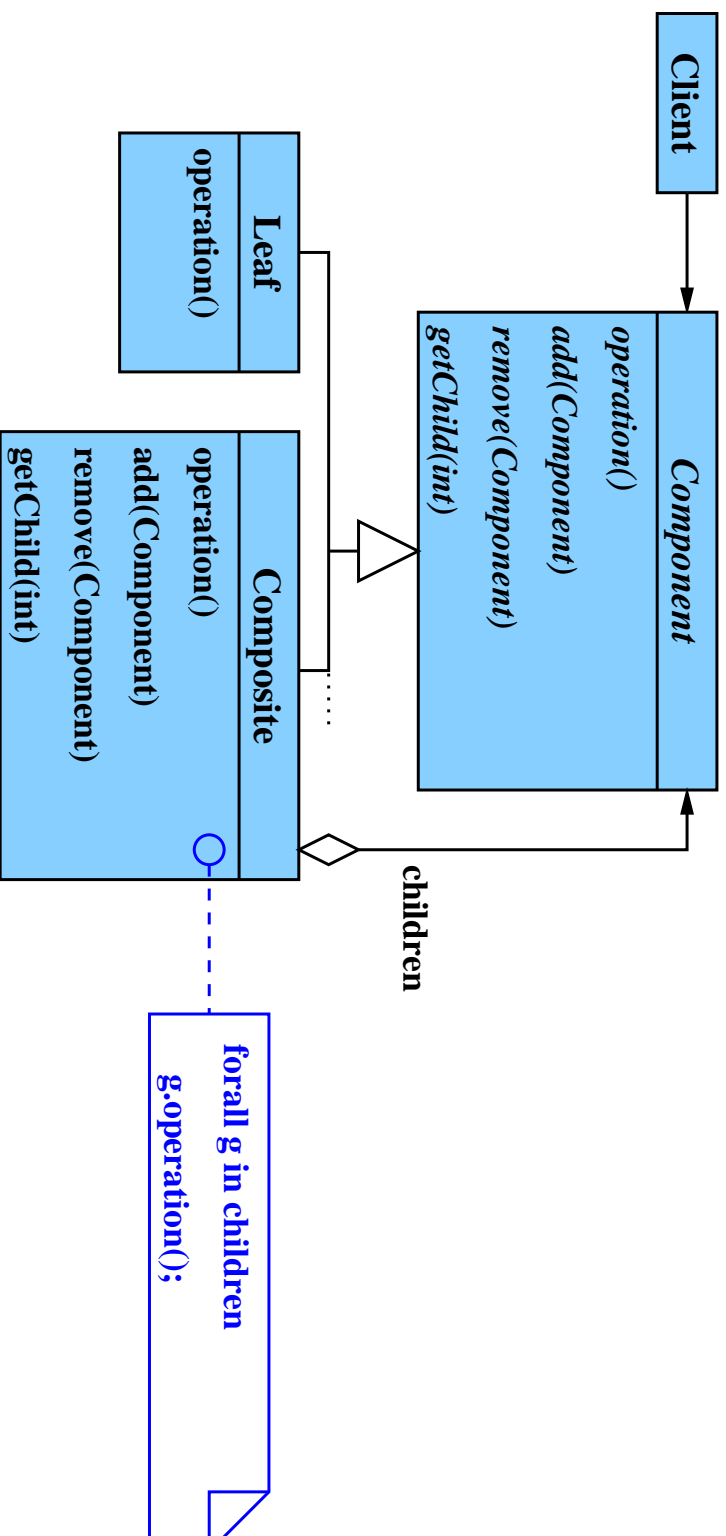
- recursive object structures
- uniform treatment of leaf components and containers

Motivation

1. components of a document (window) contain further components:
nested enumeration lists, figures, . . .
2. control structures of PL can be nested arbitrarily: nested loops, conditionals, . . .
→ syntax tree
processing in a compiler (type checking, code generation) centered around syntax tree
3. arithmetic expression consists of subexpressions
evaluation follows tree structure



Structure



Applicability: recursive object structures

Consequences

- uniform client code
- easy to add new composite classes as well as leaf classes

Related Patterns

- Decorator
- Flyweight (for leaves)
- Iterator (for traversing)

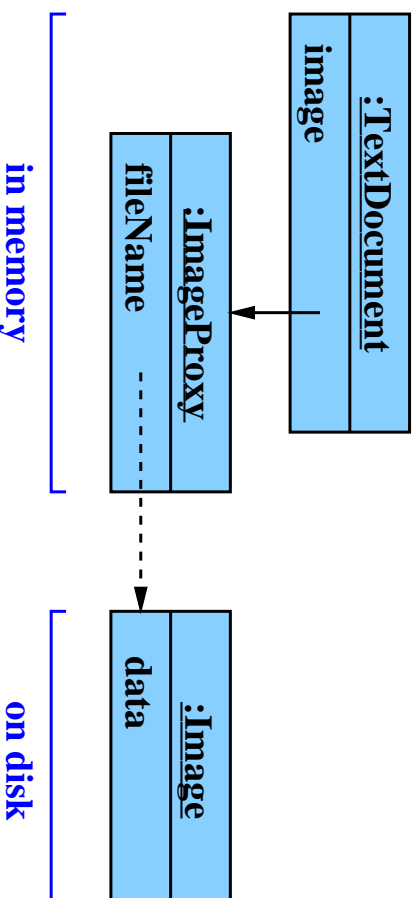
Pattern: Proxy (Surrogate)

object, structural

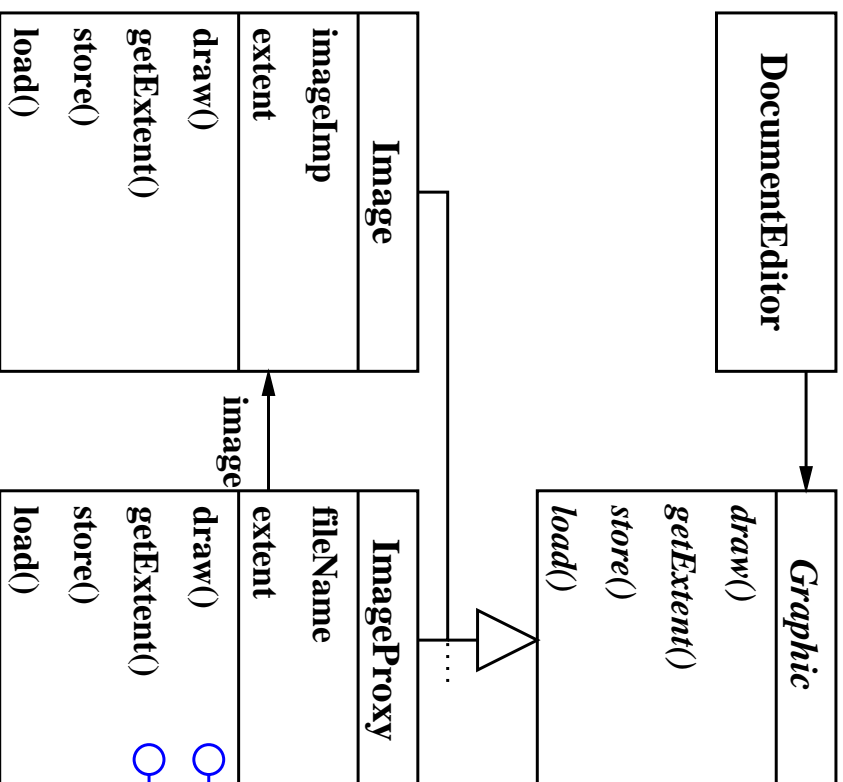
Intent: controll access to object

Motivation

- multi-media editor loads images, audio clips, videos etc on demand
- represented by proxy in document
- proxy loads the “real object” on demand



Motivation (2)



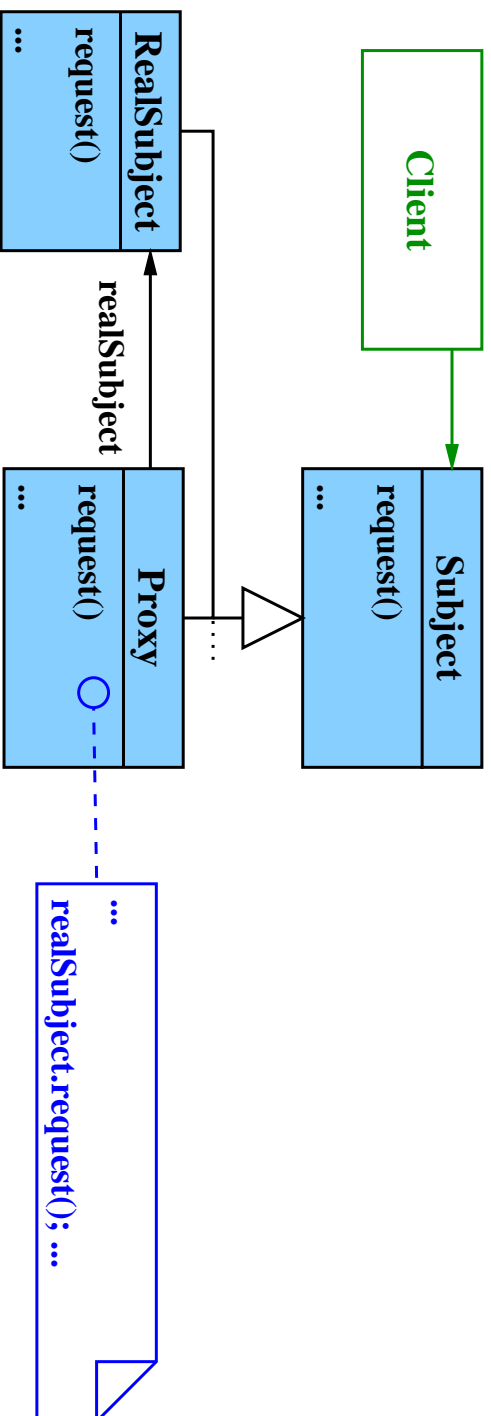
```

if (image == NULL)
    image = loadImage(fileName);
image.draw();
  
```

```

if (image == NULL)
    return extent;
else return image.getExtent();
  
```

Structure



Applicability

1. *remote proxy* communication with object on server (CORBA)
2. *virtual proxy*
 - creates expensive objects on demand
 - delays cost of creation and initialization
3. *protection proxy* controls access permission to original object
4. *smart reference* additional operations: reference counting, locking, copy-on-write

Comparison of Structural Patterns

- similar underlying concepts:
 - class-based → inheritance
 - object-based → object composition
- different goals

Adapter vs. Bridge vs. Facade

- all: flexibility through indirection
- differences

Adapter: reconciling differences between existing interfaces

Facade: bundling of interfaces

Bridge: interface with multiple, dynamically exchangeable implementations

Composite vs. Decorator

- both: recursive composition to organize open-ended number of objects
- Decorator adds responsibilities without subclassing
- Composite enables uniform processing of object graphs
- complementary → often used in concert

Decorator vs. Proxy

- both: indirection and forwarding
- Proxy:
 - controls access to particular object
 - not recursive
- Decorator:
 - stepwise addition of responsibilities
 - recursive

4.4.3 Behavioral Patterns

- algorithms and assignment of responsibilities
- patterns of communication

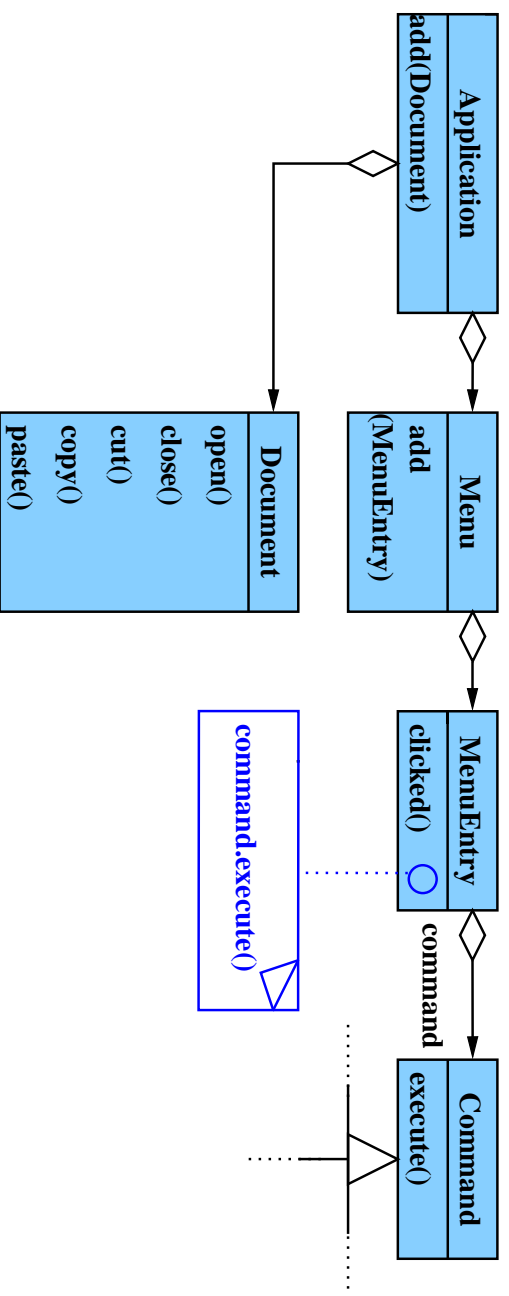
Pattern: Command (Action, Transaction) object, behavioral

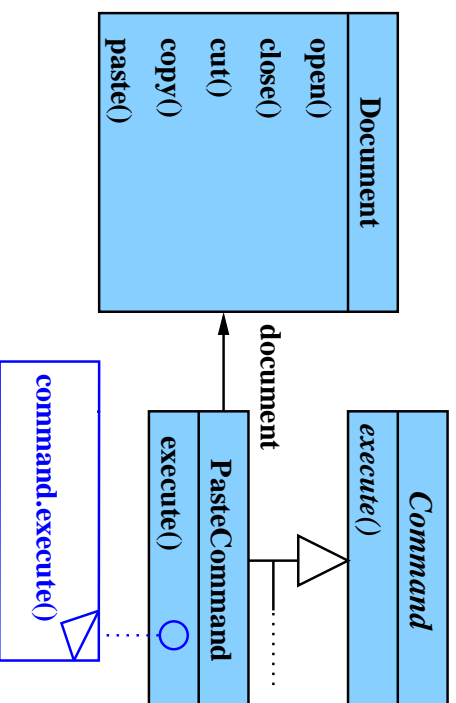
Intent

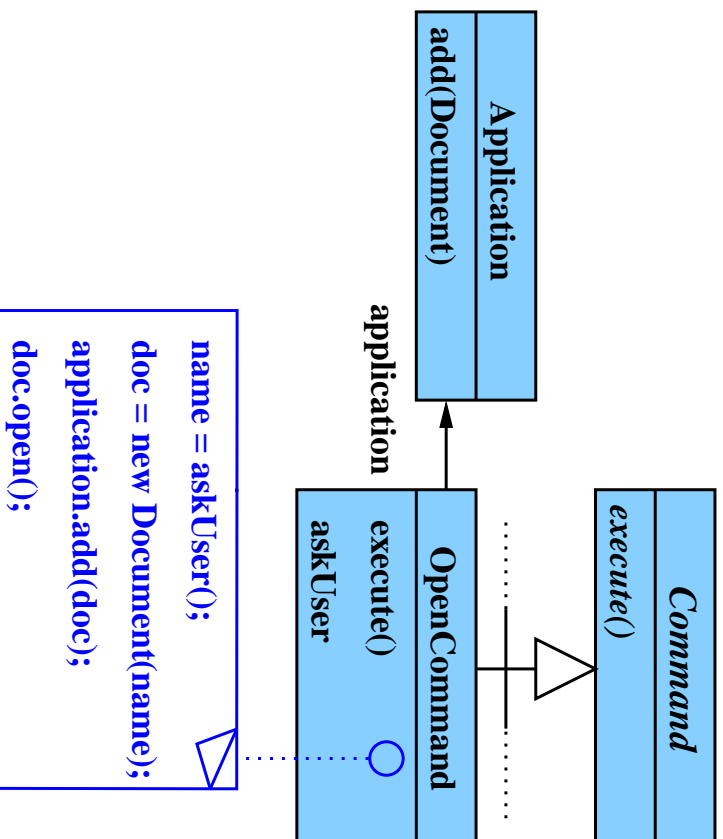
- Encapsulate request as object
- parameterize clients with different requests
- queue or log requests
- support undo-able operations

Motivation

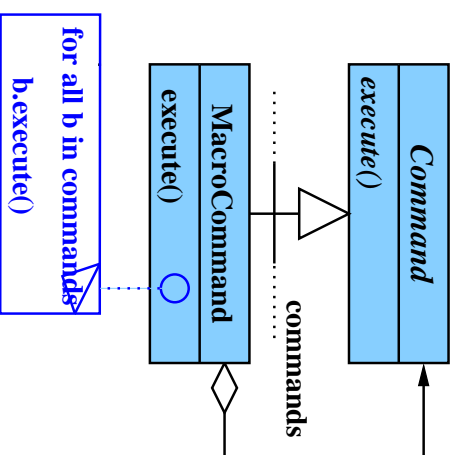
- toolkit object (buttons, menus) must trigger an operation
- but which operation and on which object?
- → request object encapsulates operation and receiver



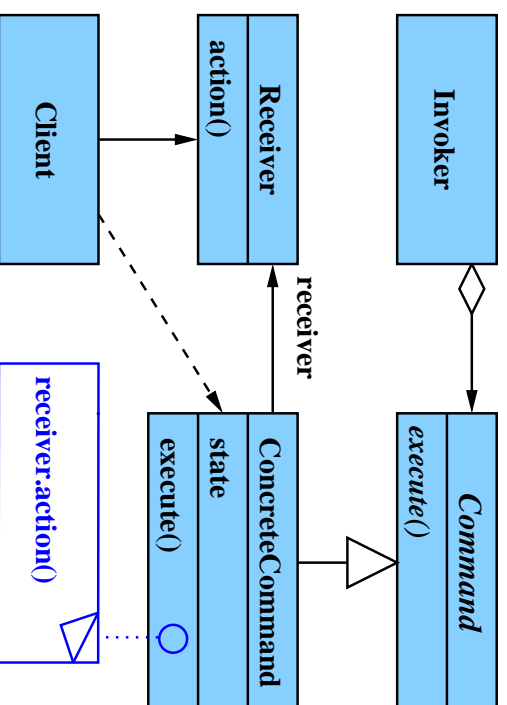


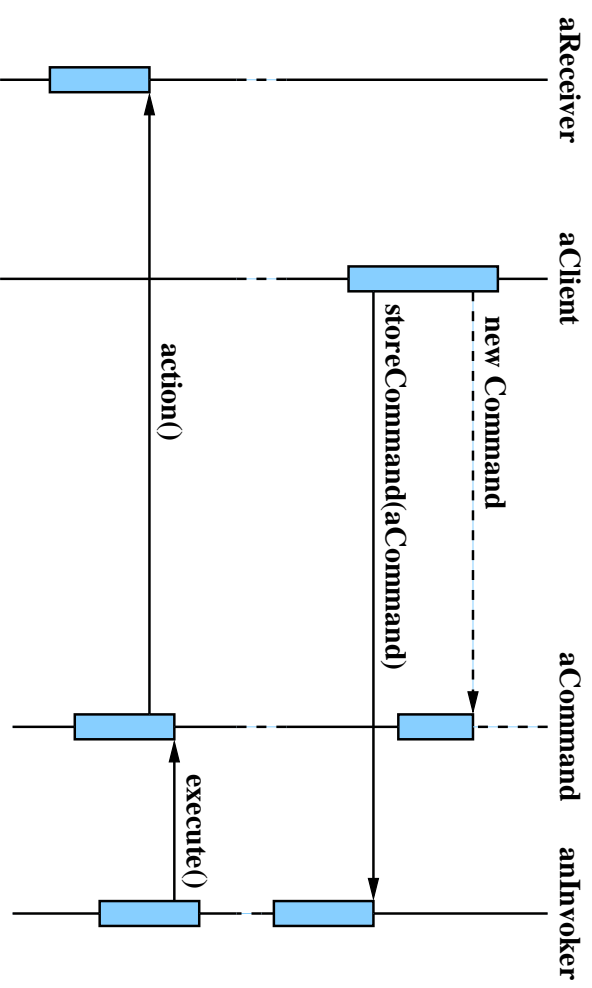


Macro Commands



Structure





Applicability

- parameterize objects by action to perform → oo callback
- specify, queue, and execute requests at different times
- support “undo”
- keep change log (**recovery**)
- model transactions

Consequences

- Commands vs. first-class functions
- assemble commands to composite commands
- easy to add new Commands

Pattern: Observer

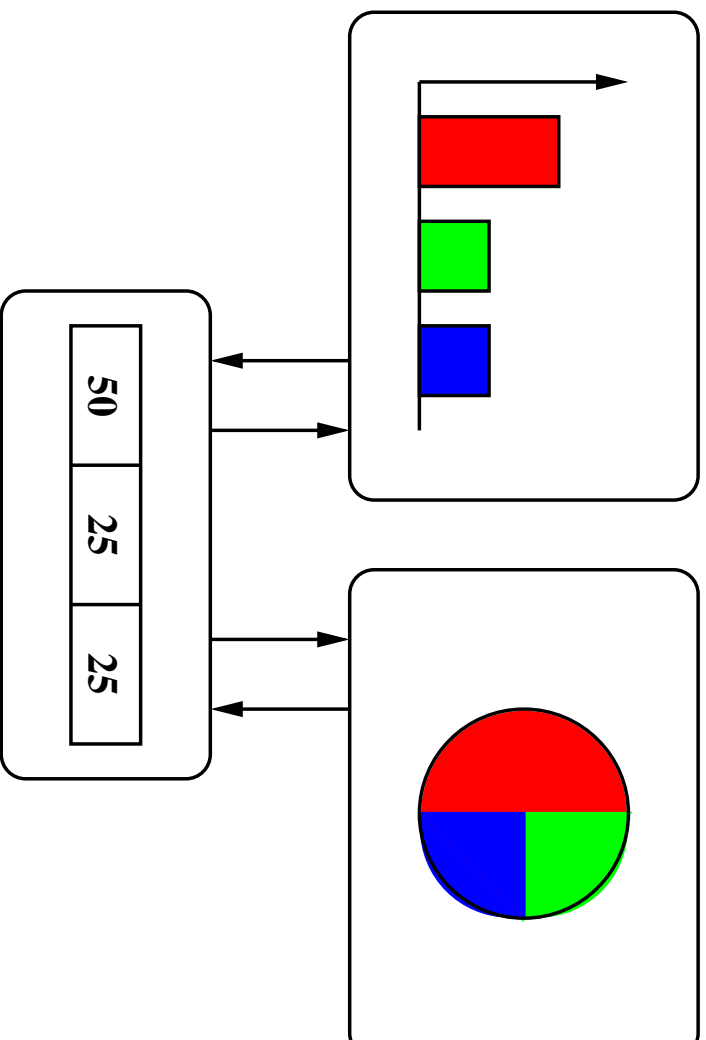
object, behavioral

Intent

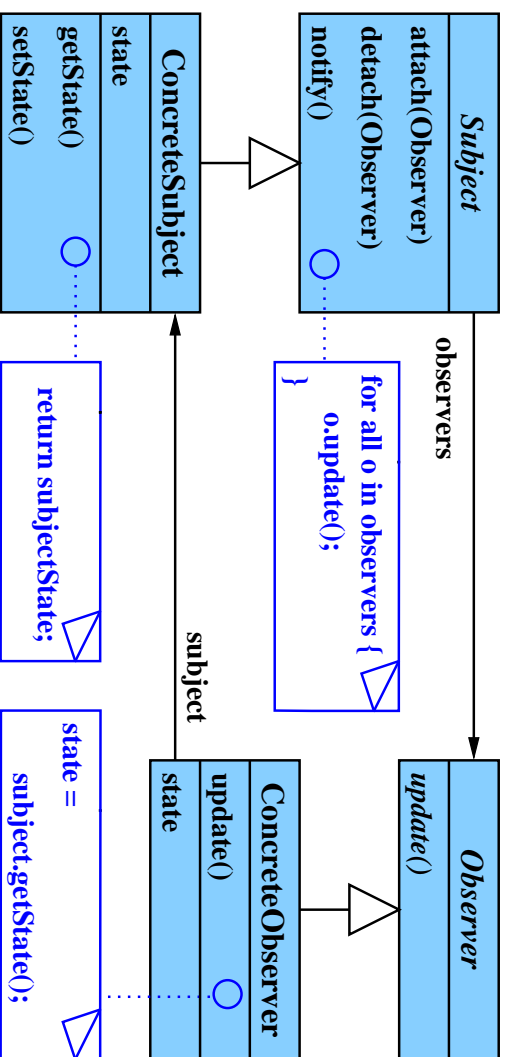
- define 1 : n -dependency between objects
- state-change of one object notifies all dependent objects

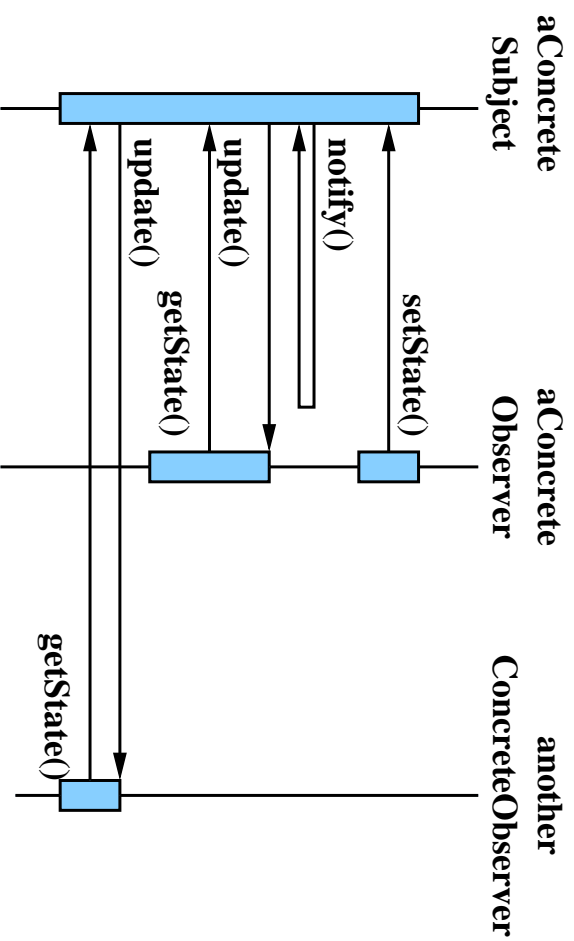
Motivation

- maintain consistency between internal model and external views



Structure





Applicability

- objects with at least two mutually dependent aspects
- propagation of changes
- anonymous notification

Consequences

- Subject and Observer are independent (abstract coupling)
- broadcast communication
- observers dynamically onfigurable
- simple changes in Subject may become costly
- granularity of update ()

Pattern: Visitor

object, behavioral

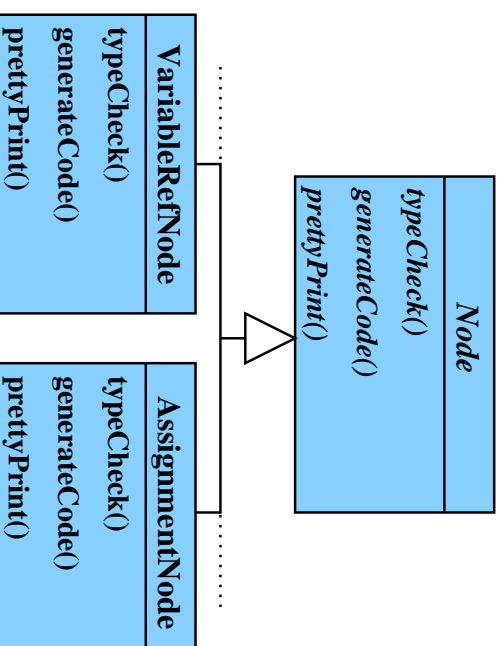
Intent

- represents operations on an object structure by objects
- new operations without changing the classes

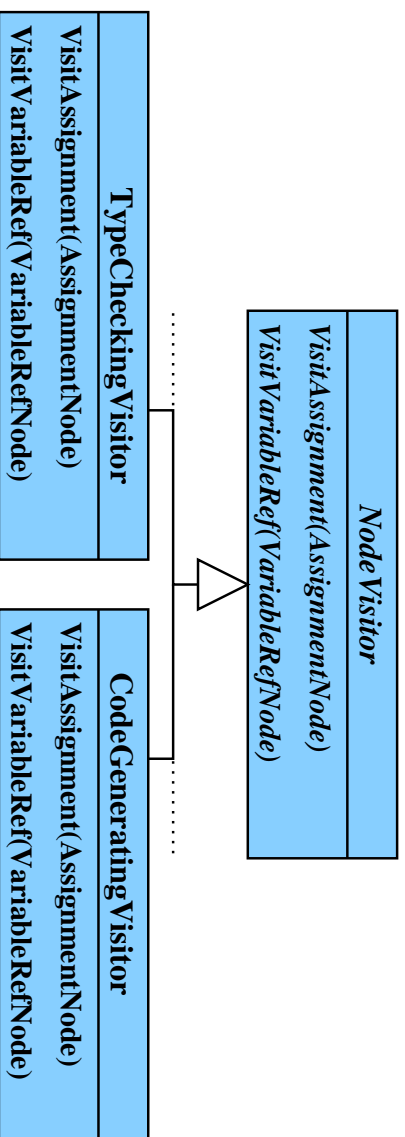
Motivation

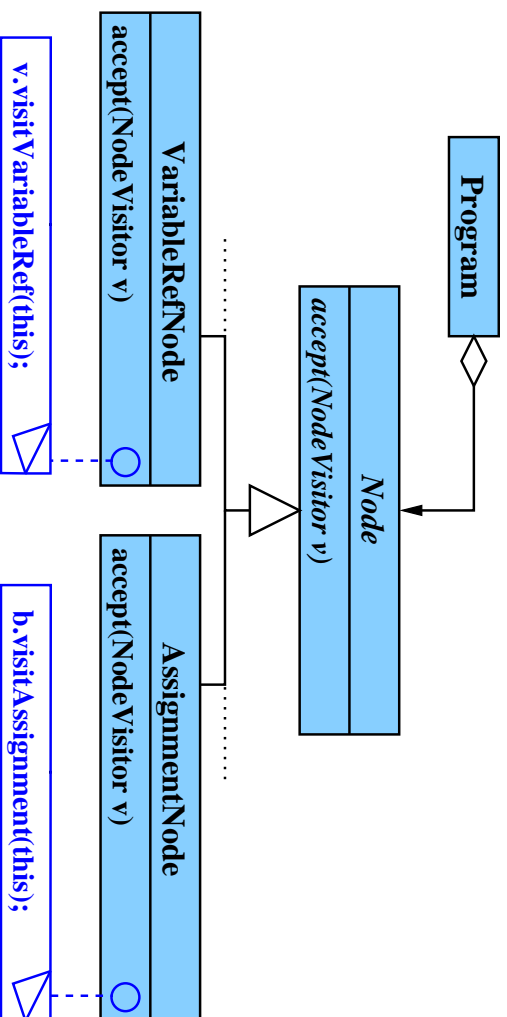
- processing of a syntax tree in a compiler: type checking, code generation, pretty printing, ...
- naive approach: put operations into node classes → hampers understanding and maintainability
- here: realize each processing step by a visitor

without visitor:

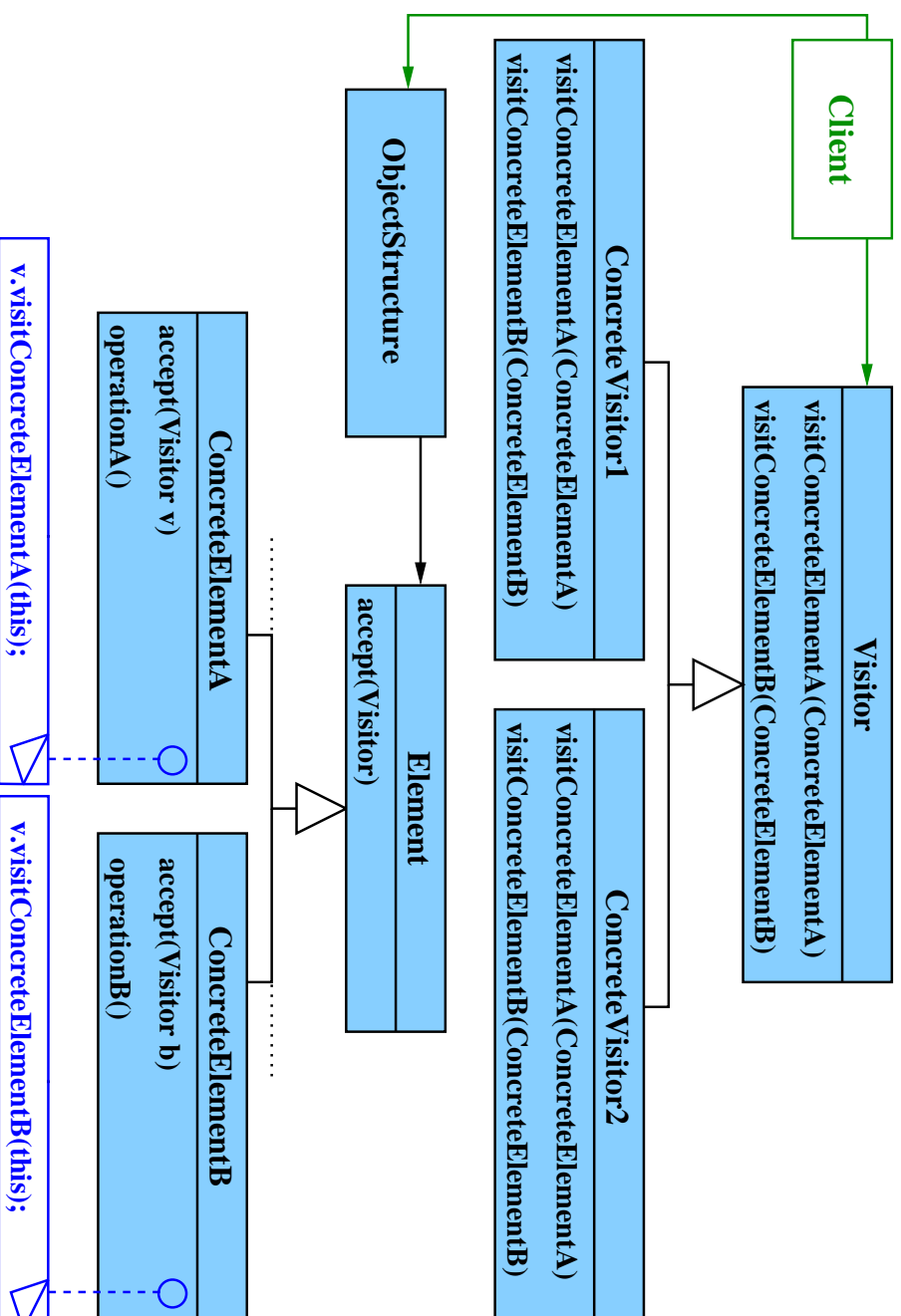


Syntax Tree with Visitors

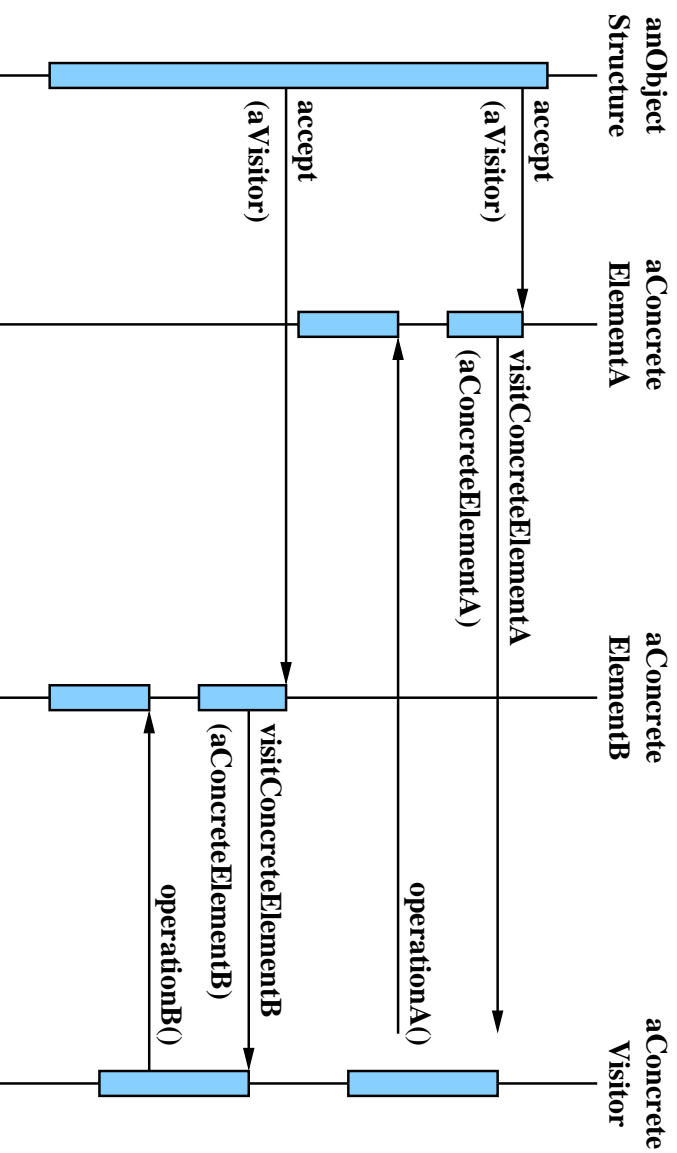




Structure



Visitor: Interaction Diagram



Applicability

- object structure with many differing interfaces; processing depends on concrete class
- distinct and unrelated operations on object structure
- not suitable for evolving object structures

Consequences

- adding new operations easy
- visitor gathers related operations
- adding new ConcreteElement classes is hard
- visitors with state
- partial breach of encapsulation

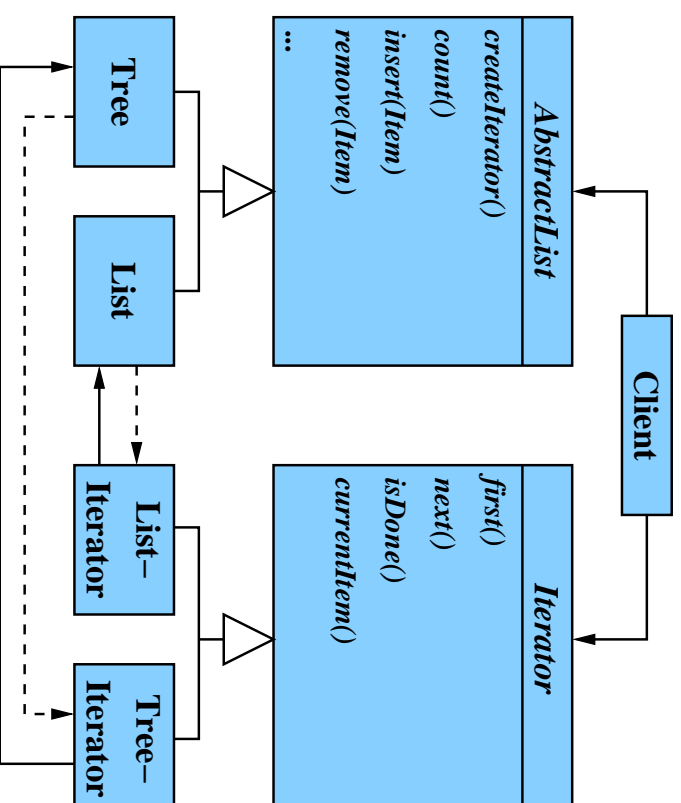
Pattern: Iterator (Cursor)

object, behavioral

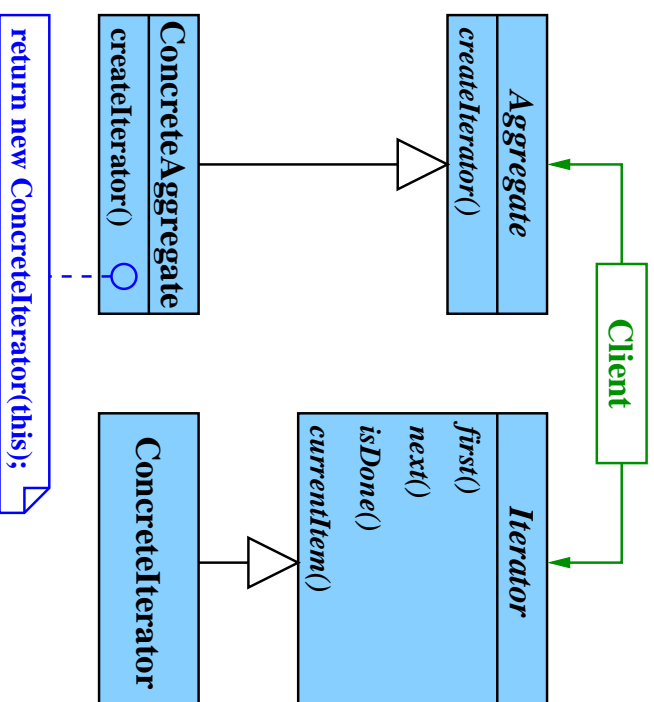
Intent

- sequential access to components of a container object
- representation of object hidden

Motivation



Structure



- ConcreteIterator administers current object and determines subsequent object(s)

Applicability

- access objects “contents” without exposing representation
- support multiple traversals
- uniform interface for traversing different containers

Consequences

- easy switching between different styles of traversal
- simplifies Aggregate’s interface
- more than one pending traversal
- control of iteration (internal vs. external)
- traversal algorithm (Iterator vs. Aggregate)
- robustness (are modifications visible?)

Pattern: Memento

object, behavioral

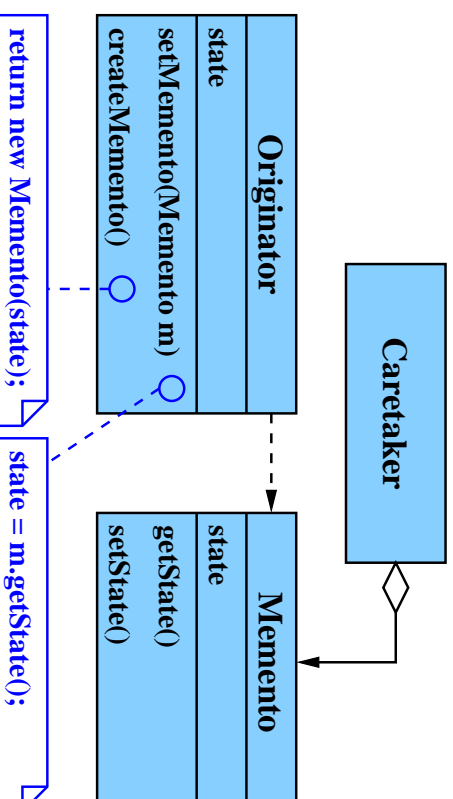
Intent

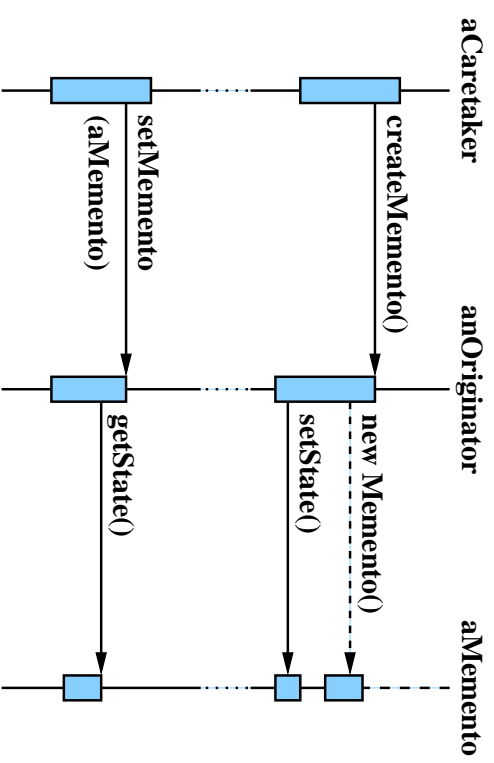
- capture object's internal without breaking encapsulation

Motivations

1. drawing tools (undo)
2. editors (undo)
3. search algorithms (backtracking)

Structure





Consequences

- preserves encapsulation boundaries
- simplifies Originator
- Mementos might be expensive
- hidden administrative cost

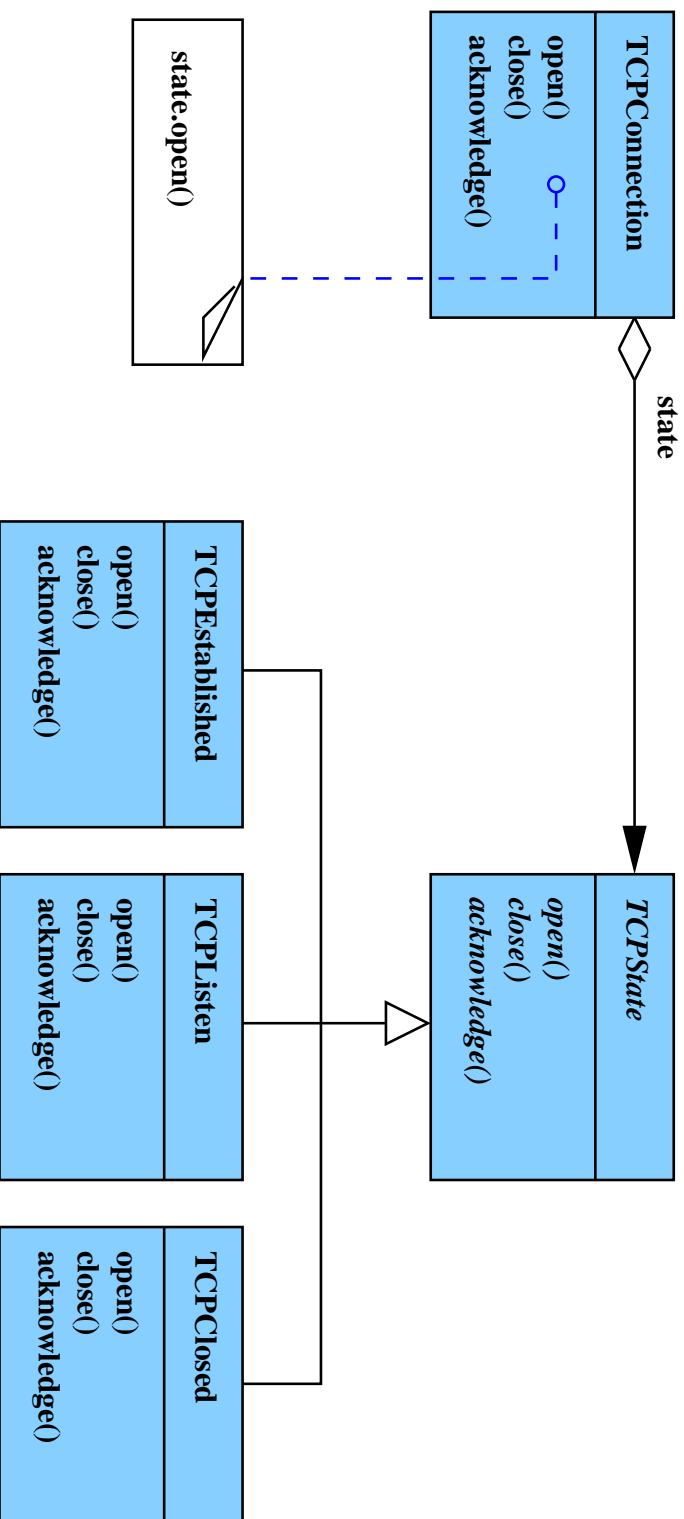
Pattern: **State**

object, behavioral

Intent: object changes behavior according to internal state change

Motivation

- lifecycle of a TCP connection:
 - Established
 - Listening
 - Closed
- object must respond differently to operations depending on state



Applicability

- objects behavior depends on (dynamic) state
- large conditional statements in operations

Consequences

- localizes state-specific behavior
- makes state transitions explicit (who is responsible?)
- sharing of state objects (creation, destruction?)

Pattern: Strategy (Policy)

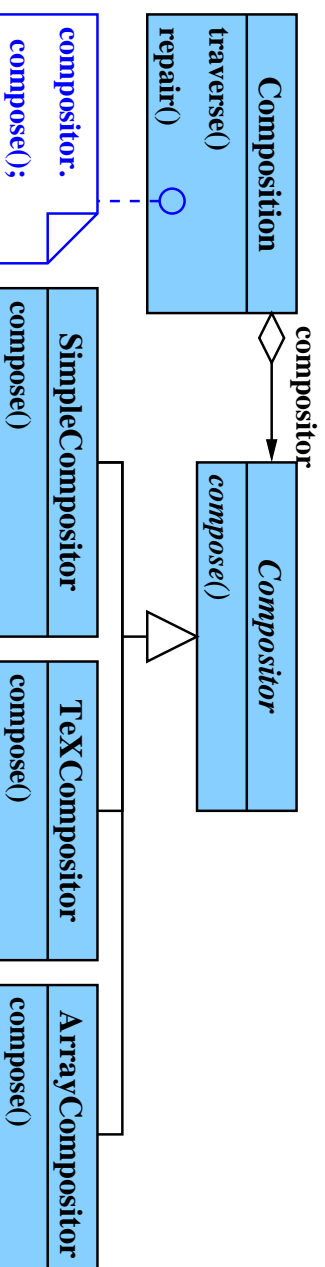
object, behavioral

Intent

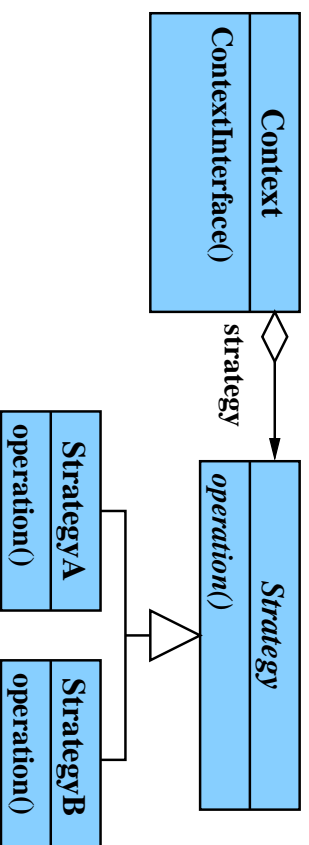
- encapsulate algorithms
- dynamic exchange of algorithms

Motivation

- breaking text into lines
- many algorithms with different trade-offs
- should not be tied to client classes
→ hard to exchange
- hence: encapsulate algorithms



Structure



Applicability

- families of algorithms with identical interface

Further behavioral patterns

Interpreter

Template Method:

- algorithmic skeleton in super class
- auxiliary methods in subclasses (cf. Factory Method)

Mediator: encapsulate interaction of a set of objects into an object

Chain of Responsibility