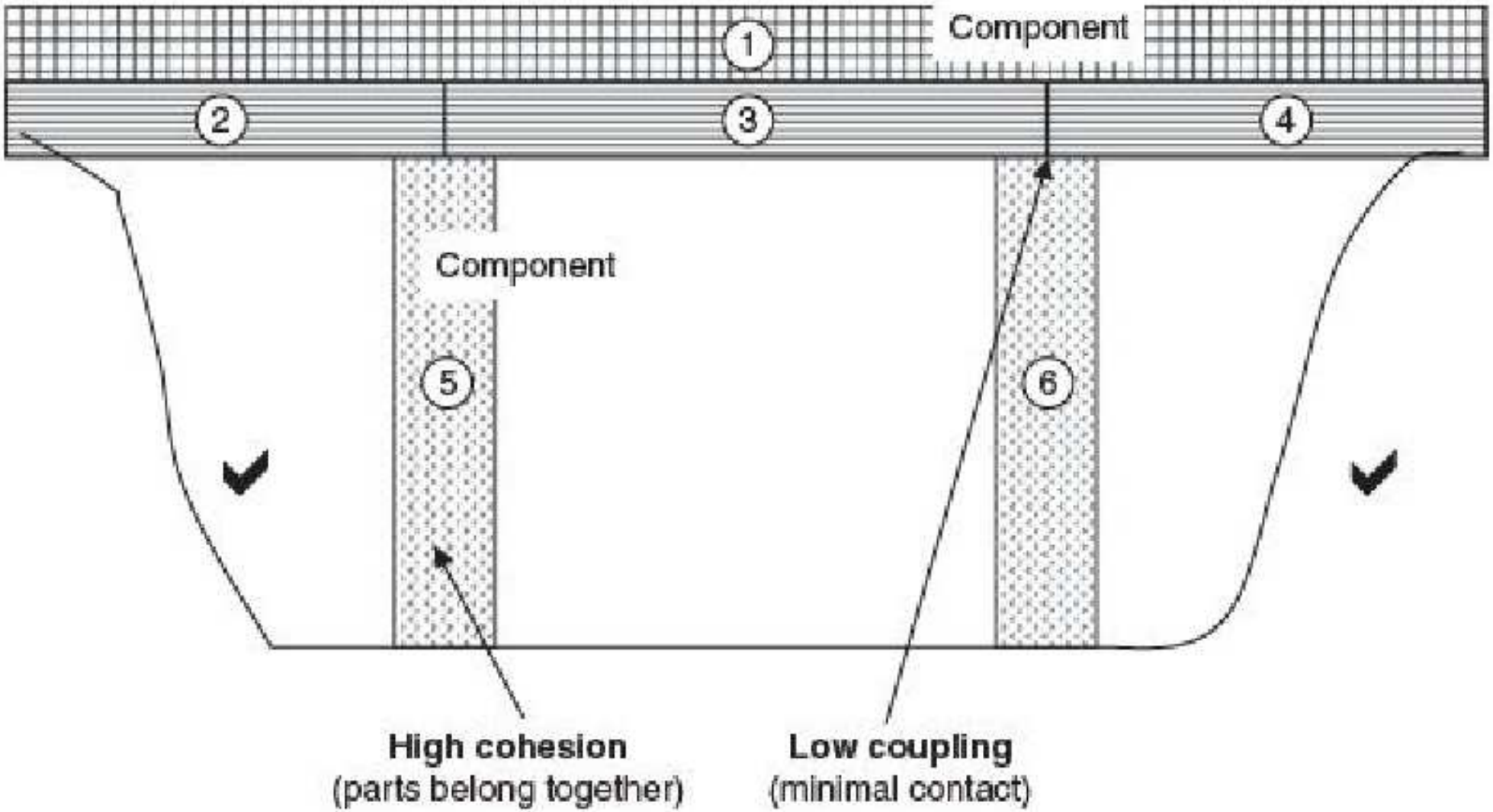
Tvorba informačných systémov

5. prednáška – návrh, architektúra, návrhové vzory

Principal goals of software design:

- *Sufficiency*: handles the requirements
- *Understandability*: can be understood by intended audience
- *Modularity*: divided into well-defined parts
- *Cohesion*: organized so that like-minded elements are grouped together
- *Coupling*: organized to minimize dependence between elements
- *Robustness*: can deal with wide variety of input
- *Flexibility*: can be readily modified to handle changes in requirements
- *Reusability*: can use parts of the design and implementaion in other applications
- *Information hiding*: module internals are hidden from others
- *Efficiency*: executes within acceptable time and space limits
- *Reliability*: executes with acceptable failure rate



High cohesion and low coupling—bridge example

Poznámky

- aby malo zmysel zaoberať sa architektúrou a jej popisom, musí tento popis **skryť** niektoré (menej podstatné) informácie
- popis architektúry je teda abstraktným pohľadom na systém
- predpoklad: podstatné informácie sú tie, ktoré sa týkajú **vzťahov medzi komponentami** – architektúra ostatné informácie skrýva
- **externe viditeľné vlastnosti** = predpoklady o komponente, s ktorými môžu ostatné komponenty počítať, napríklad:
  - poskytované služby,
  - výkonnostné vlastnosti,
  - spracovanie chybových stavov,
  - použitie zdieľaných zdrojov (pamäť, zásobník, kom. kapacita, ...),
  - ...

Poznámky 2

Architektúra pozostáva z viac než jednej štruktúry, napríklad:

- **statické štruktúry:** moduly/triedy ako kompilačné jednotky a vzťahy „je súčasťou“, „používa“, „dedí“ (v prípade tried) medzi nimi
- **dynamické štruktúry:** komponenty rozpoznateľné počas behu (procesy, bežiace programy, objekty, ...) a vzťahy „komunikuje s“, „čaká na“, „posiela údaje pre“, „je spojený smerníkom s“ ... medzi nimi
- **fyzické štruktúry:** vzťahy „beží na“ medzi procesmi, programami, objektami a pod. a hardvérovými zariadeniami; vzťahy „je spojený s“ medzi hardvérovými zariadeniami
- definícia architektúry nehovorí o typoch komponentov ani vzťahov

Poznámky 3

- **každý systém má architektúru**
  - potenciálne (už) neznámu
- **architektúra môže byť dobrá alebo zlá** – vždy vzhľadom k požiadavkám, ktoré sú na systém kladené
  - umožní alebo zabráni dosiahnuť systému požadované vlastnosti

Architektúra a návrh

Návrh obsahuje aj iné než architektonické rozhodnutia, napríklad:

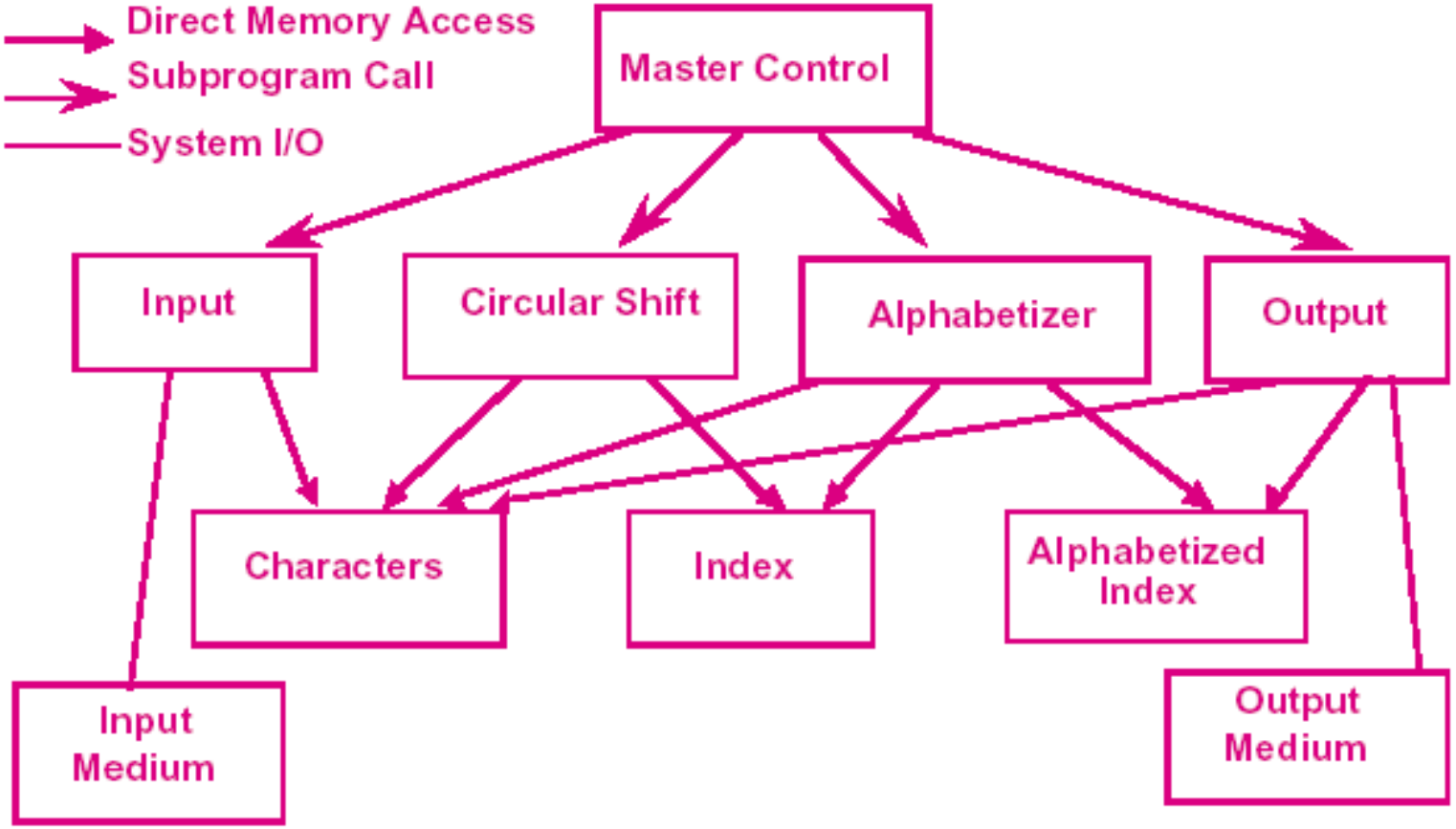
- aké algoritmy a dátové štruktúry budú použité vnútri modulov
- vzhľad používateľského rozhrania

Príklad: Key Word In Context

- systém vezme riadky textu, vyrobí všetky cyklicky posunuté varianty a utriedi ich podľa abecedy
- príklad – vstup:
  - hello world
  - this is a software architecture
- výstup:
  - a software architecture this is
  - architecture this is a software
  - hello world
  - is a software architecture this
  - software architecture this is a
  - this is a software architecture
  - world hello

Architektúra č. 1

zdieľané pamäťové štruktúry

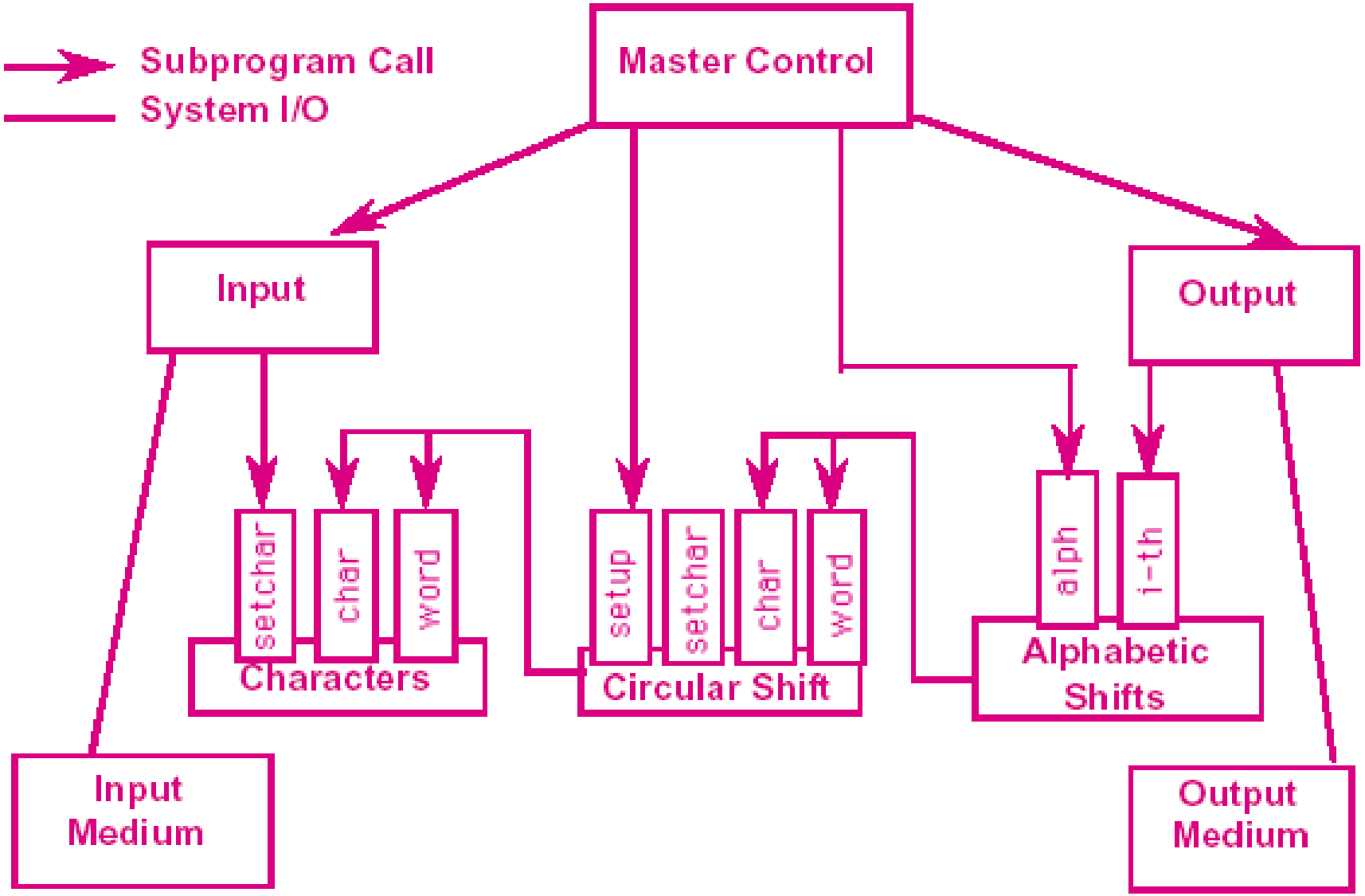


Vlastnosti

- výkon (čas, pamäť): **dobrý**
- ľahkosť zmeny vnútornej reprezentácie údajov: **nízka**
- ľahkosť opakovaného použitia komponentov: **nízka**

Architektúra č. 2

skrývanie informácií

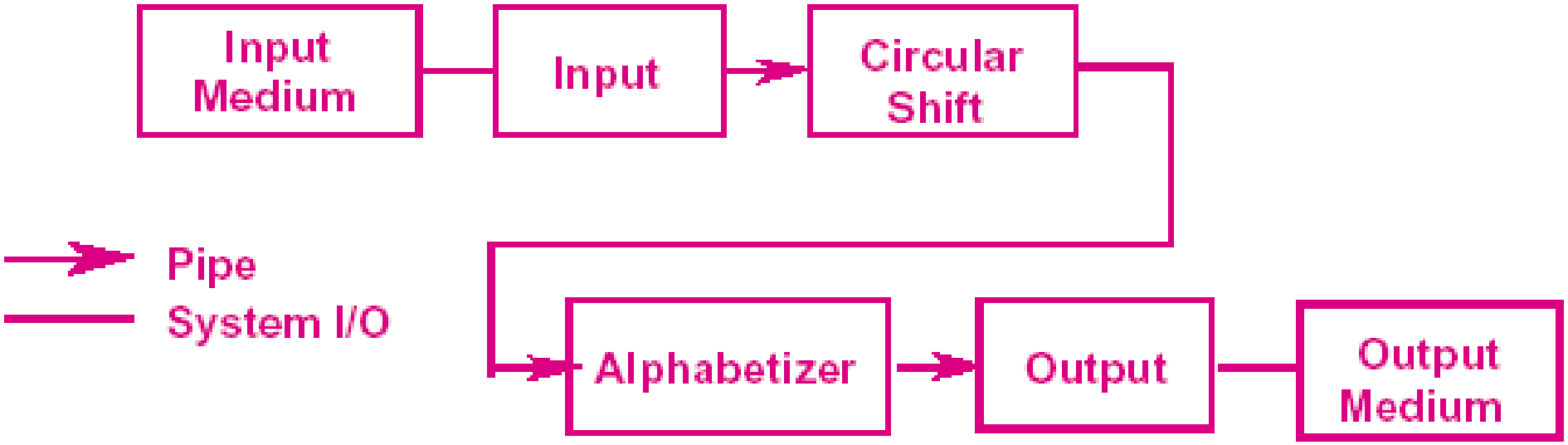


Vlastnosti

- výkon (čas, pamäť): **pomerne dobrý**
- ľahkosť zmeny vnútornej reprezentácie údajov: **dobrá**
- ľahkosť opakovaného použitia komponentov: **pomerne dobrá**

Architektúra č. 3

pipes-and-filters



Vlastnosti

- výkon (čas, pamäť): **pomerne slabý**
- ľahkosť zmeny vnútornej reprezentácie údajov: **dobrá**
- ľahkosť opakovaného použitia komponentov: **výborná**

Výkon (performance)

- výkon = (orientačná definícia) čas potrebný na reakciu systému resp. množstvo spracovaných udalostí za jednotku času

- **riešenie na architektonickej úrovni:** minimalizovať komunikáciu medzi komponentami (najmä ak sú na rôznych počítačoch, v rôznych procesoch a pod.)

Bezpečnosť (security)

- bezpečnosť = schopnosť systému odolávať pokusom o neoprávneným prístup, resp. narušenie činnosti

- **riešenia na architektonickej úrovni:**
  - oddelenie funkcií kritických z pohľadu bezpečnosti (autentifikácia, autorizácia, ...) do samostatných komponentov, ktoré sú dobre oddelené a chránené a ktoré môžu byť overiteľne korektne implementované
  - obmedzenie komunikácie medzi systémom a jeho okolím
  - ...

Spoľahlivosť (availability, reliability)

- availability = pravdepodobnosť, že systém je v danom okamihu dostupný
- reliability = pravdepodobnosť, že systém dá správny výsledok (resp. že nezlyhá)

- **riešenia na architektonickej úrovni:**
  - redundantné komponenty
  - dobre riešená komunikácia o chybách a ich spracovanie
  - špeciálne komponenty pre správu a monitorovanie (napr. watchdog timers)
  - ...

Funkčnosť (functionality)

- funkčnosť = systém poskytuje požadované funkcie (potrebné na plnenie úloh, pre riešenie ktorých bol vytvorený)
- táto vlastnosť v prevažnej miere nesúvisí s architektúrou

Jednoduchosť použitia (usability)

= ako rýchlo a ľahko sa používatelia naučia pracovať so systémom, do akej miery systém pomáha vyhnúť sa bežným používateľským chybám, či sa dá so systémom pracovať efektívne ...

táto vlastnosť v prevažnej miere nesúvisí s architektúrou

- vyplýva najmä z kvality používateľského rozhrania

Modifikovateľnosť (modifiability)

- modifikovateľnosť = ako ľahko sa v systéme robia zmeny (pridanie, odobranie, resp. zmena funkčnosti, prispôsobenie sa zmenám prostredia, reštrukturalizácia, oprava chýb)

- **podstatne závisí od architektúry**
  - námaha potrebná na vykonanie zmeny závisí najmä od toho, akú veľkú časť systému je potrebné upraviť
    - jeden modul
    - viac modulov
    - potrebný je zásah do architektúry

Modifikovateľnosť 2

Riešenie na architektonickej úrovni:

- izolovať miesta možných zmien do samostatných komponentov (a takto ich oddeliť od zvyšku systému príslušným rozhraním)

Portabilita

- portabilita = ako ľahko je systém upraviť tak, aby pracoval v inom (hardvérovo-softvérovom) prostredí
- špeciálny prípad modifikovateľnosti

- **riešenie na architektonickej úrovni:** uzavrieť kód špecifický pre platformu do jedného (resp. niekoľko málo) modulov – „portability layer“

Opakovaná použiteľnosť (reusability)

- reusability = časti systému resp. jeho architektúra môžu byť opakovane použité v budúcich systémoch

- **riešenie na architektonickej úrovni:** voľné väzby medzi komponentami

Integrovateľnosť (integrability)

- integrovateľnosť = schopnosť poskladať samostatne vyvinuté komponenty do fungujúceho celku (resp. možnosť použiť pri vytváraní systému už existujúce komponenty)

- riešenie na architektonickej úrovni:
  - zníženie externej zložitosti komponentov a interakčných mechanizmov
  - jasné rozdelenie zodpovednosti medzi komponenty
  - (pri existujúcich komponentoch samozrejme prispôsobenie sa ich rozhraniam)

súvisiaca vlastnosť: interoperabilita

Ďalšie faktory

- rýchlosť vytvorenia (time to market)
  - napr. výraznejším použitím existujúcich komponentov
- cena
  - závisí od architektúry napr. tým, aké technológie sa použijú
- očakávaná životnosť systému
  - ak dlhá – podstatná je modifikovateľnosť a portabilita (ovplyvňuje cenu a rýchlosť vytvorenia)

Vzory

Definícia: Vzor (pattern) je vopred pripravená množina návrhových (resp. analytických alebo implementačných) rozhodnutí vhodných pre istú situáciu, o ktorej sú známe určité vlastnosti. Túto množinu rozhodnutí je možné ako celok prevziať a po príslušnej úprave použiť.

Vzory existujú na rôznych úrovniach:

- architektonické vzory (štýly)
- návrhové vzory
- implementačné vzory (idiómy)
- analytické vzory

Súčasťou vzoru je obyčajne:

- názov
- popis problému
- popis riešenia
- dôsledky aplikovania riešenia (vlastnosti riešenia)

Príklady arch. štýlov

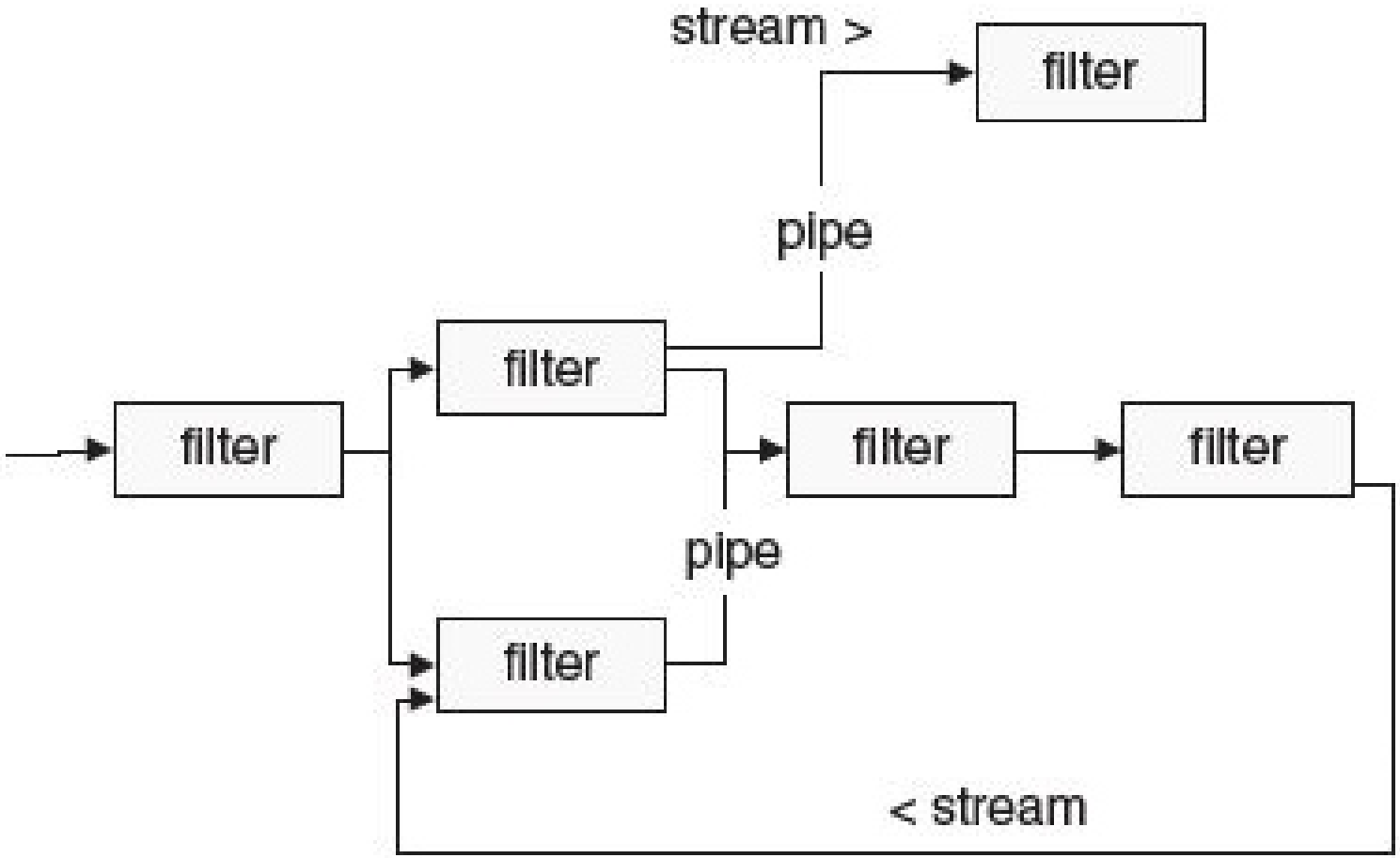
- Data-Centered
  - Repository
  - Blackboard
- Data Flow
  - Batch Sequential
  - Pipes and Filters
- Call and Return
  - Main Program and Subroutine
  - Object Oriented
  - Layered
- Virtual Machine
  - Interpreter
  - Rule-Based System
- Independent Components
  - Communicating Processes
  - Event Systems
    - Implicit Invocation
    - Explicit Invocation

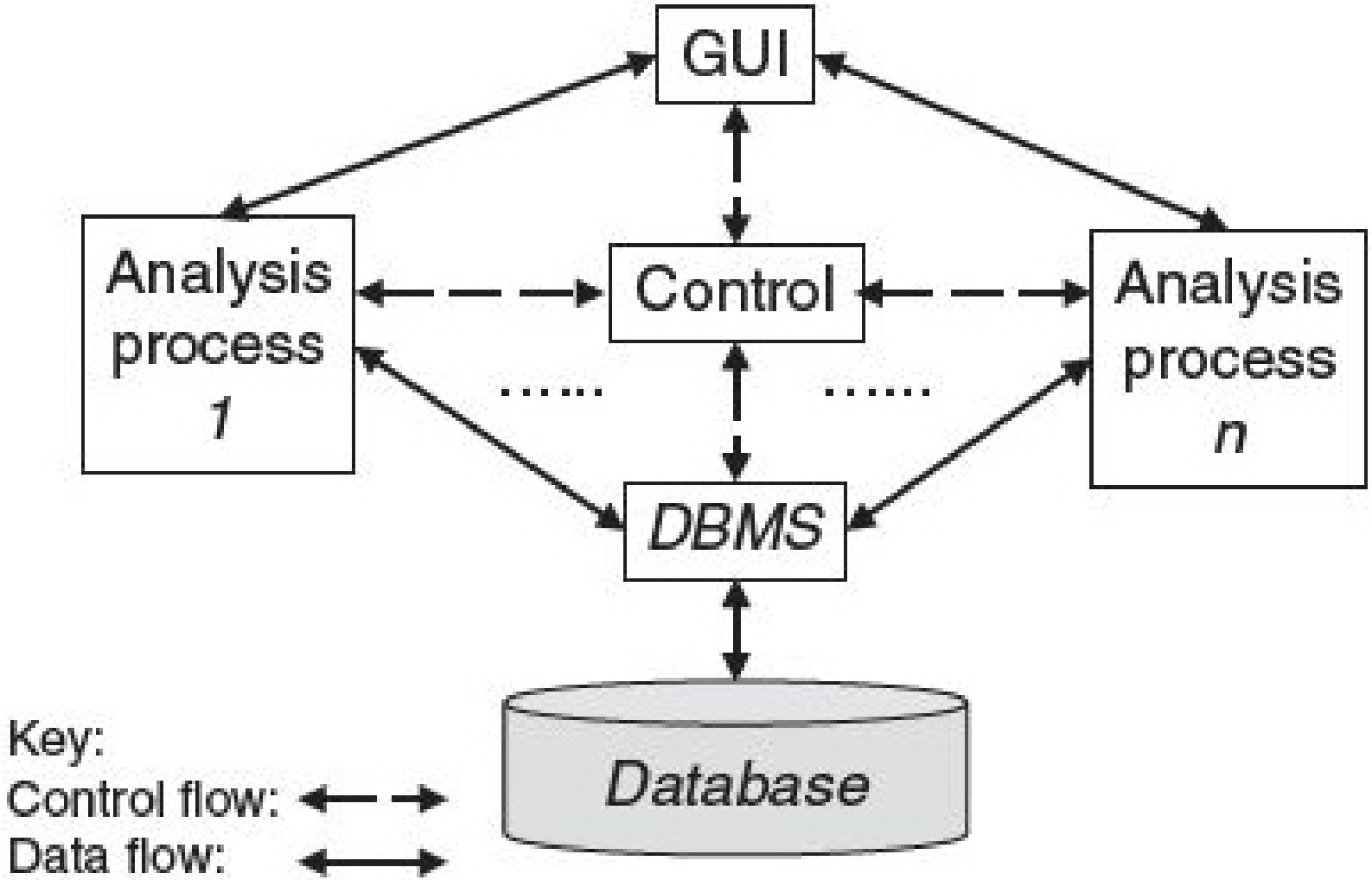
tieto štýly **nie sú** disjunktné !

Príklady architektonických štýlov (Software Engineering, Modern Approaches)

- Dataflow architectures
  - Pipes and filters
  - Batch sequential
- Independent components
  - Client-server systems
  - Parallel communicating processes
  - Event systems
  - Service-oriented (added)
- Virtual machines
  - Interpreters
  - Rule-based systems
- Repository architectures
  - Databases
  - Hypertext systems
  - Blackboards
- Layered architectures

Shaw and Garlan's categorization of software architectures

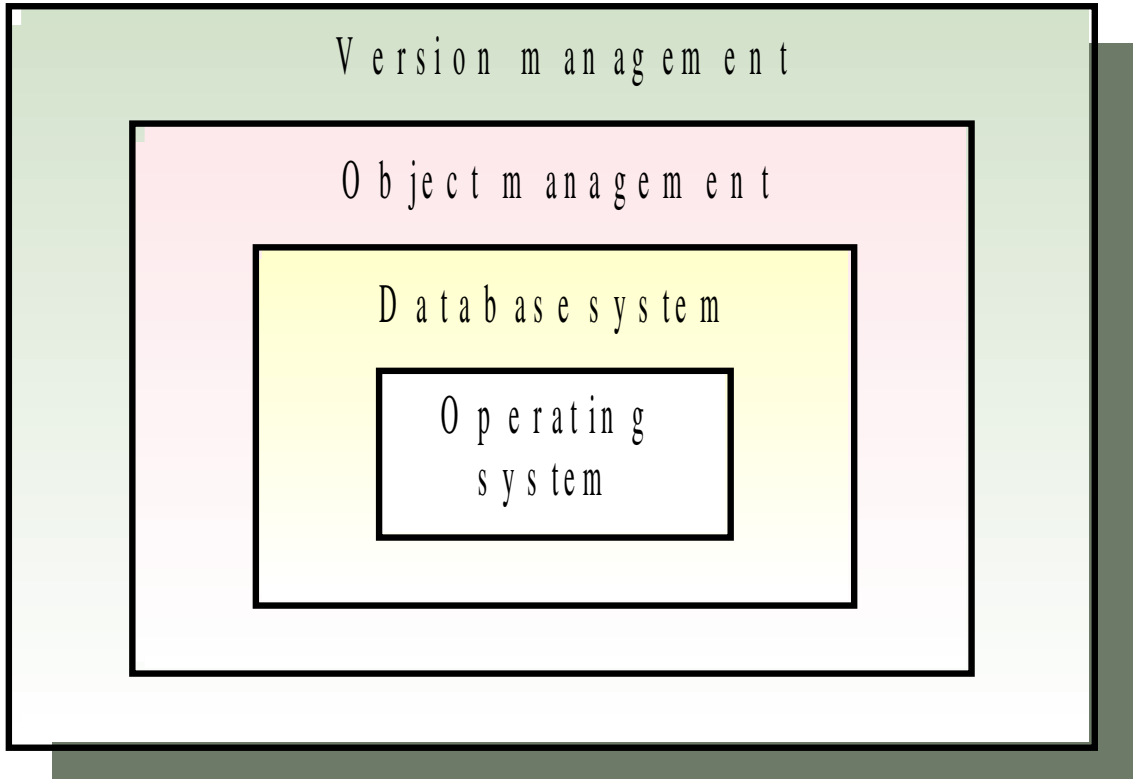

stream >
filter
pipe
filter
filter
filter
filter
pipe
filter
< stream



A typical repository architecture

Príklad č. 1: štýl Layered

- **organizuje systém do postupnosti vrstiev, z ktorých každá poskytuje definovanú množinu služieb**
- vrstva N je implementovaná pomocou služieb vrstvy N-1 (alternatívne: pomocou služieb vrstiev 1 až N-1)
- dá sa použiť s rôznymi typmi komponentov a konektorov



Iné príklady:

niektoré operačné systémy, OSI Reference Model, ...

Vlastnosti

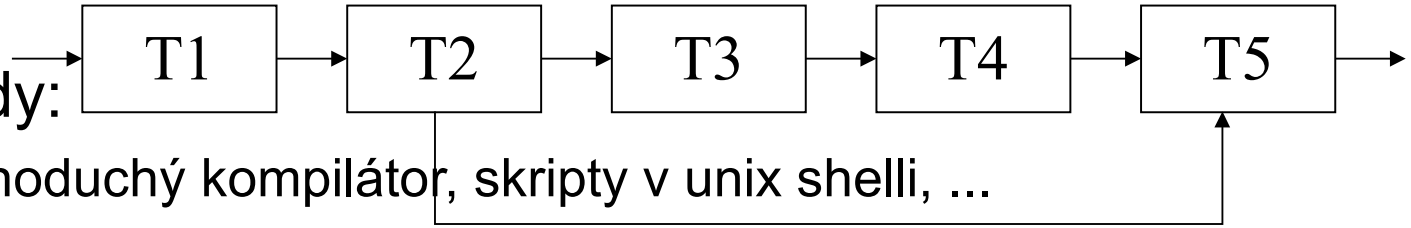
- Výhody:
  - prehľadnosť
  - jednoduchosť modifikácie, portabilita
- Nevýhody (najmä pri striktnej verzii modelu):
  - nie je vždy možné systém organizovať týmto spôsobom, pretože niektoré služby môžu byť potrebné vo všetkých vrstvách
  - problém s efektívnosťou

Kedy použiť

- keď rozdelenie na vrstvy pomôže znížiť zložitosť problému bez ujmy na ostatných vlastnostiach systému
- keď je dôležitá portabilita

Príklad č. 2: Pipes and Filters

- **popisuje systém ako sériu postupných transformácií údajov**
- komponenty:
  - filtre – transformujú vstupné údaje na výstupné, nemajú trvalý vnútorný stav, majú len minimálne väzby na okolie, bežia (navzájom) asynchrónne
- konektory:
  - pipes – bezstavové „rúry“, ktorými prechádzajú údaje
- príklady:
  - jednoduchý kompilátor, skripty v unix shelli, ...



Vlastnosti

- Výhody:
  - jednoduchosť - komponenty interagujú veľmi obmedzeným spôsobom
  - ľahké opakované použitie komponentov
  - ľahká paralelizovateľnosť a distribuovateľnosť
- Nevýhody:
  - obmedzená použiteľnosť tohto štýlu (je ťažké realizovať napr. interaktívne aplikácie)
  - často nízky výkon

Kedy použiť

- keď je možné vnímať systém ako sériu postupných transformácií vstupných údajov na výstupné
- keď je dôležitá integrovateľnosť existujúcich a znovupoužiteľnosť vyvinutých komponentov

Príklad č. 3: Repository

- **centrálnym prvkom je štruktúrovaný sklad údajov, s ktorým pracujú jednotliví (nezávislí) klienti**
- komponenty:
  - sklad údajov – je pasívny
  - klienti – majú navzájom nezávislé toky riadenia
- konektory:
  - dotazy a aktualizácie – ak sa odovzdáva riadenie skladu údajov, tak len za účelom vykonania príslušnej operácie (čítanie/zápis)
- príklady:
  - „klasický“ informačný systém
  - nástroje CASE založené na spoločnej databáze

Vlastnosti štýlu Repository

+ efektívny spôsob zdieľania veľkého množstva dát
+ ak ide o perzistentné údaje, dá sa centrálne riešiť zálohovanie, bezpečnosť, prístupové práva, zotavenie (repository manager)
+ ľahká integrácia nových klientov, ktorí sú schopní pracovať s údajmi v danej štruktúre
- spoločný dátový model môže byť príliš zložitý
- zmeny v modeli sú ťažko realizovateľné
- jednotliví klienti môžu mať rôzne požiadavky na zálohovanie, bezpečnosť, zotavenie …
- nie je jednoduché distribuovať repository na viacero počítačov

Kombinované štýly

Systémy málokedy obsahujú jeden architektonický štýl, štýly sú často kombinované:

- jednotlivé časti systému používajú rôzne štýly
- na rôznych úrovniach dekompozície sa používajú rôzne štýly
- štruktúra systému sa dá popísať viacerými štýlmi naraz, napr.:
  - layered (je organizovaný po vrstvách)
  - repository (najspodnejšou vrstvou je spoločná db)
  - client/server (jednotlivé vrstvy sú realizované v samostatných procesoch, ktoré komunikujú štýlom klient-server)

Návrh dátového modelu: návrhové vzory

Co to je návrhový vzor

Návrhový vzor vysvětluje a popisuje nějaký dílčí problém se strukturou objektů. Každý návrhový vzor má následující strukturu:

- Název vzoru. Název je krátké nejlépe jednoslovní označení.
- Popis problému, který návrhový vzor řeší.
- Řešení, které návrhový vzor přináší. Zde se nejlépe uplatňuje kombinace diagramu s textem.
- Důsledky použití vzoru.

Design Patterns

- Creational
- Structural
- Behavioral

- Class scope
- Object scope

- Each pattern described by: Name, Intent, AKA, Motivation, Applicability, Structure, Participants, Collaborations, Consequences, Implementation, Sample Code, Known Issues, Related Patterns

- E.Gamma et al.: Design Patterns, Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995.

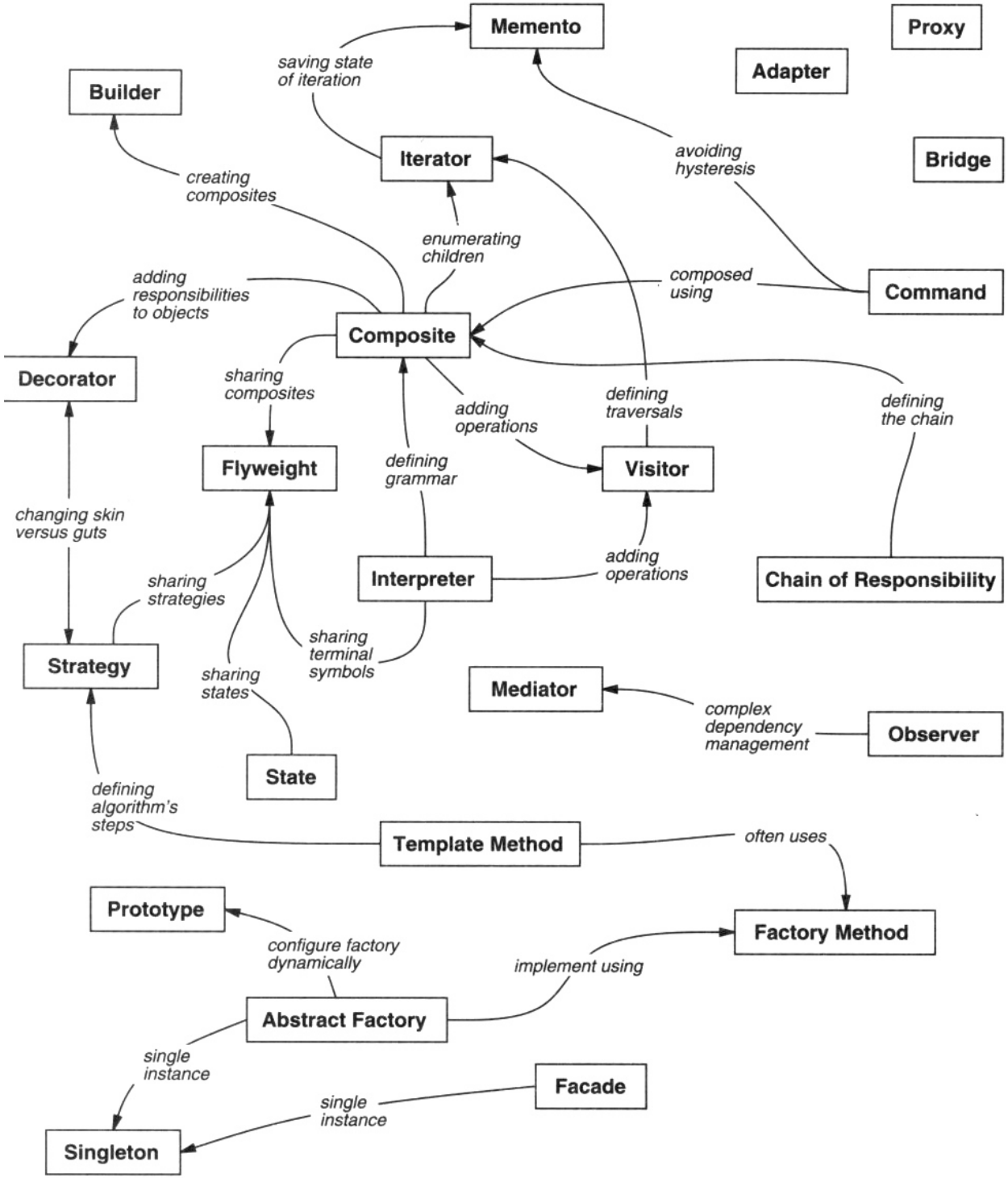


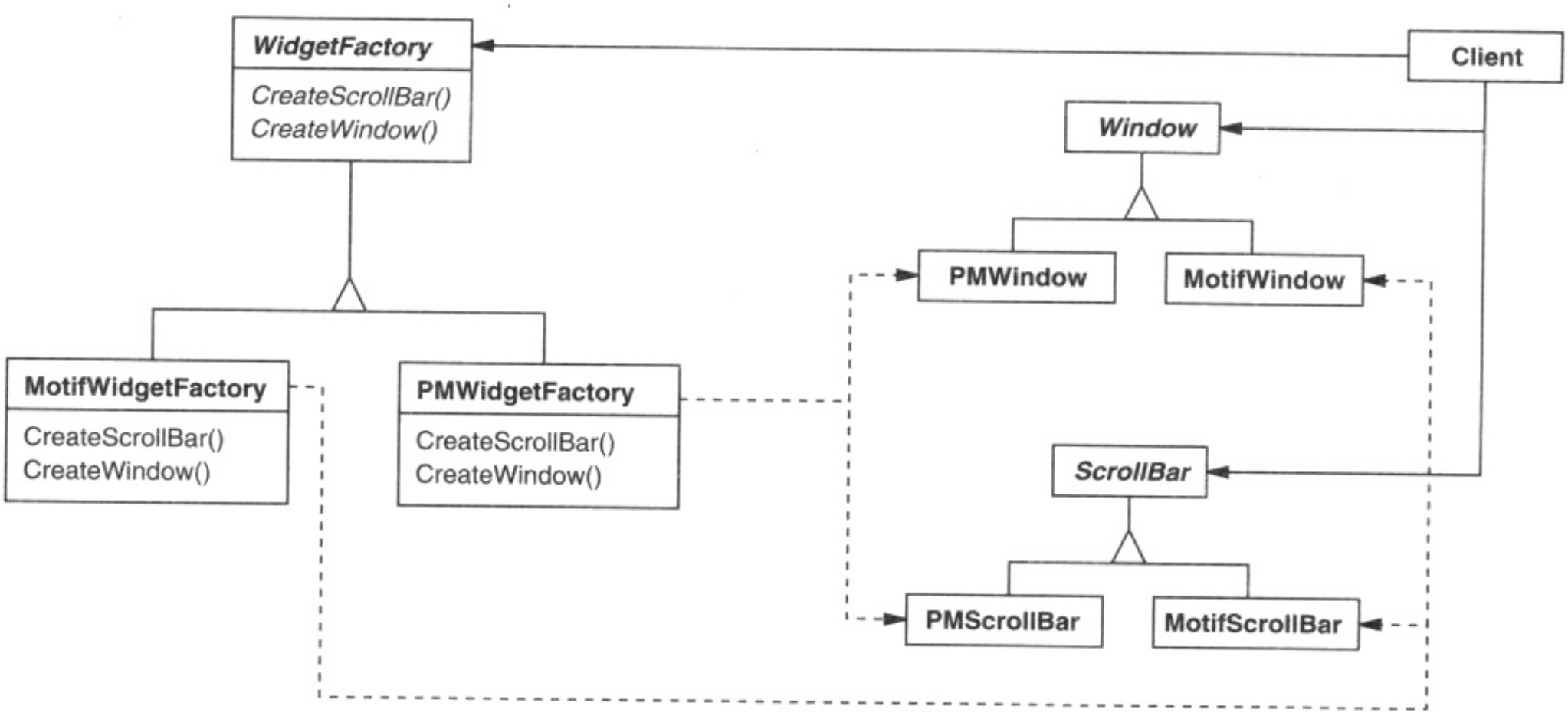
Figure 1.1: Design pattern relationships

Purpose	Design Pattern	Aspect(s) That Can Vary
Creational	Abstract Factory (87)	families of product objects
	Builder (97)	how a composite object gets created
	Factory Method (107)	subclass of object that is instantiated
	Prototype (117)	class of object that is instantiated
	Singleton (127)	the sole instance of a class
Structural	Adapter (139)	interface to an object
	Bridge (151)	implementation of an object
	Composite (163)	structure and composition of an object
	Decorator (175)	responsibilities of an object without subclassing
	Facade (185)	interface to a subsystem
	Flyweight (195)	storage costs of objects
	Proxy (207)	how an object is accessed; its location
Behavioral	Chain of Responsibility (223)	object that can fulfill a request
	Command (233)	when and how a request is fulfilled
	Interpreter (243)	grammar and interpretation of a language
	Iterator (257)	how an aggregate's elements are accessed, traversed
	Mediator (273)	how and which objects interact with each other
	Memento (283)	what private information is stored outside an object, and when
	Observer (293)	number of objects that depend on another object; how the dependent objects stay up to date
	State (305)	states of an object
	Strategy (315)	an algorithm
	Template Method (325)	steps of an algorithm
	Visitor (331)	operations that can be applied to object(s) without changing their class(es)

Table 1.2: Design aspects that design patterns let you vary

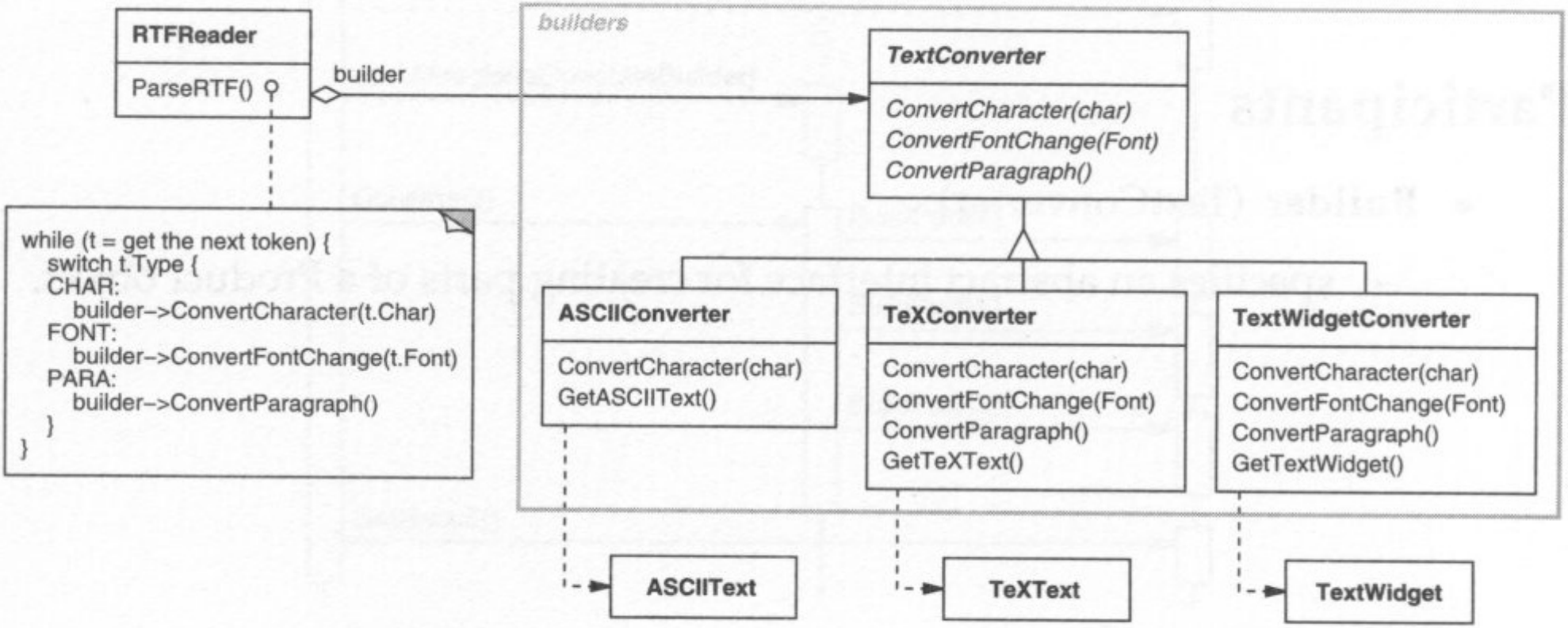
Abstract Factory

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



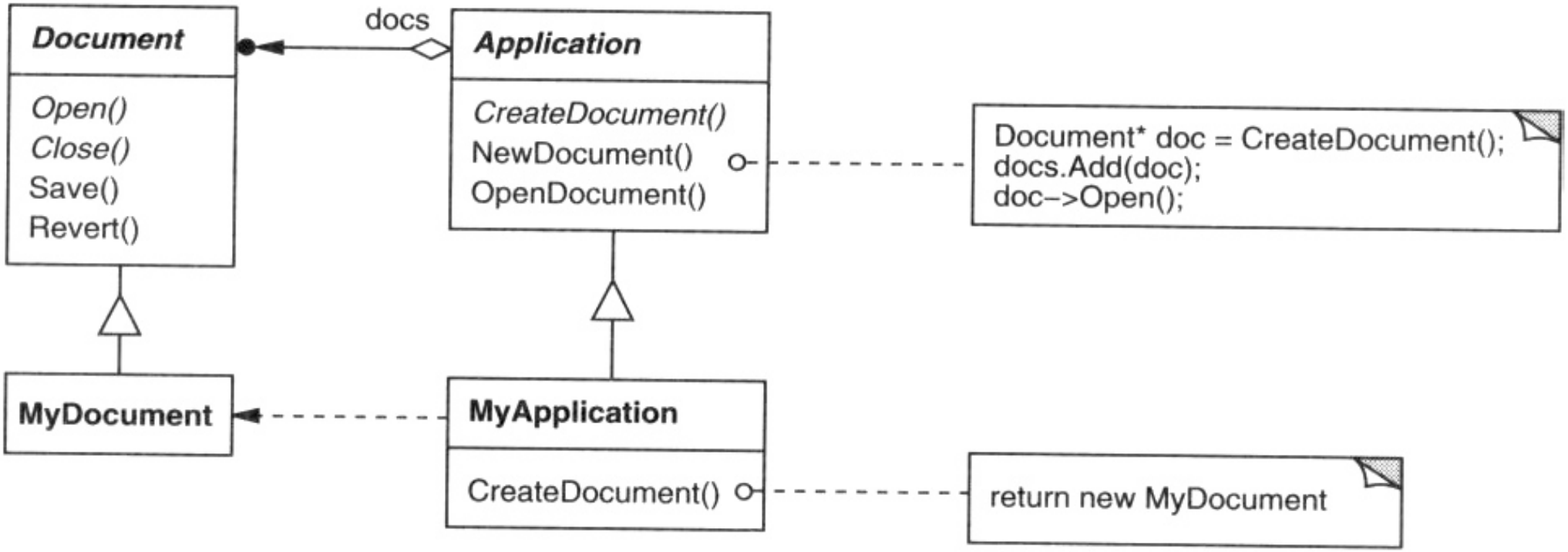
Builder

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



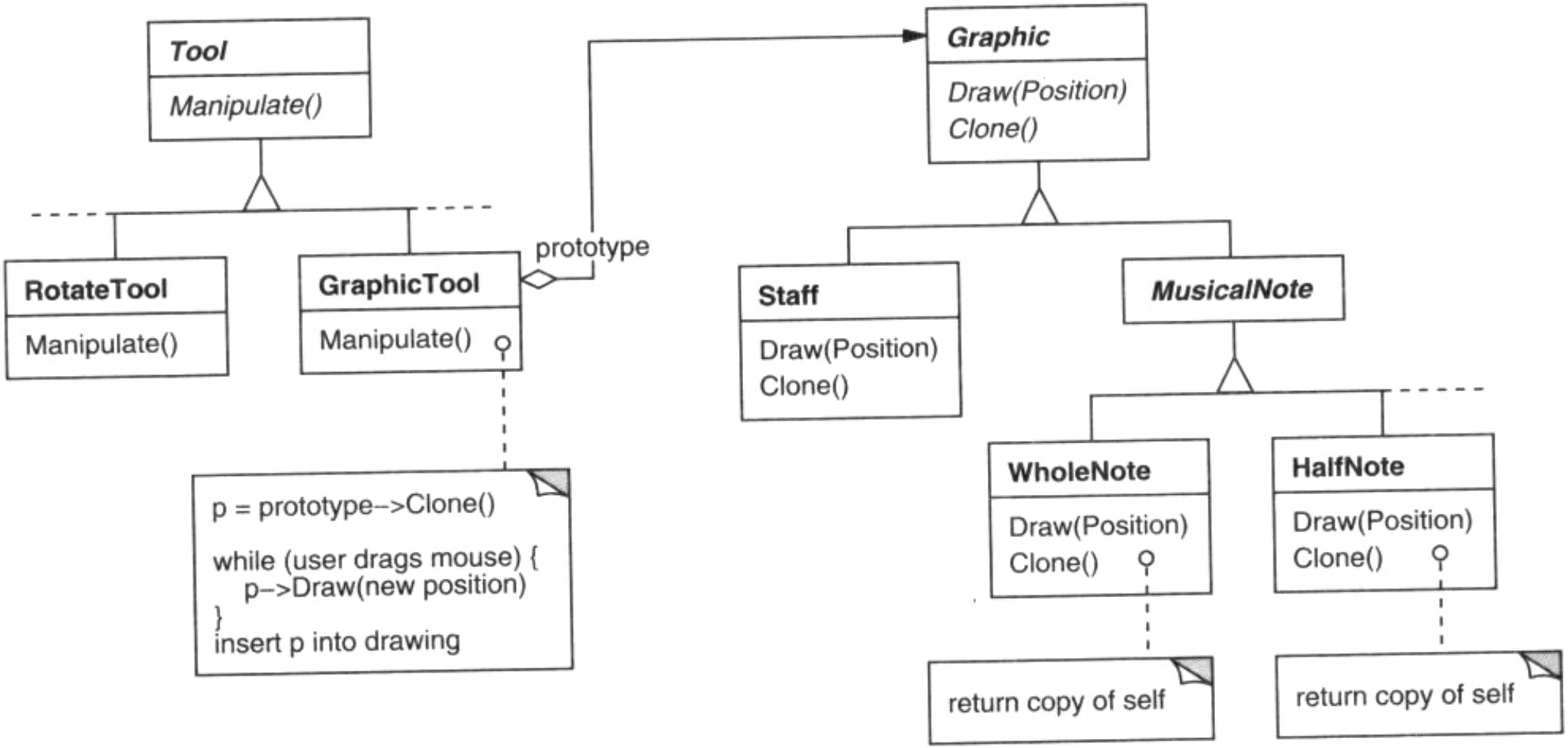
Factory Method

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



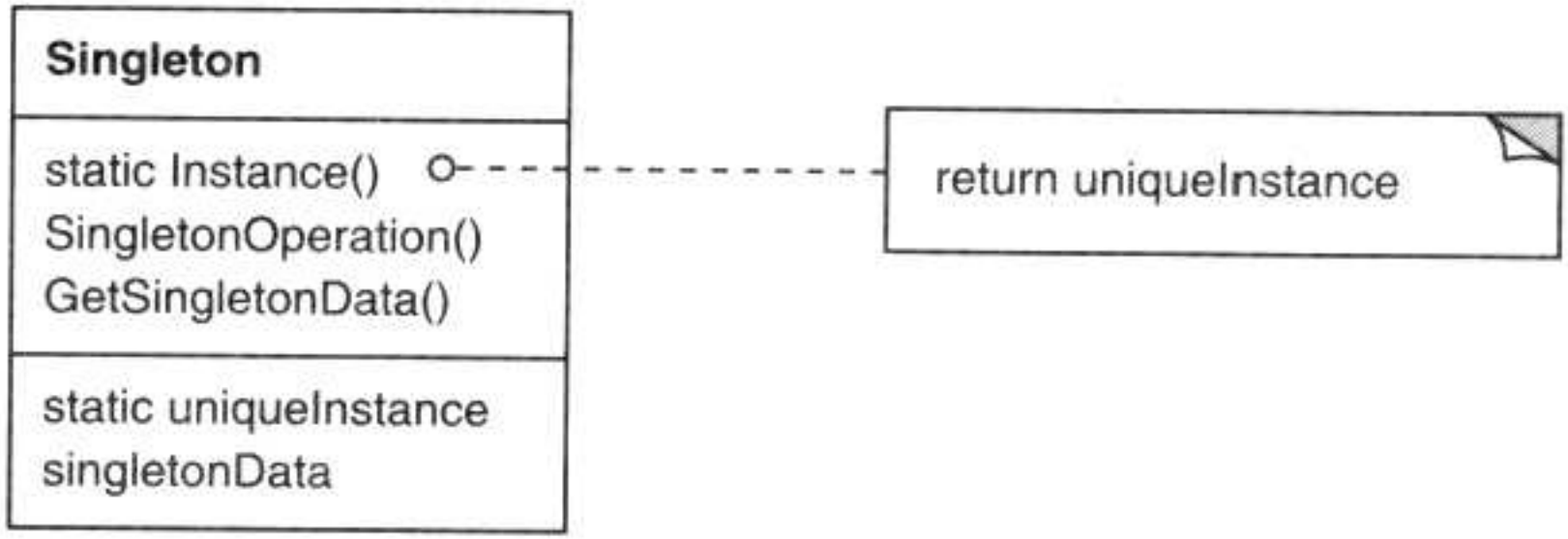
Prototype

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



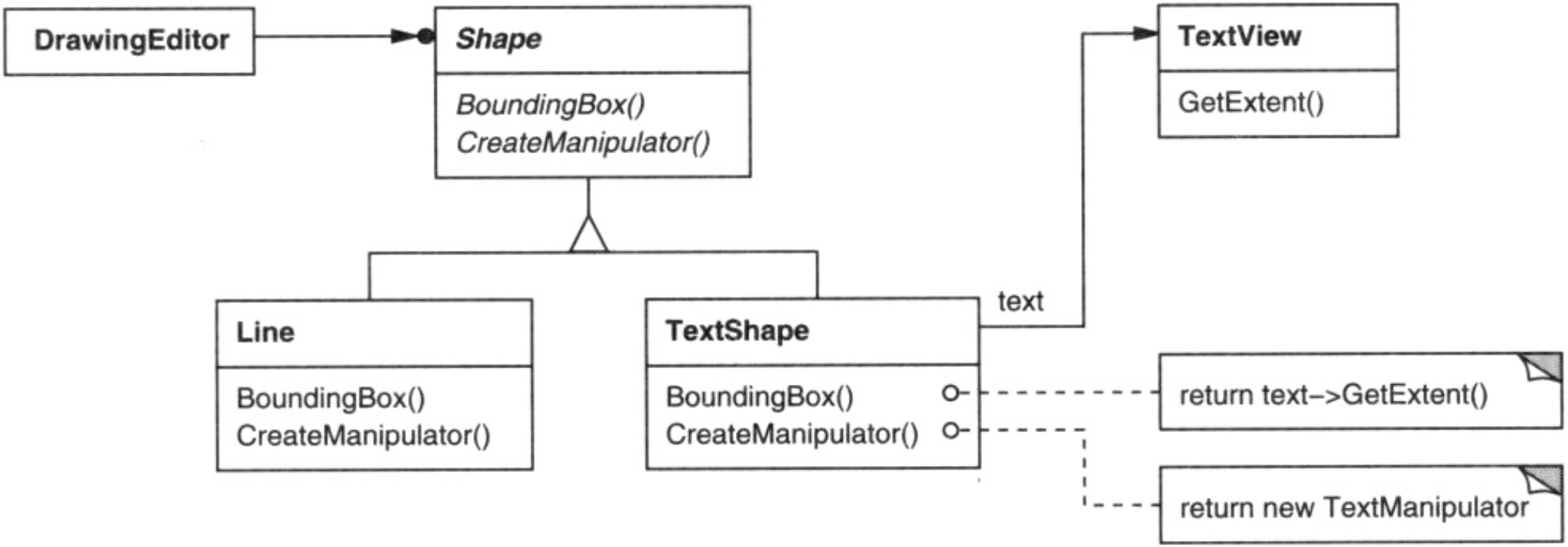
Singleton

Ensure a class only has one instance, and provide a global point of access to it.



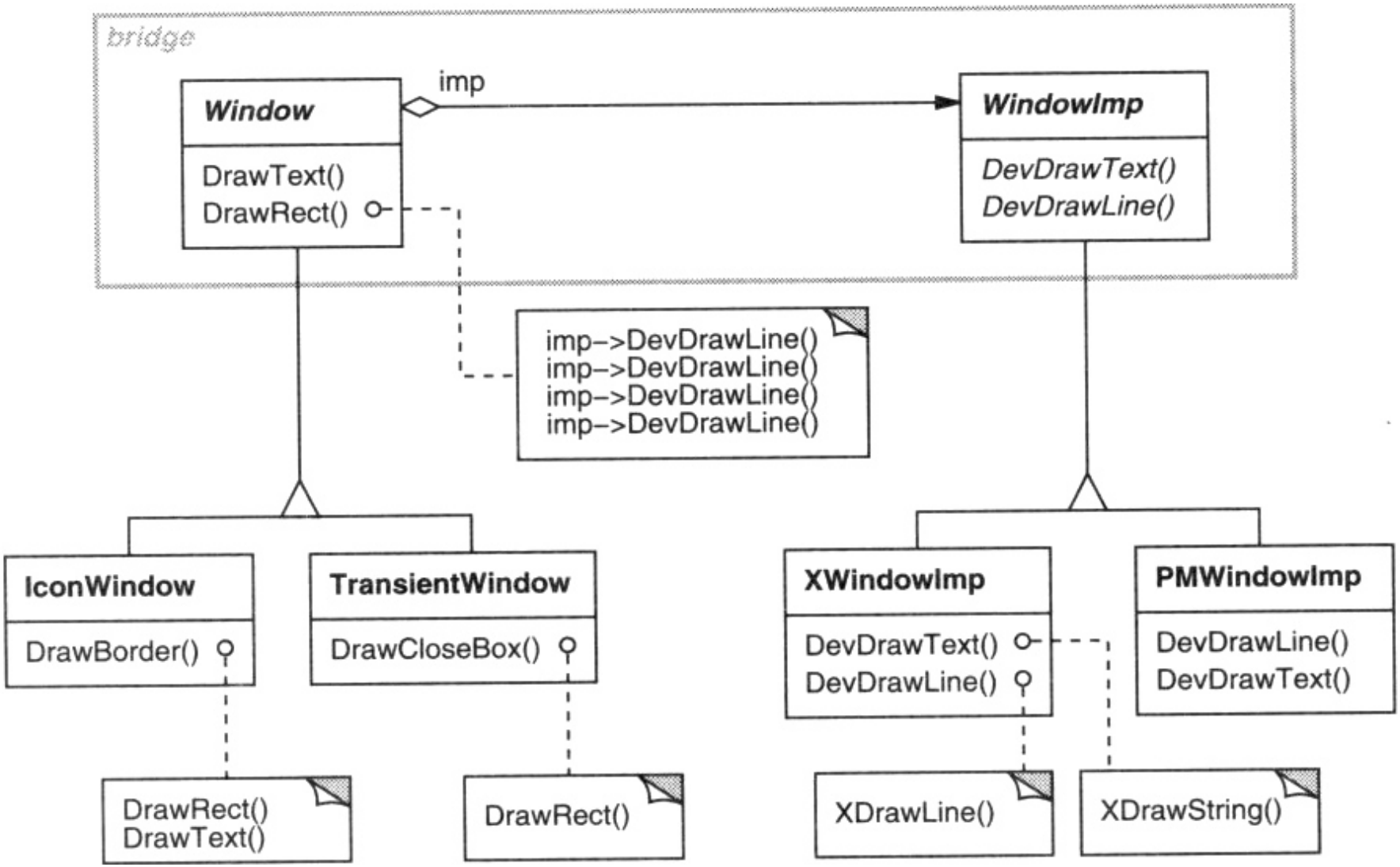
Adapter

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



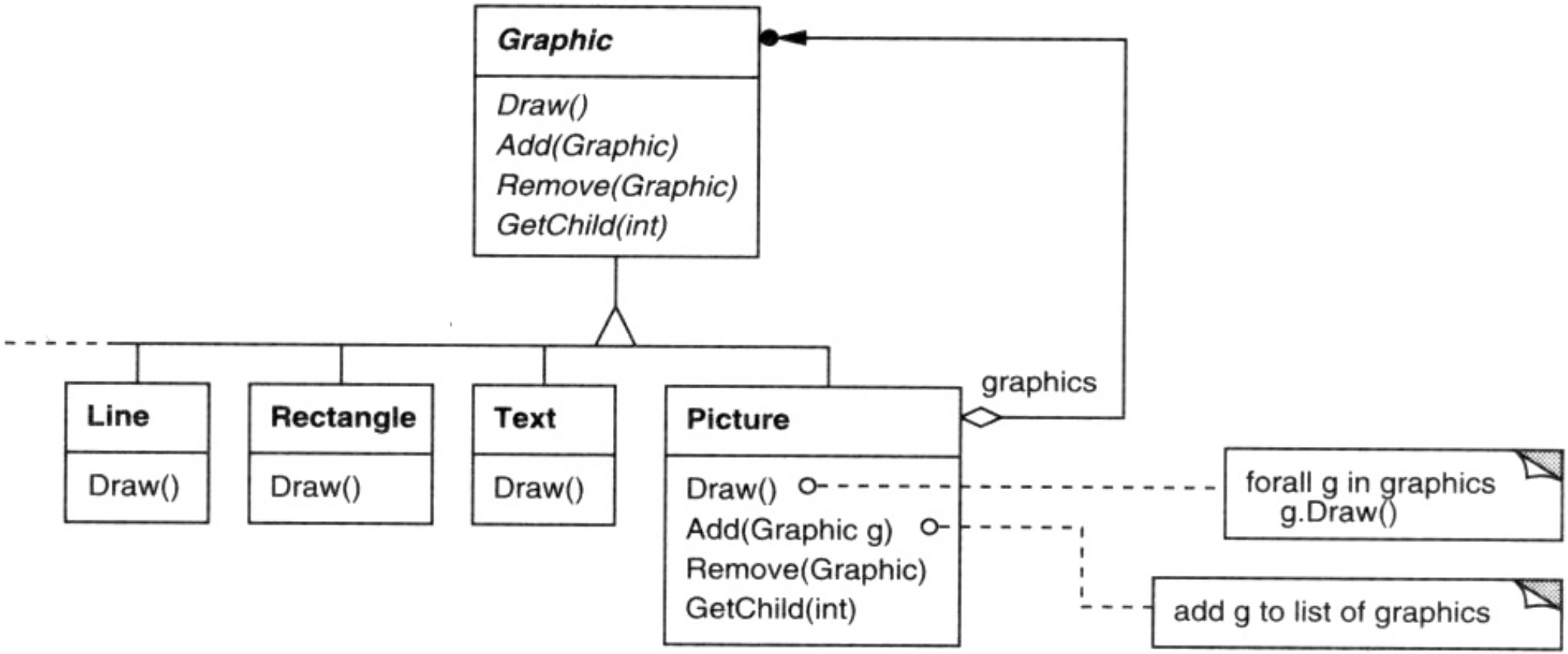
Bridge

Decouple an abstraction from its implementation so that the two can vary independently.



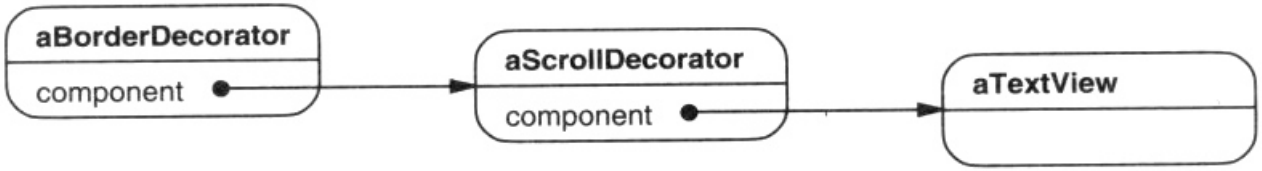
Composite

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

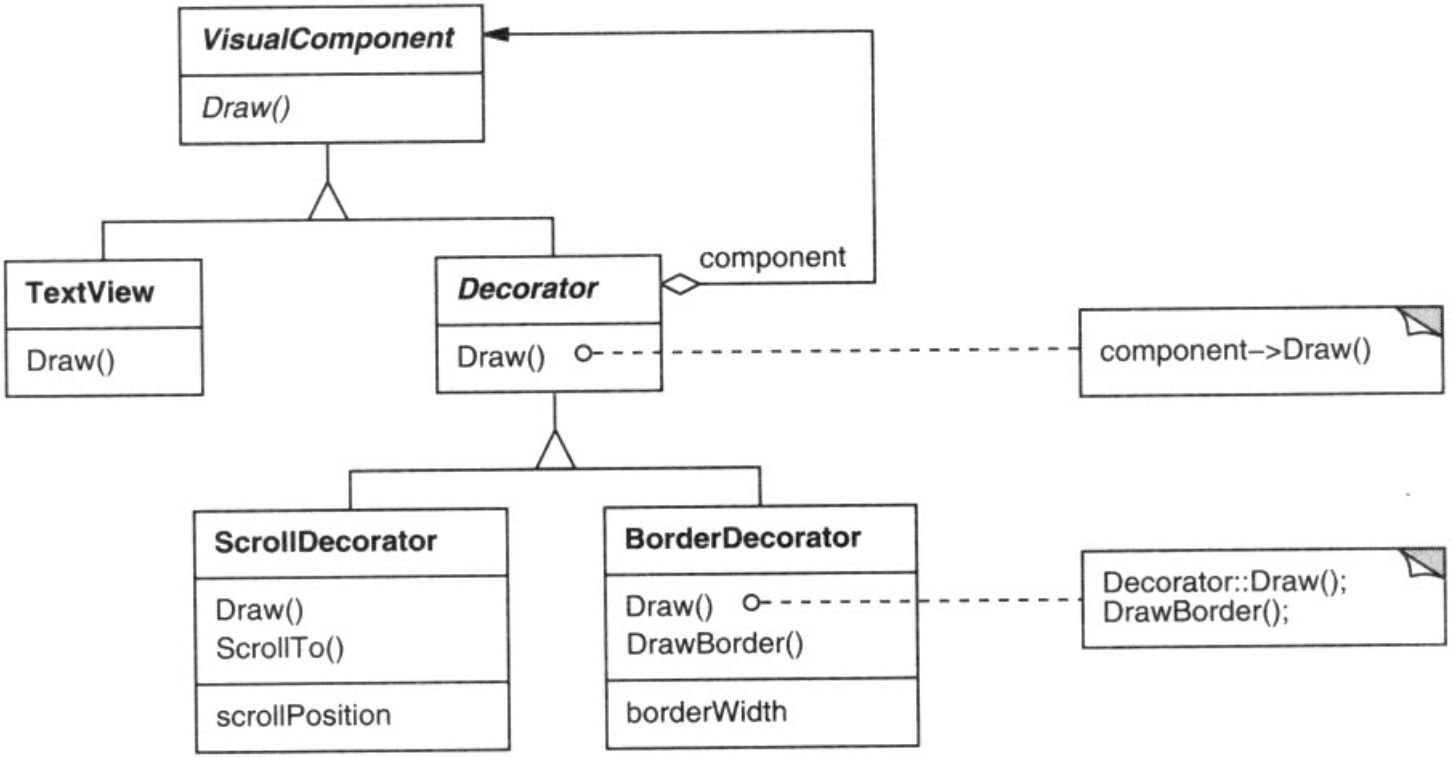


Decorator

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

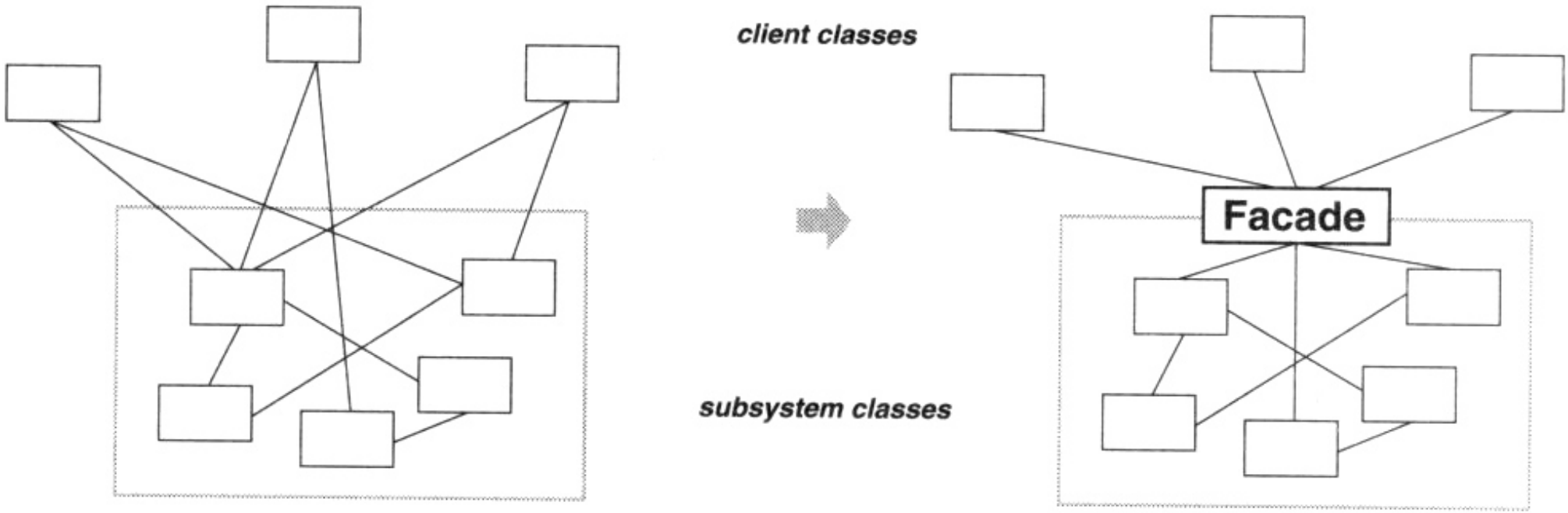


The ScrollDecorator and BorderDecorator classes are subclasses of Decorator, an abstract class for visual components that decorate other visual components.



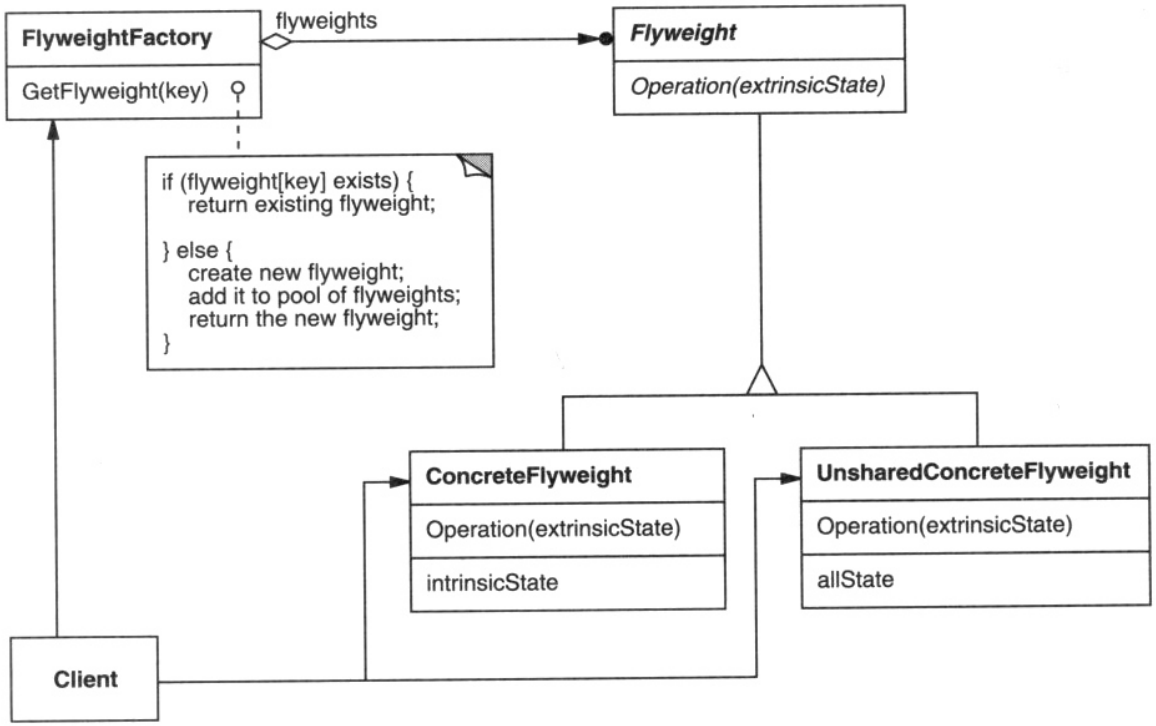
Facade

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.

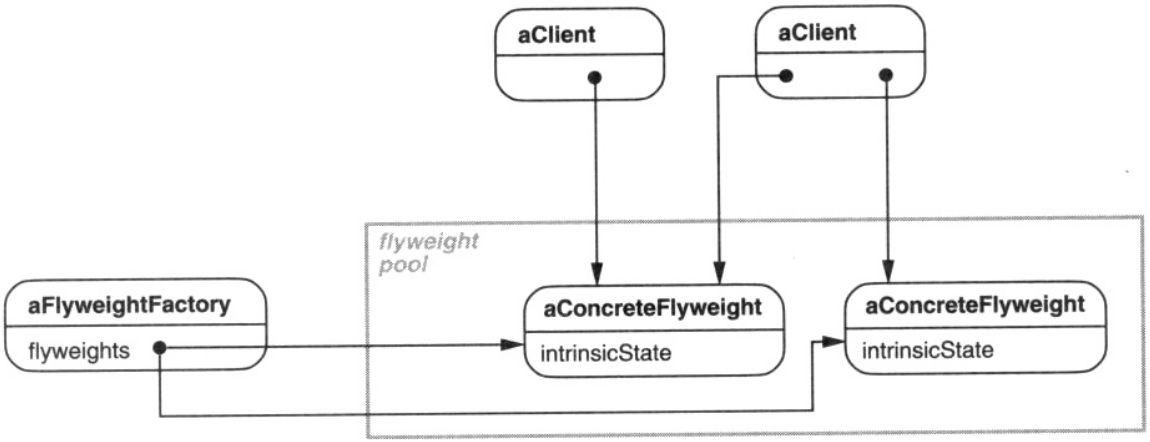


Flyweight

Use sharing to support large numbers of fine-grained objects efficiently.

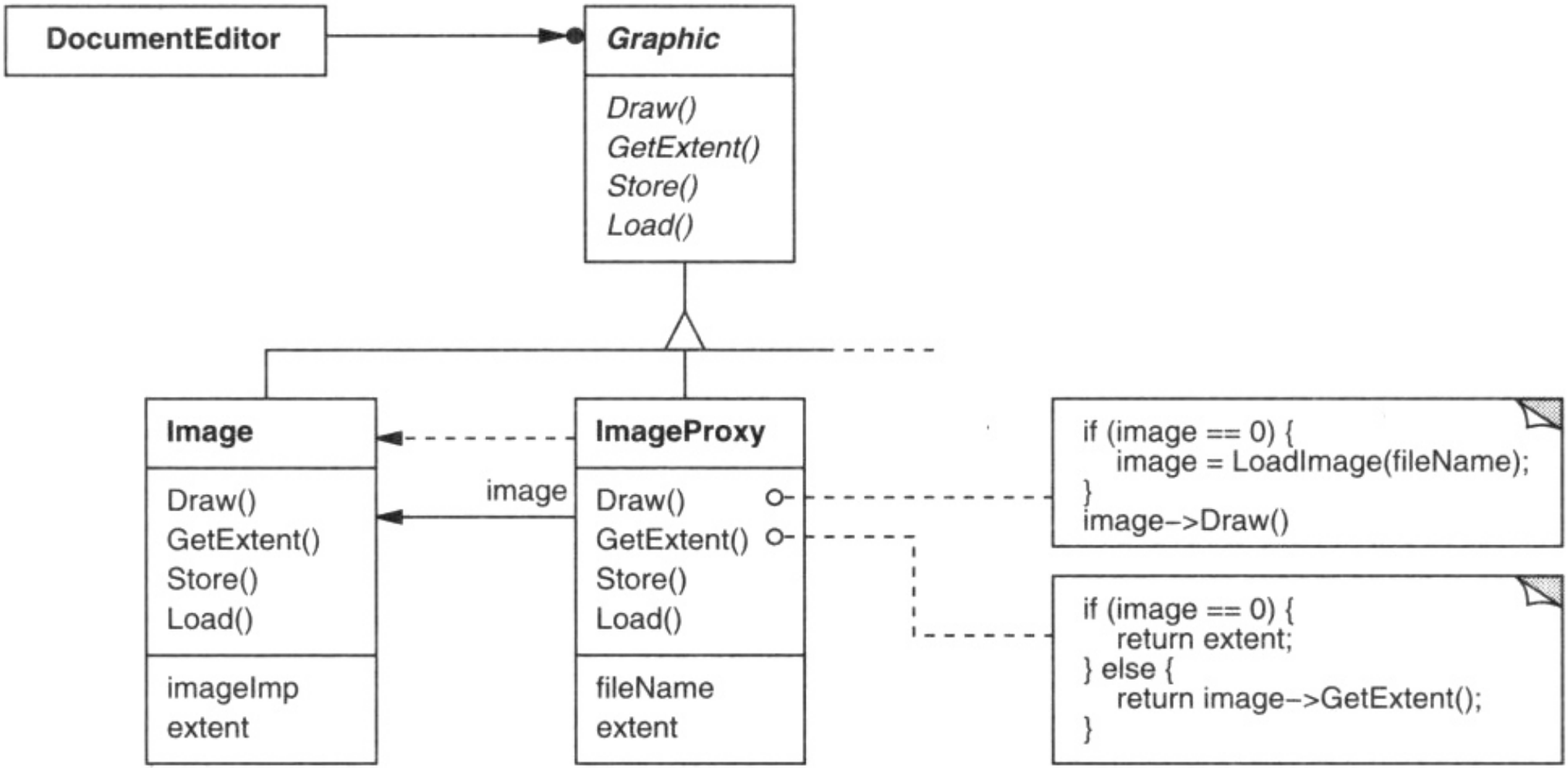


The following object diagram shows how flyweights are shared:



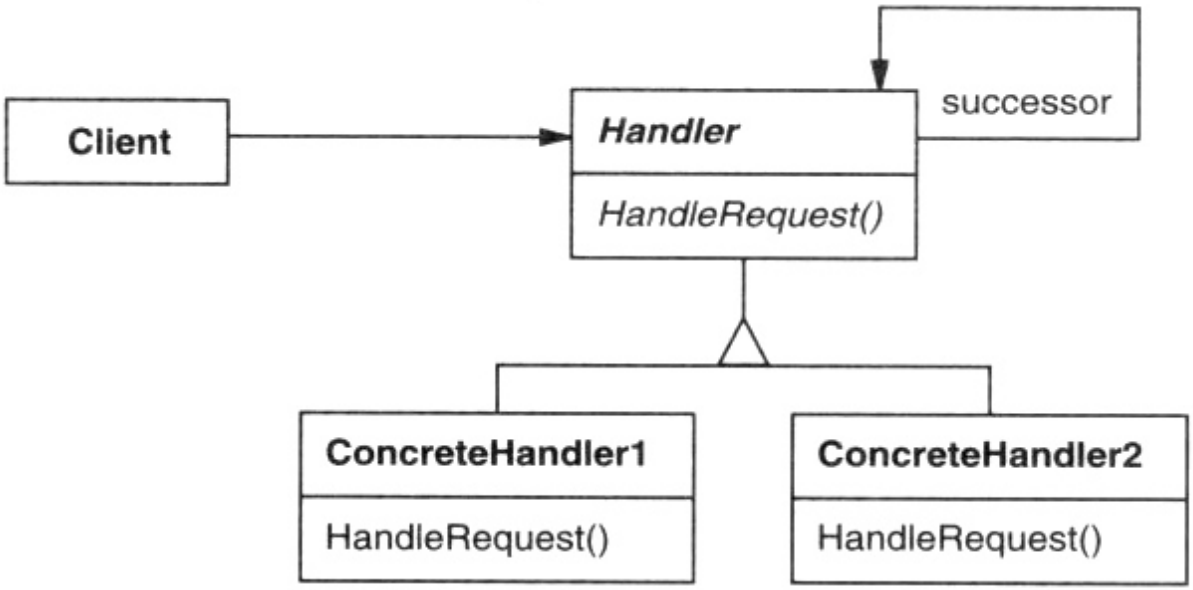
Proxy

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

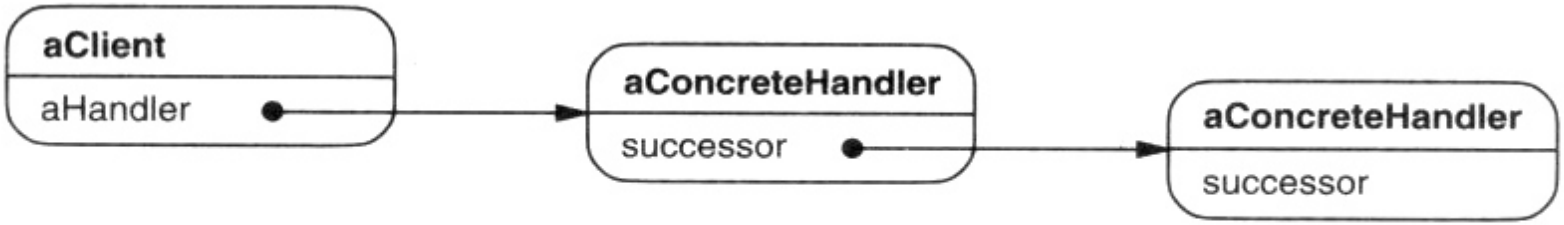


Chain of Responsibility

Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.

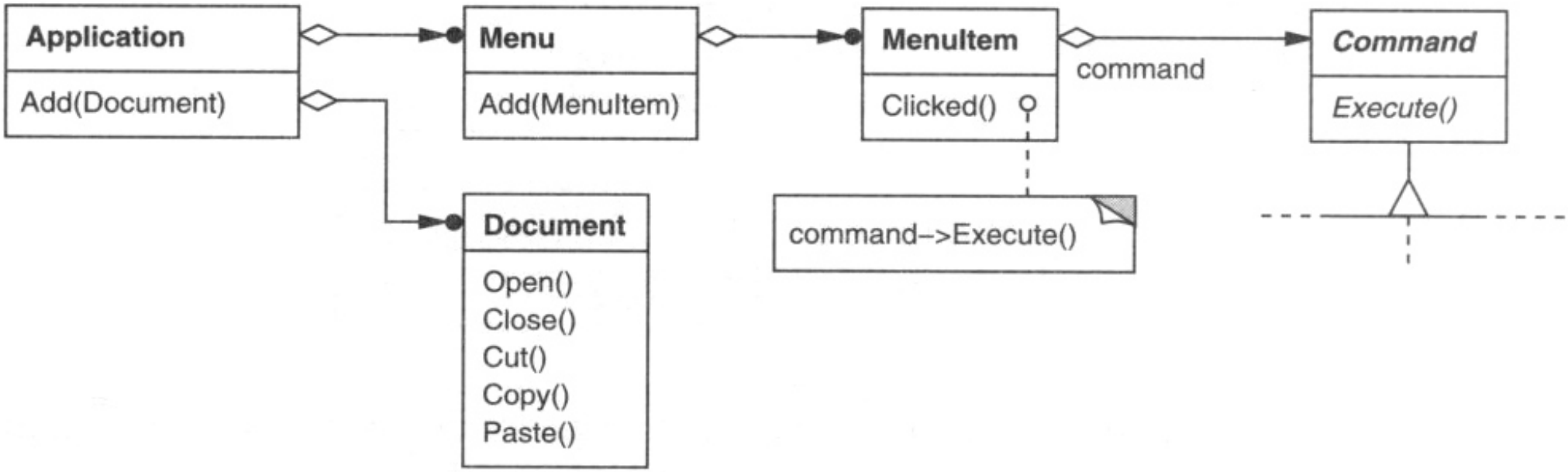


A typical object structure might look like this:



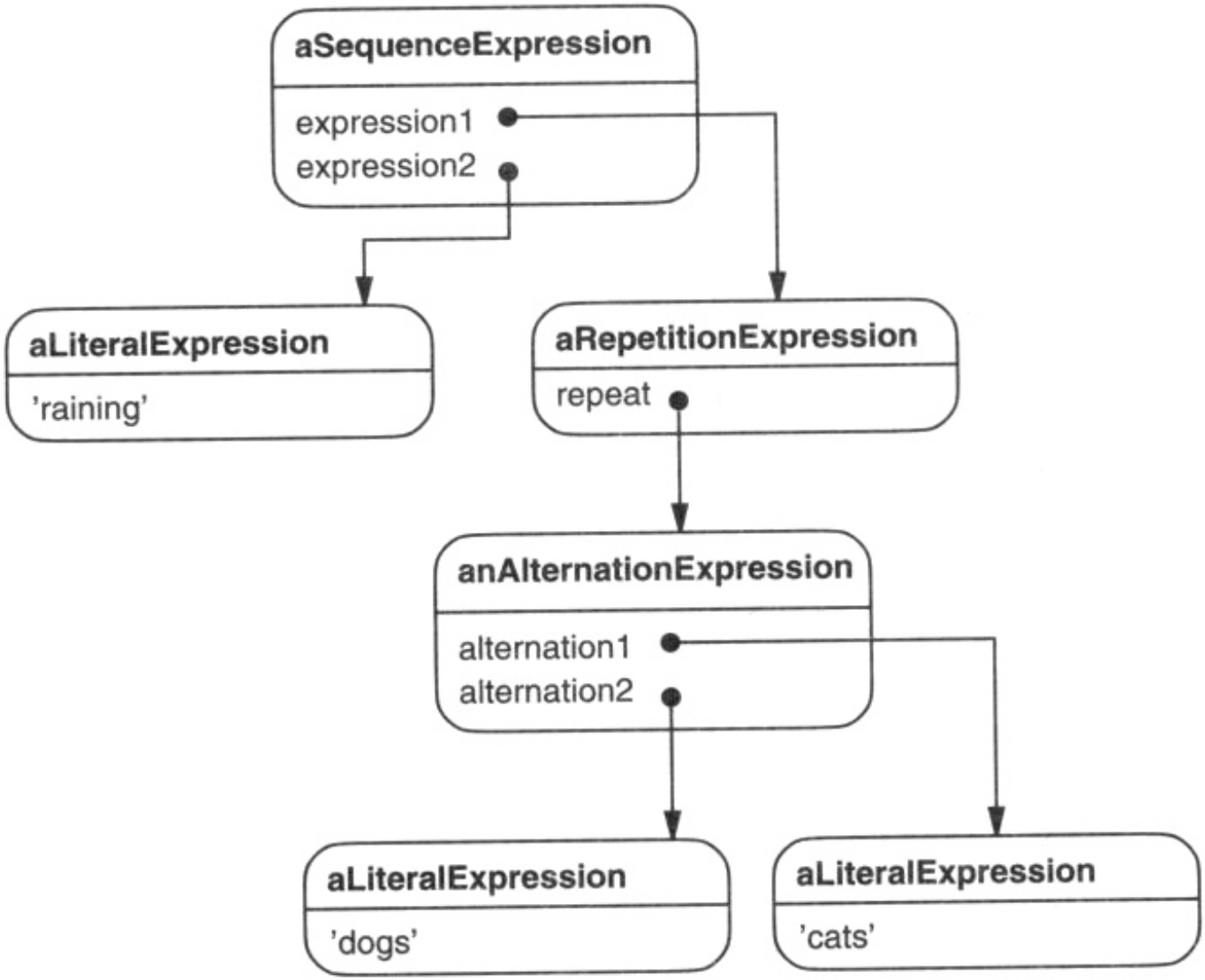
Command

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



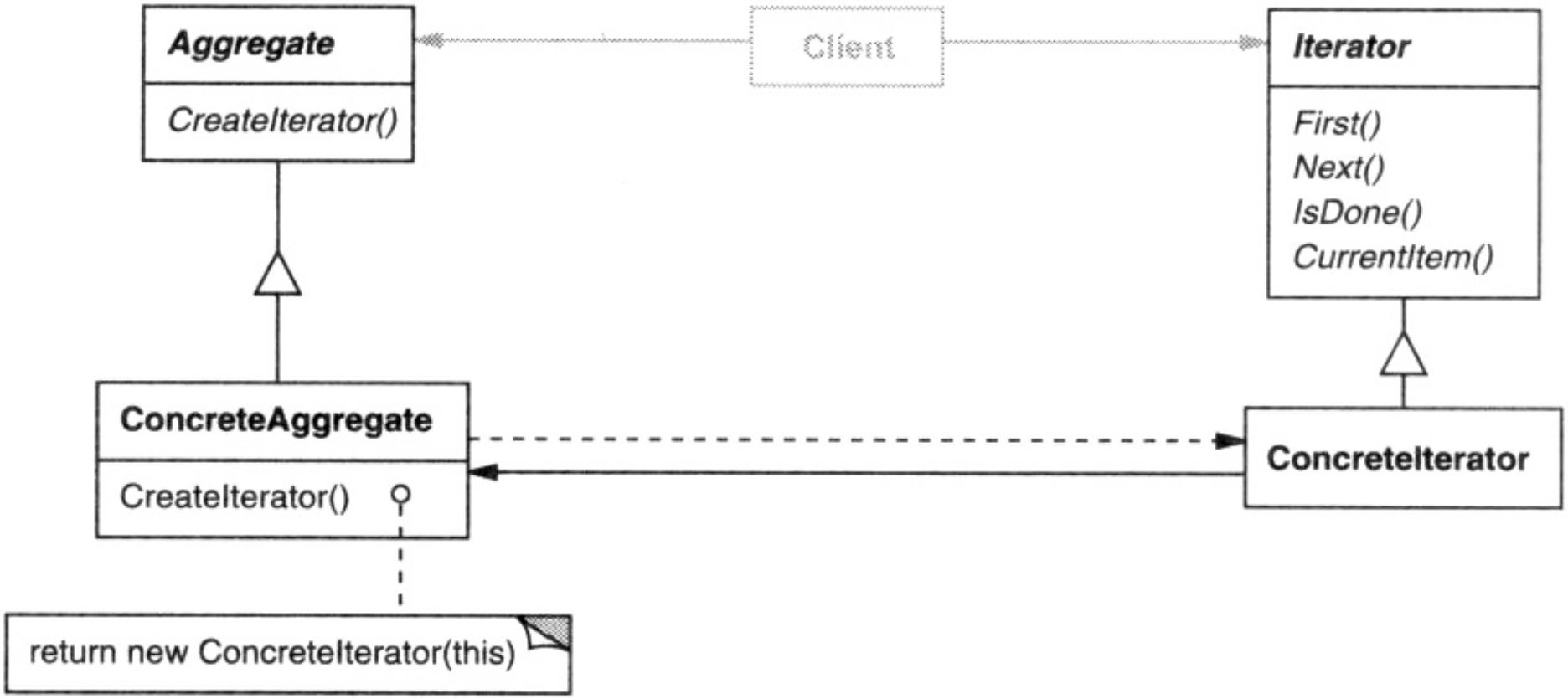
Interpreter

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



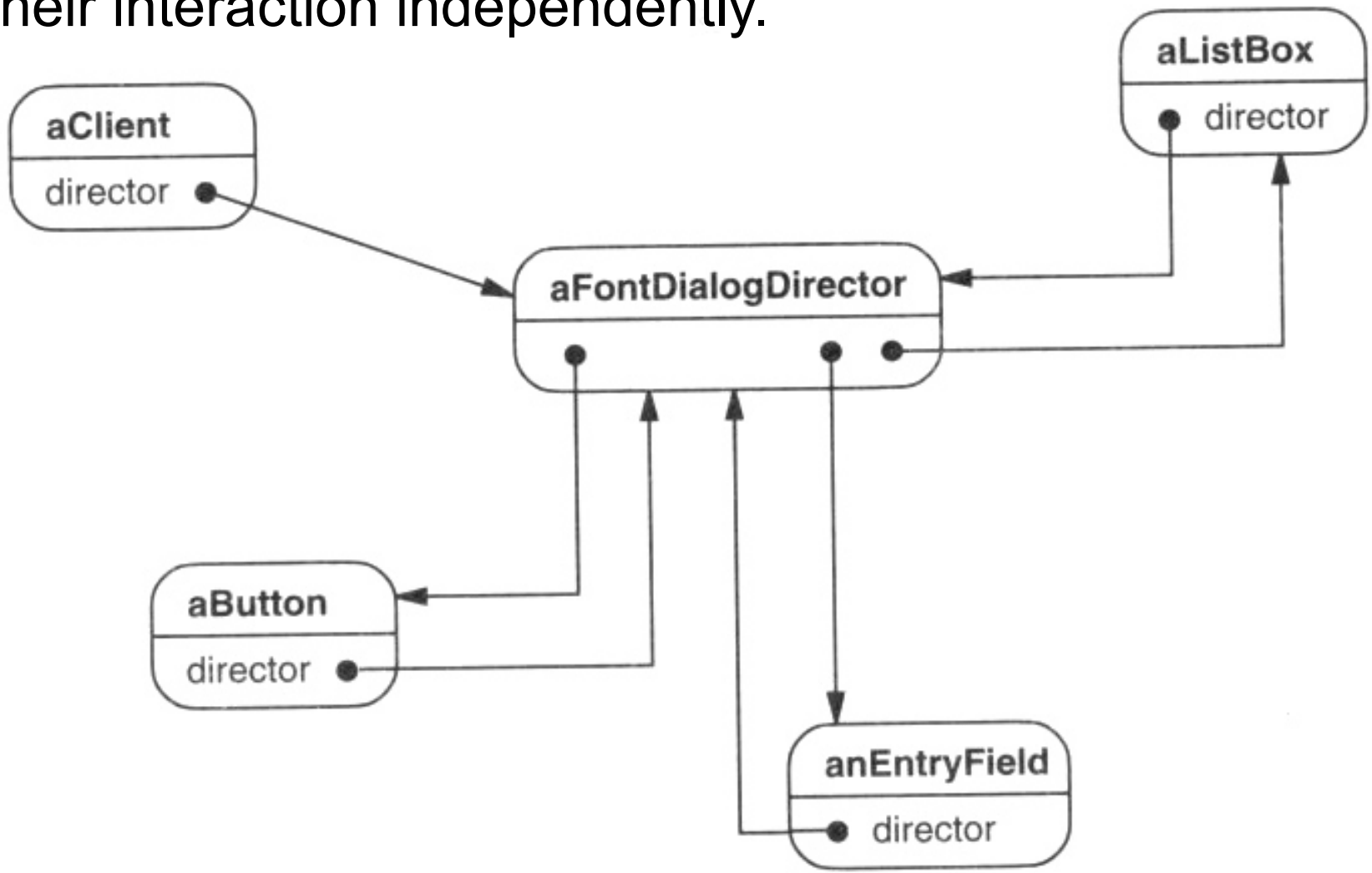
Iterator

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



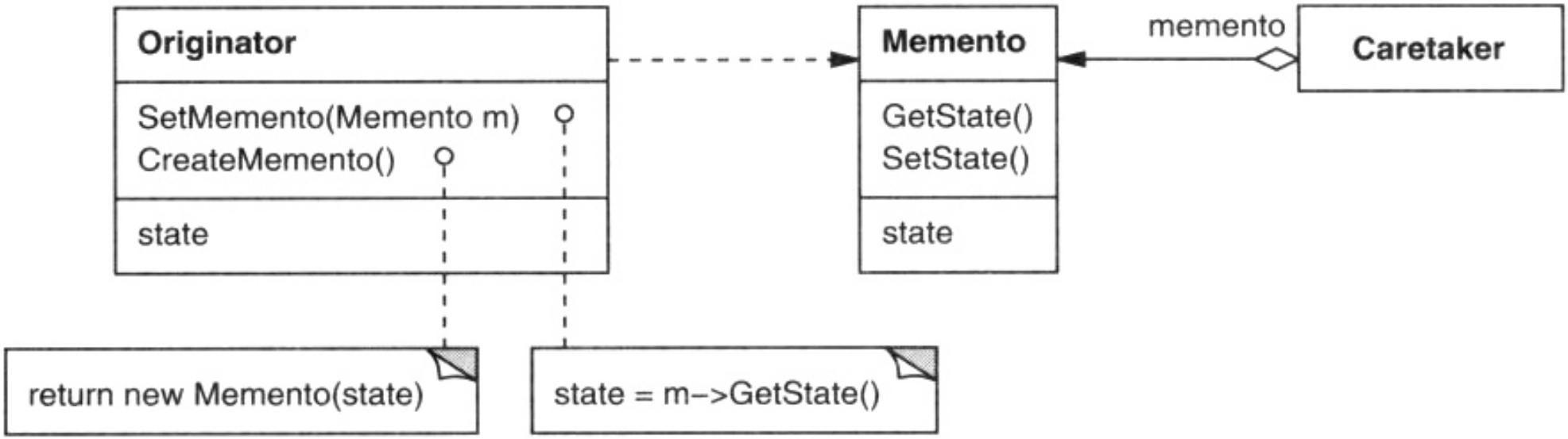
Mediator

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 it lets you vary their interaction independently.



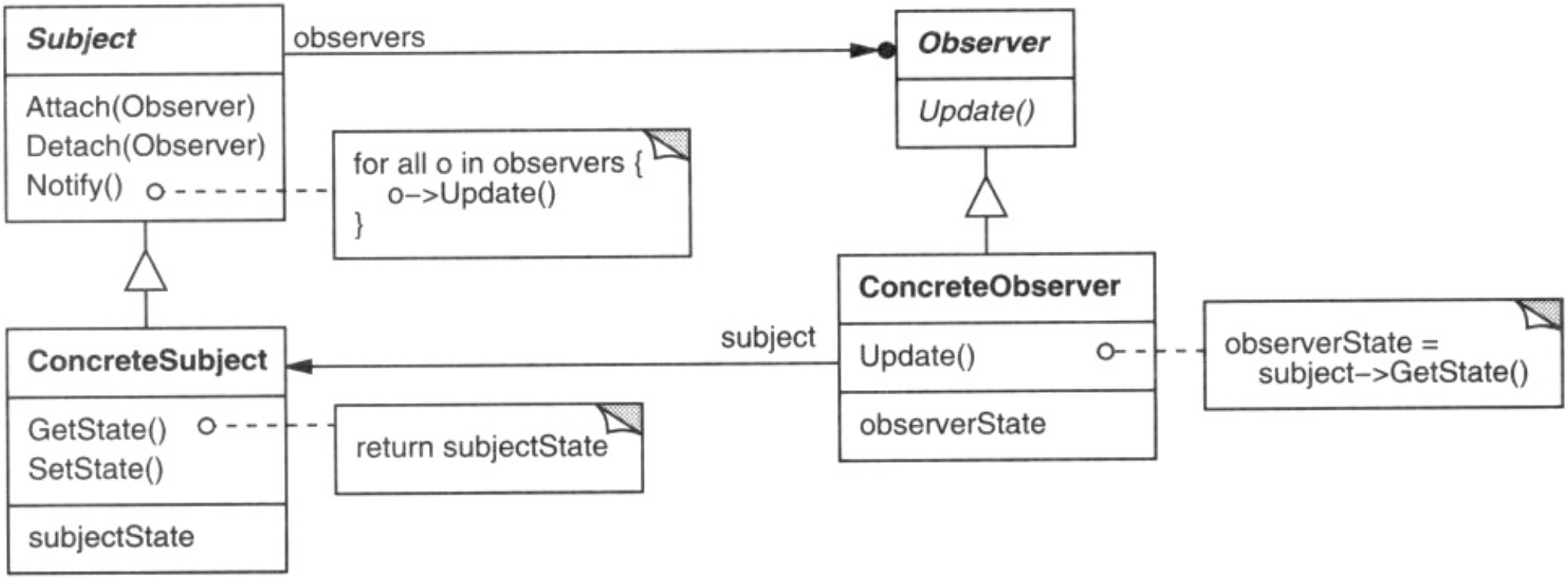
Memento

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



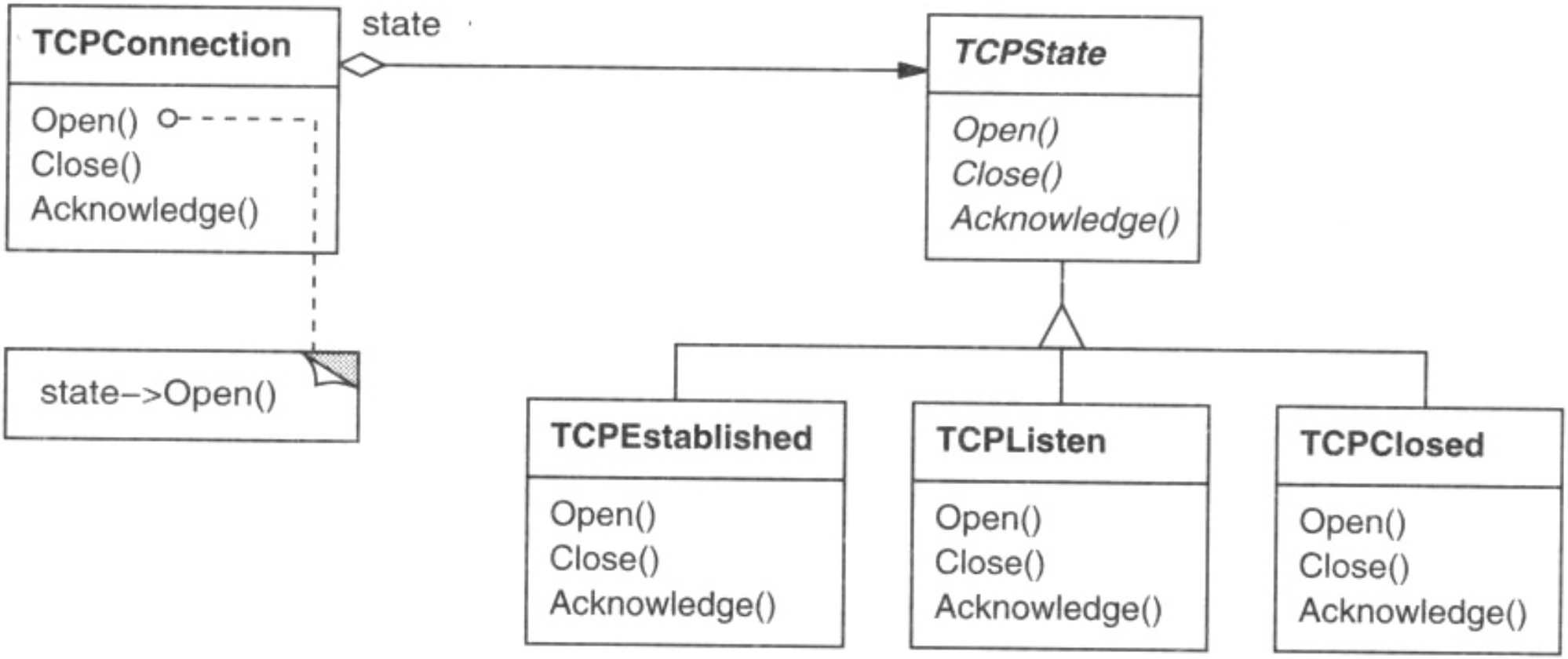
Observer

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



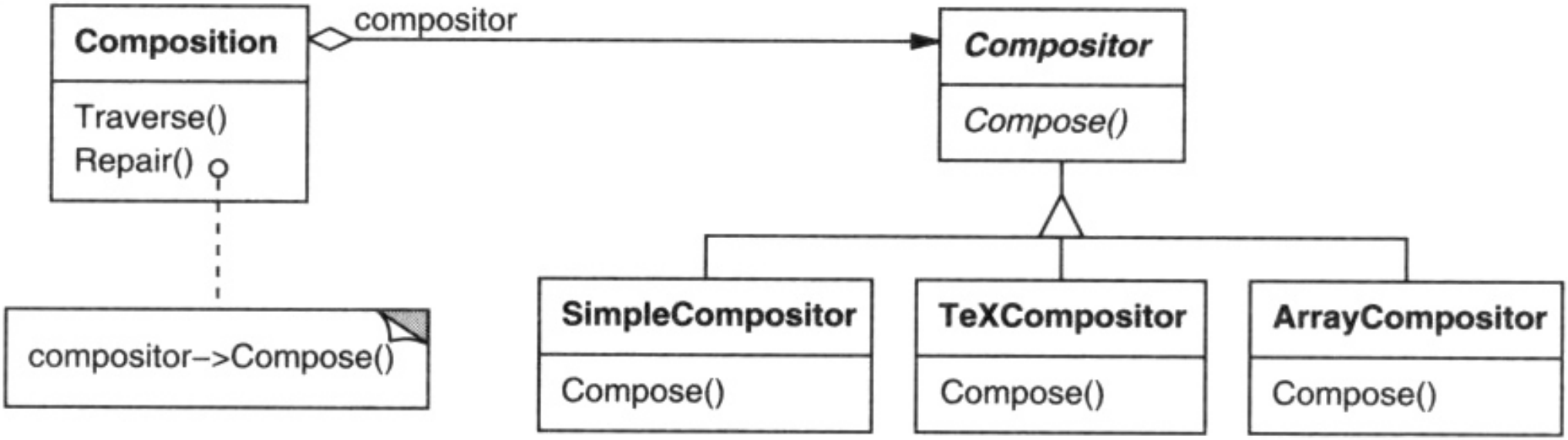
State

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



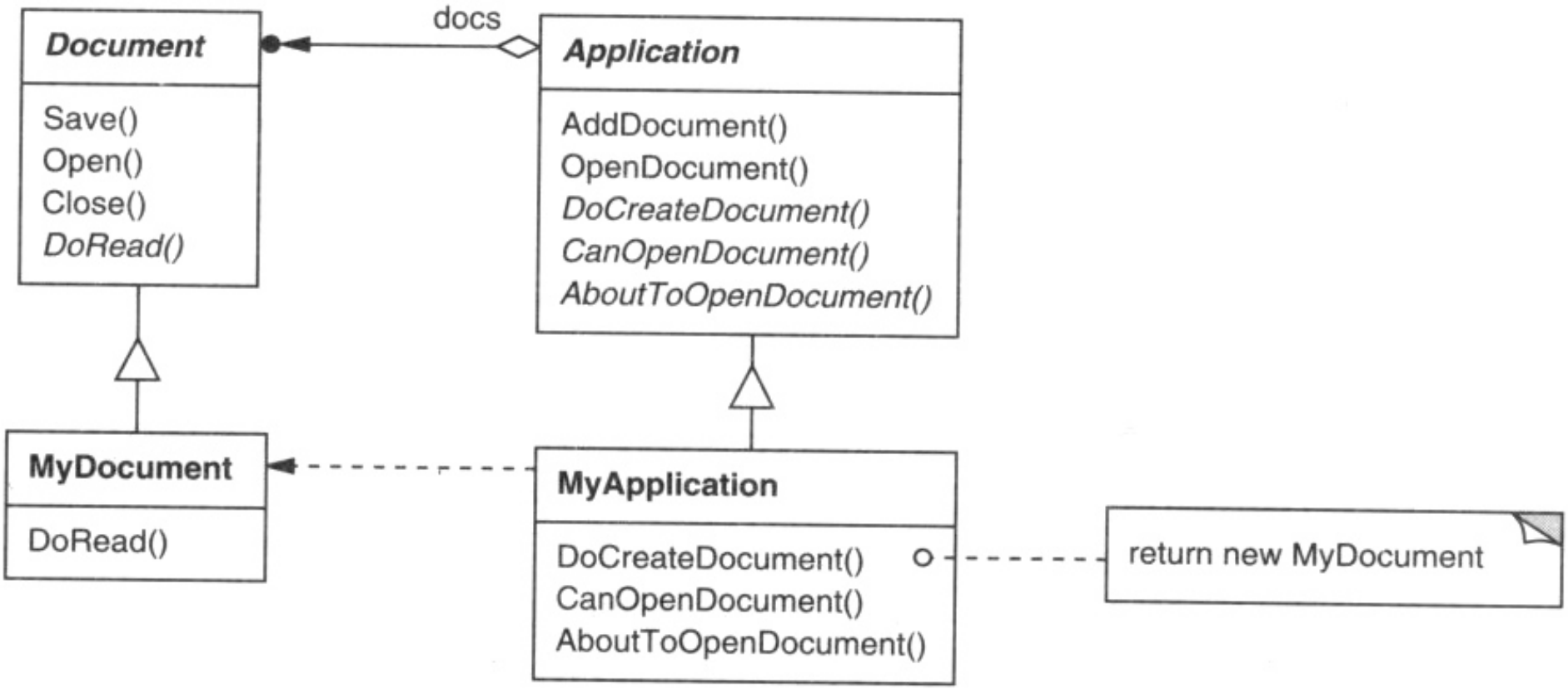
Strategy

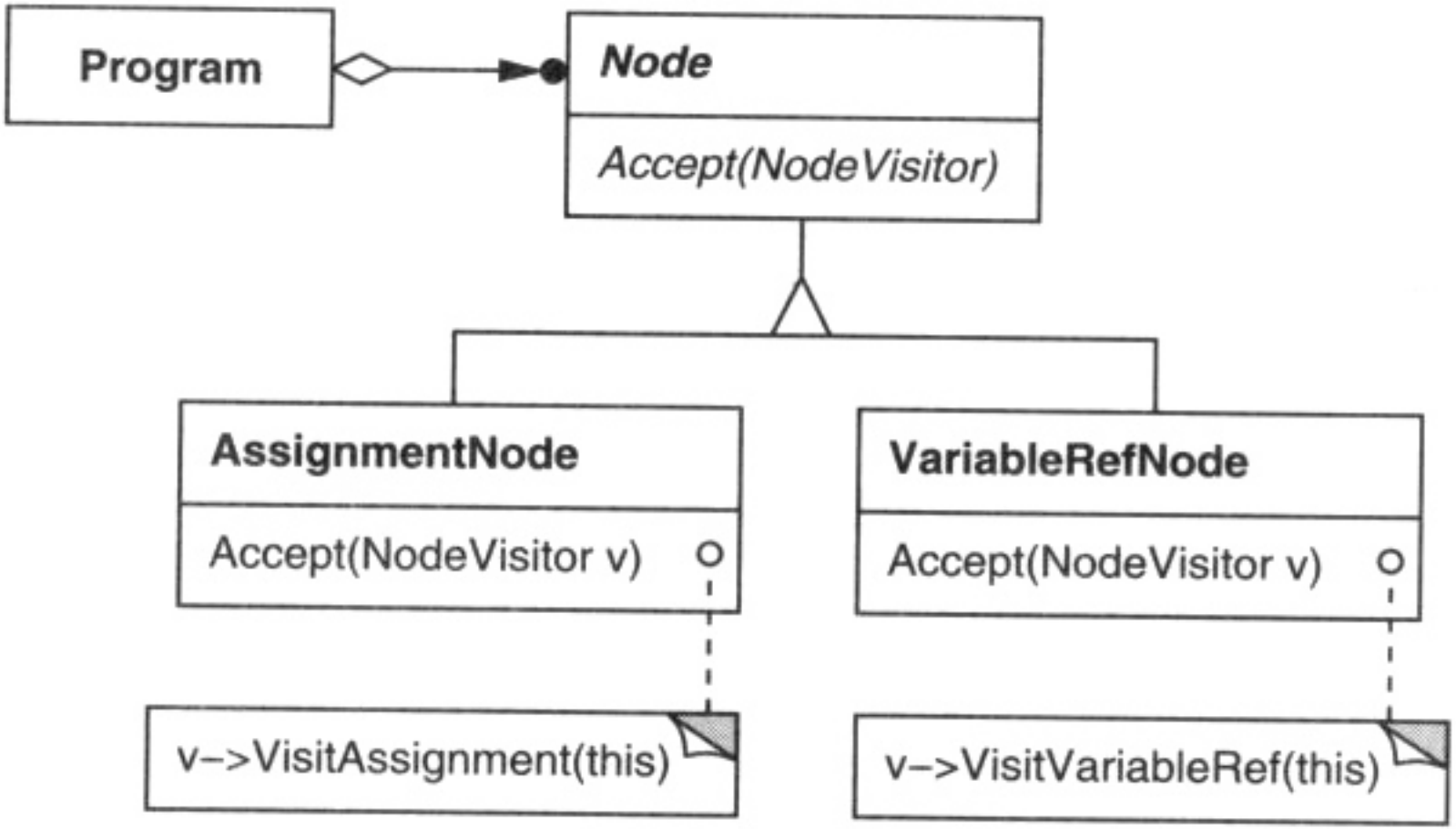
Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.



Template Method

Define a 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.



Visitor

Represent the operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the lements on which it operates.