Modellistica Medica

Maria Grazia Pia

INFN Genova

Scuola di Specializzazione in Fisica Sanitaria
Genova
Anno Accademico 2002-2003

Lezione 9

OO modeling

Design Patterns

Structural Patterns

Behavioural Patterns

Pattern classification categories

		Purpose		
		Creational	**Structural**	**Behavioral**
Scope	**Class**	Factory Method	Adapter	Interpreter Template Method
	Object	Abstract Factory Builder Prototype Singleton	Adapter Bridge Composite Decorator Facade Proxy	Chain of Responsibility Command Iterator Mediator Memento Flyweight Observer State Strategy Visitor

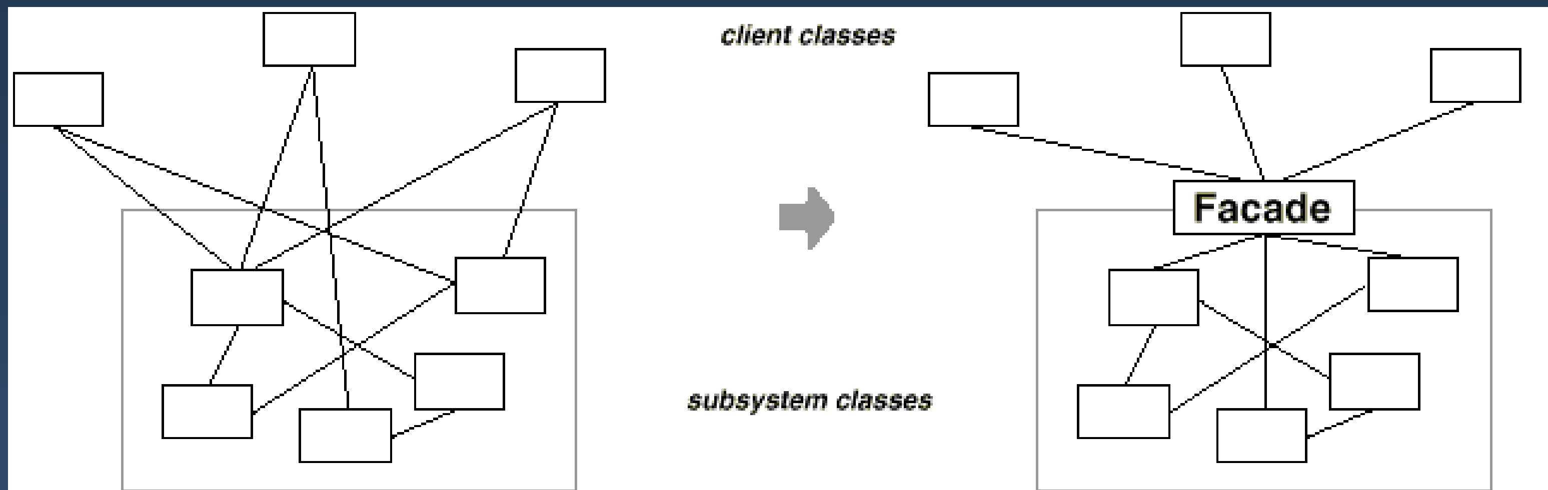
Structural Patterns

Concerned with how classes and objects are composed to form larger structures

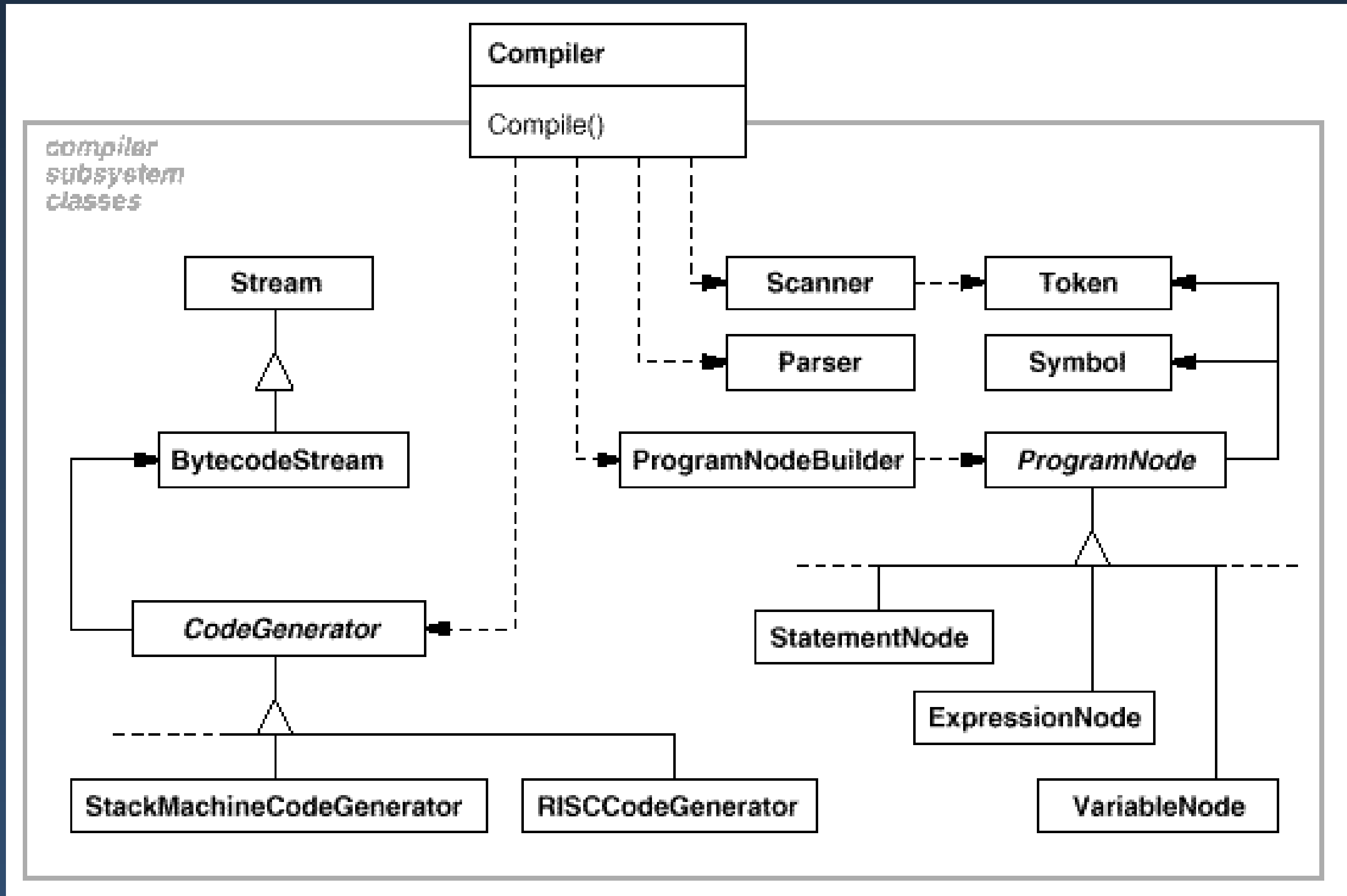
- Adapter
  - interface converter
- Bridge
  - decouple abstraction from its implementation
- Composite
  - compose objects into tree structures, treating all nodes uniformly
- Decorator
  - attach additional responsibilities dynamically
- Façade
  - provide a unified interface to a subsystem
- Flyweight
  - using sharing to support a large number of fine-grained objects efficiently
- Proxy
  - provide a surrogate for another object to control access

Façade

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



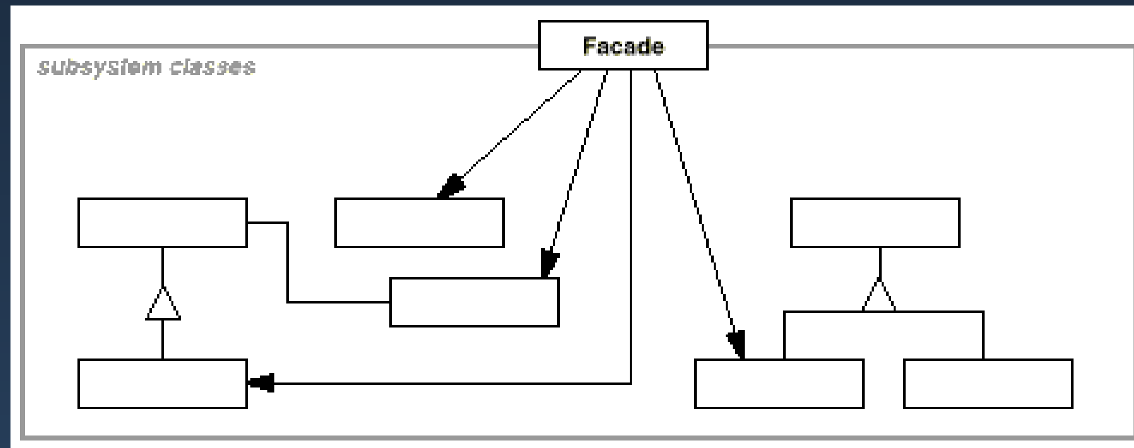
Façade



Applicability

- You want a simple interface to a complex subsystem
  - Subsystems often get more complex as they evolve
    - this makes the subsystem more reusable and easier to customize,
    - but it also becomes harder to use for clients that don't need to customize it
  - A façade can provide a simple default view of the subsystem that is good enough for most clients
    - Only clients needing more customizability will need to look beyond the façade
- There are many dependencies between clients and the implementation classes of an abstraction
  - Introduce a façade to decouple the subsystem from clients and other subsystems
- You want to layer your subsystems
  - Use a façade to define an entry point to each subsystem level

Structure



- Façade
  - knows which subsystem classes are responsible for a request
  - delegates client requests to appropriate subsystem objects

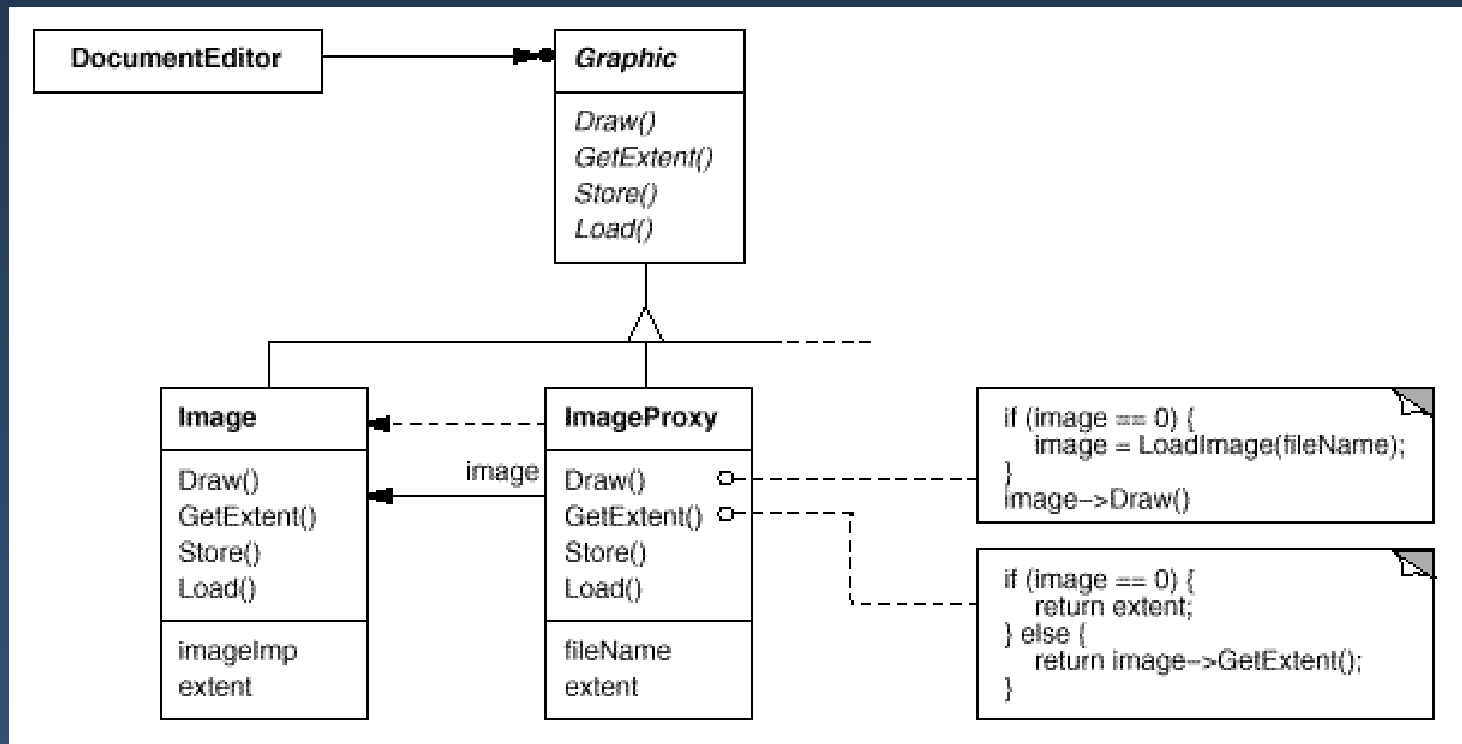
- Subsystem classes
  - implement subsystem functionality
  - handle work assigned by the Façade object
  - have no knowledge of the façade

Consequences

- Shields clients from subsystem components
  - reduces the number of objects clients see
    - easier to use subsystem
- Promotes weak coupling between the subsystem and its client
  - can vary the components of a subsystem without affecting clients
  - reduces compilation dependencies
- Does not prevent applications from using subsystem classes if they need to
  - you can choose between ease of use and generality

Proxy

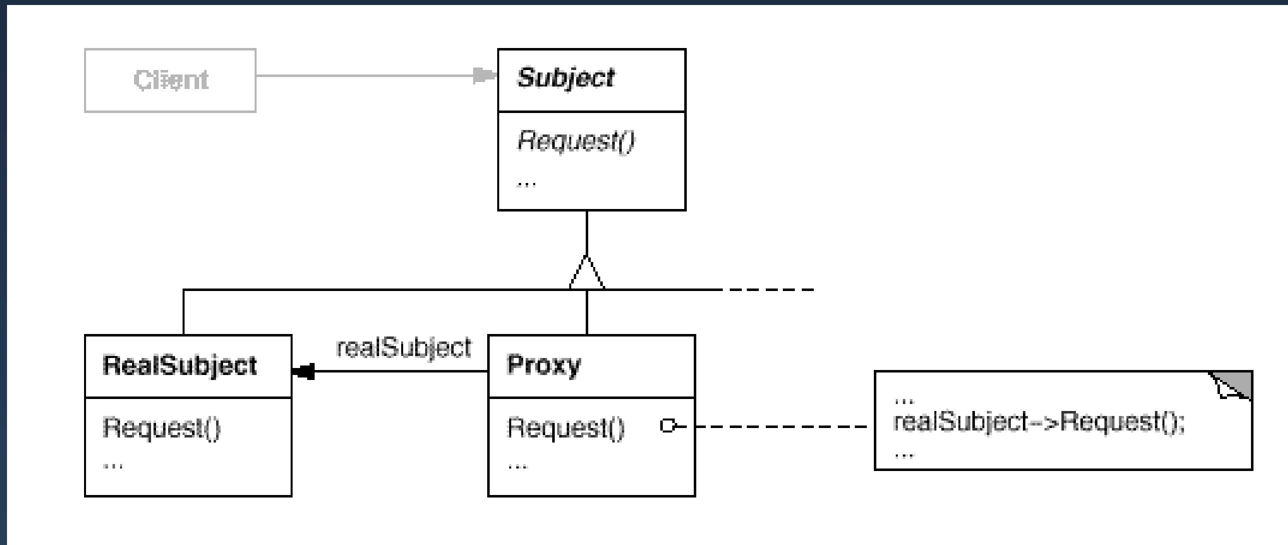
- Provide a surrogate or placeholder for another object to control access to it
  - *e.g.*, on-demand image loading
    - so that opening a document is fast



Applicability

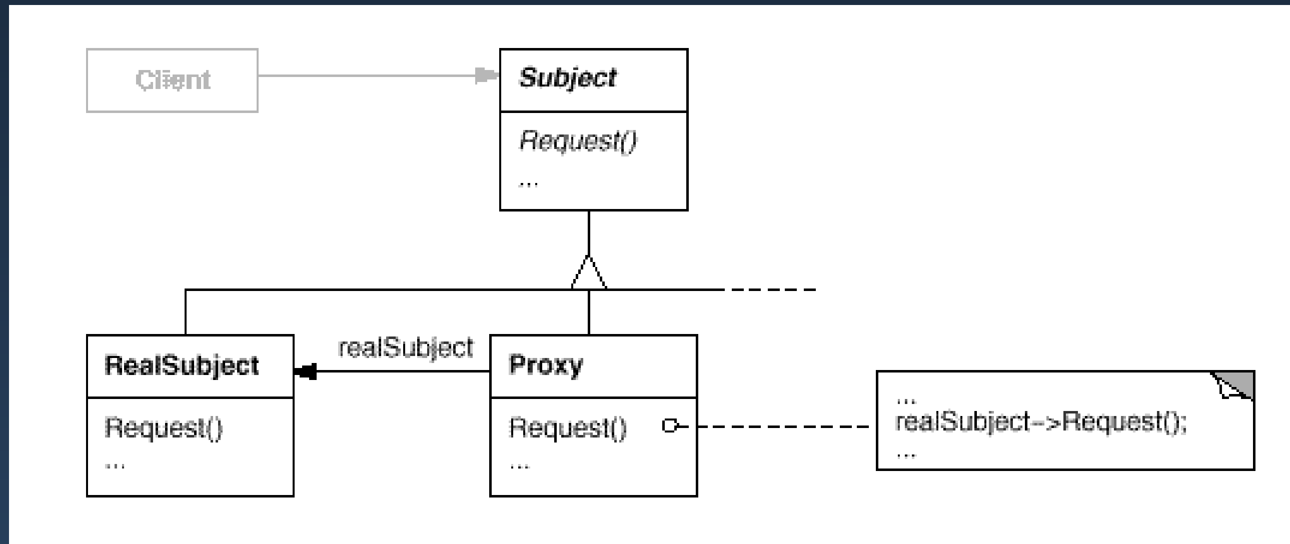
- Whenever there is a need for a more versatile or sophisticated reference to an object than a simple pointer
  - A remote proxy provides a local representative for an object in a different address space
  - A virtual proxy creates expensive objects on demand
  - A protection proxy controls access to the original object
    - Protection proxies are useful when objects should have different access rights
- A smart reference is a replacement for a bare pointer that performs additional actions when an object is accessed
  - counting the number of references to the real object (smart pointer)
  - loading a persistent object into memory when it is first referenced
  - checking that the real object is locked before it is accessed to ensure that no other object can change it

Structure



- Subject
  - defines the common interface for RealSubject and Proxy so that a Proxy can be used anywhere a RealSubject is expected
- RealSubject
  - defines the real object that the proxy represents

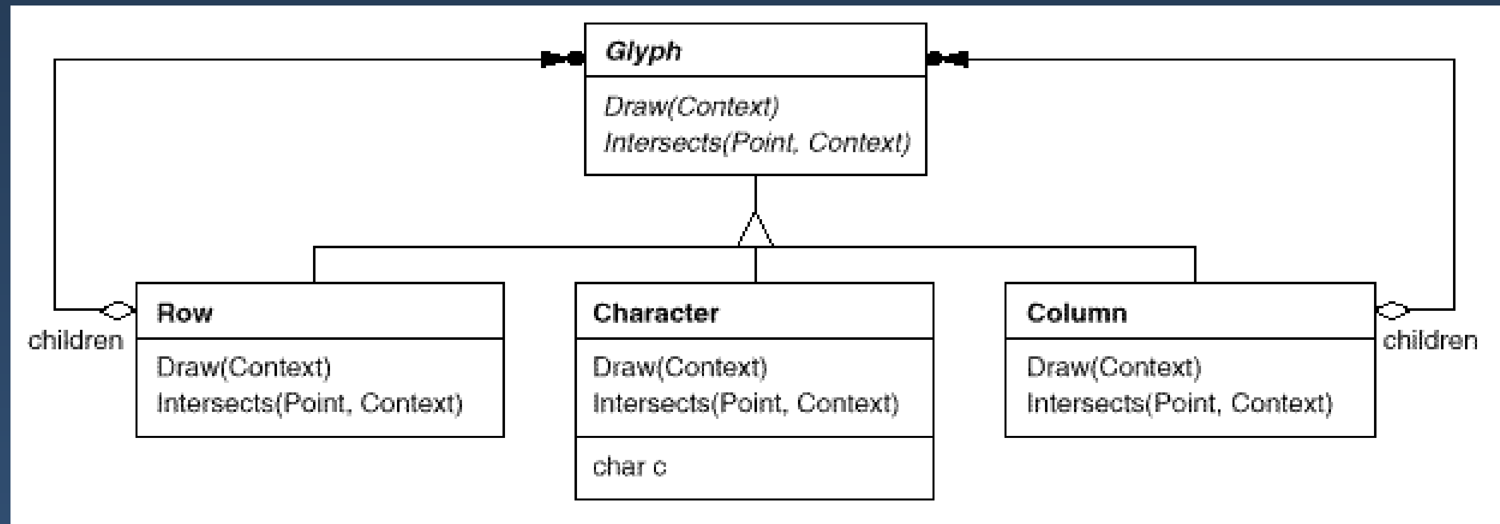
Structure



- Proxy
  - maintains a reference that lets the proxy access the real subject
  - provides an interface identical to Subject's so that a proxy can by substituted for the real subject
  - controls access to the real subject and may be responsible for creating and deleting it
    - *remote proxies* are responsible for encoding a request and its arguments and for sending the encoded request to the real subject in a different address space
    - *virtual proxies* may cache additional information about the real subject so that they can postpone accessing it
    - *protection proxies* check that the caller has the access permissions required to perform a request

Flyweight

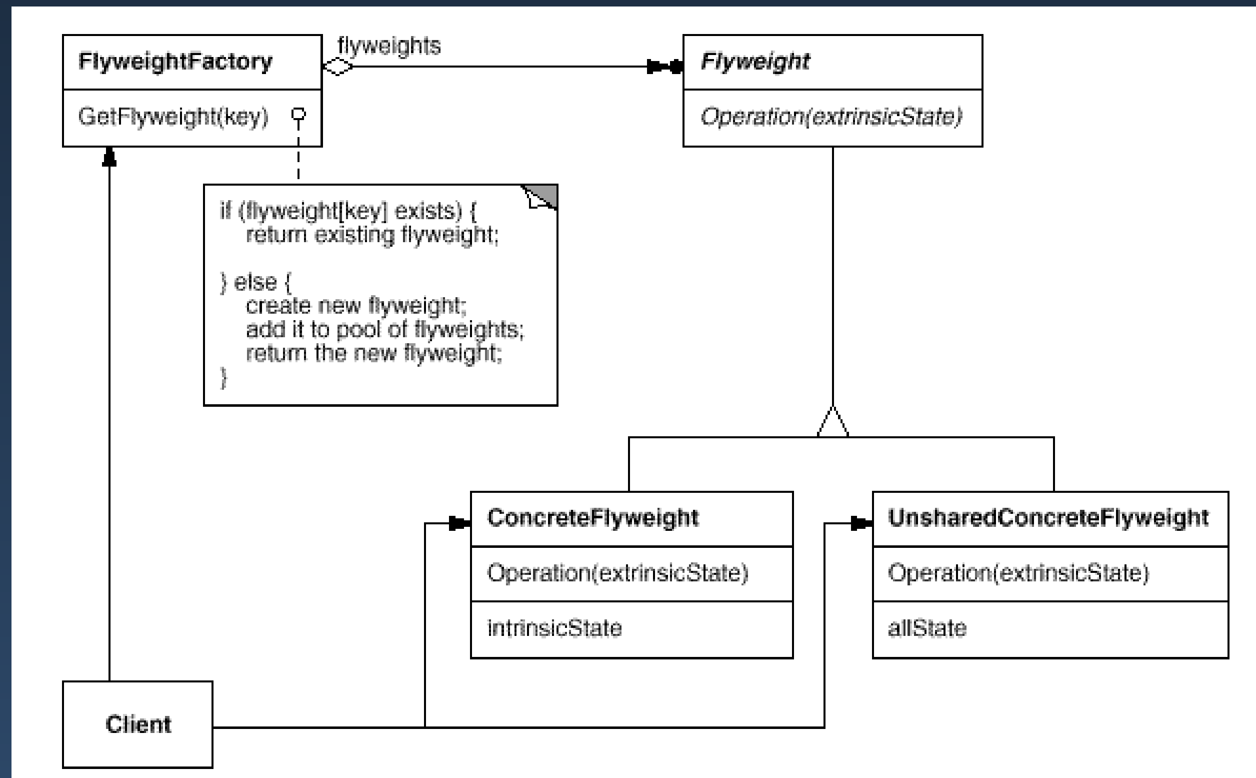
- Use sharing to support large numbers of fine-grained objects efficiently



Applicability

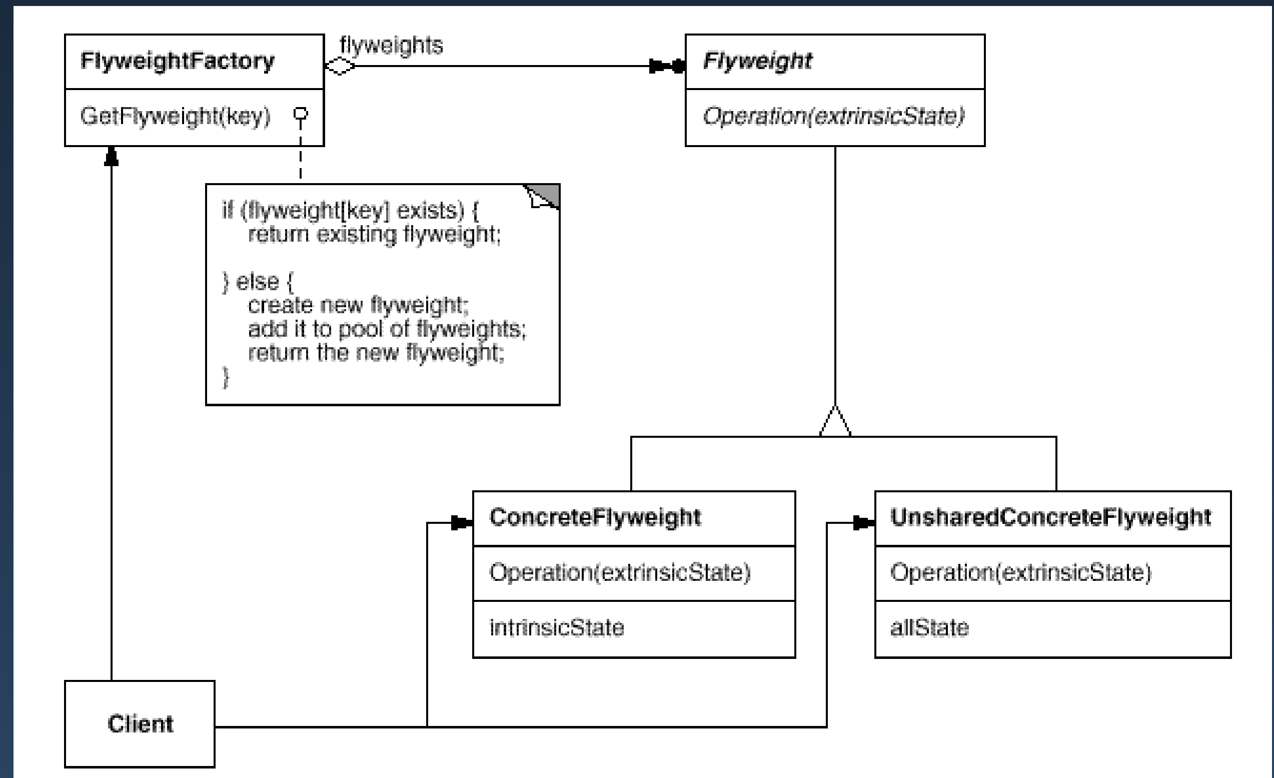
- Use when:
  - An application uses a large number of objects
  - Storage costs are high because of the sheer quantity of objects
  - Most object state can be made extrinsic
  - Many groups of objects may be replaced by relatively few shared objects once extrinsic state is removed
  - The application does not depend on object identity
    - Since flyweight objects may be shared, identity tests will return true for conceptually distinct objects

Structure



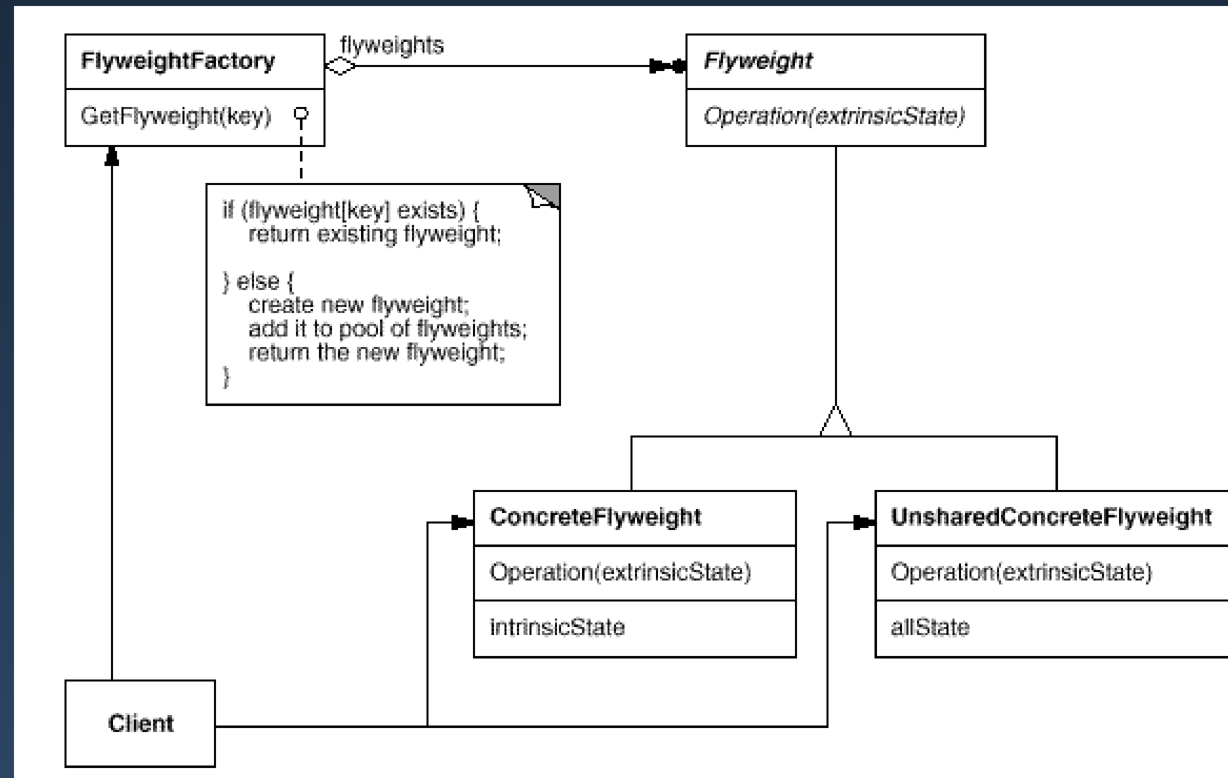
- Flyweight
  - declares an interface through which flyweights can receive and act on extrinsic state

Structure



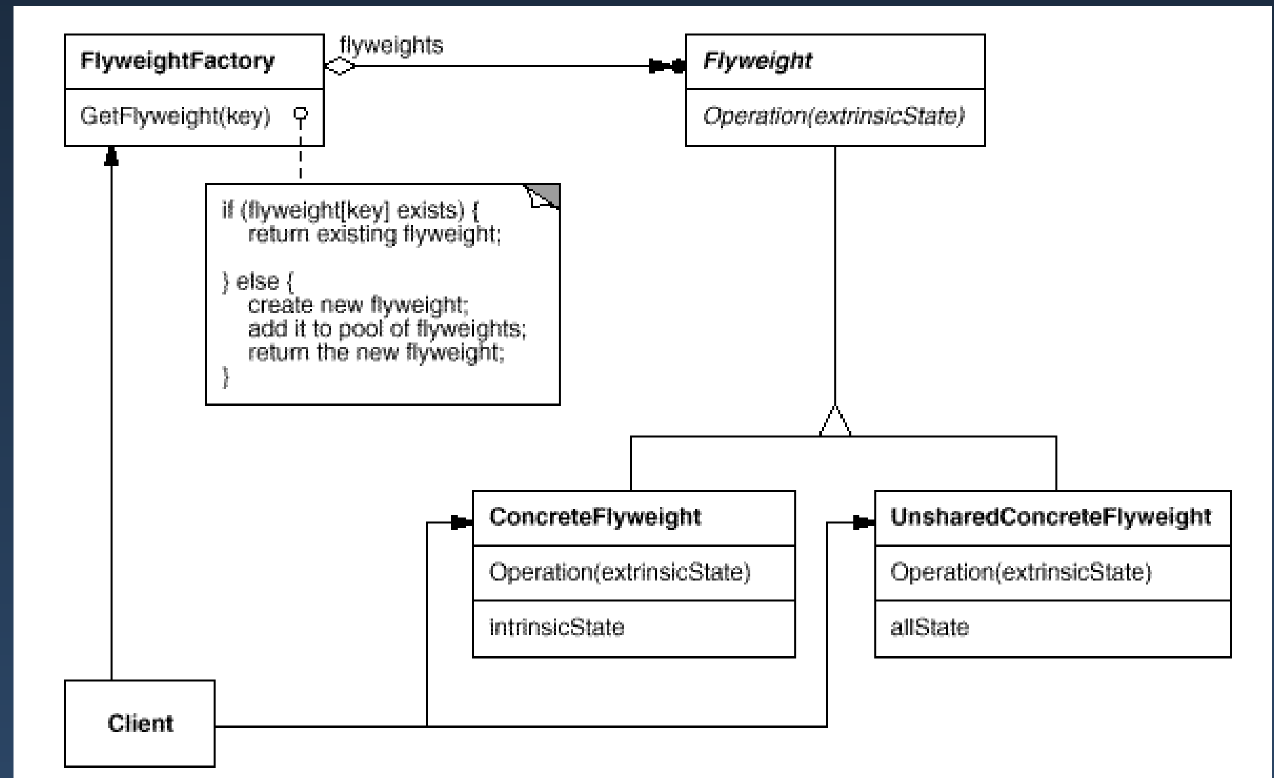
- ConcreteFlyweight
  - implements the Flyweight interface and adds storage for intrinsic state, if any
  - must be sharable
    - any state it stores must be intrinsic (independent of context)

Structure



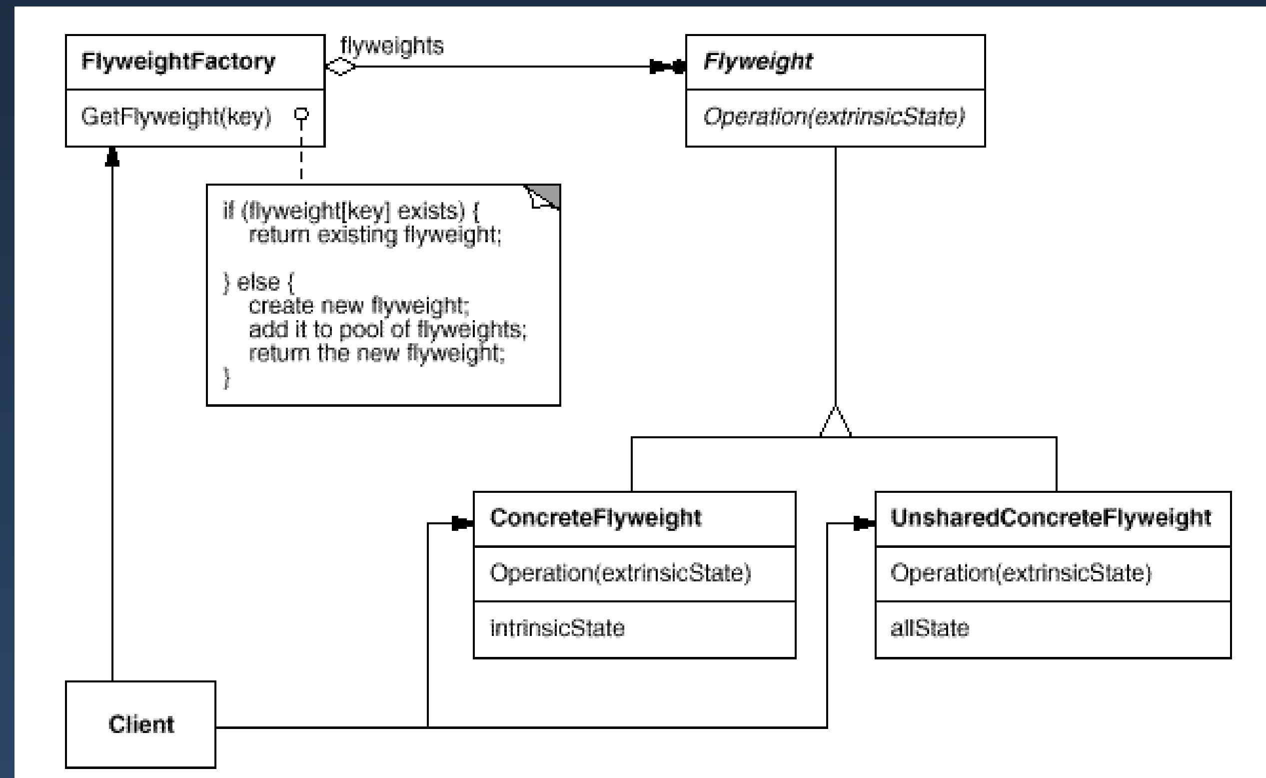
- UnsharedConcreteFlyweight
  - not all Flyweight subclasses need to be shared
  - The Flyweight interface *enables* sharing; it doesn't enforce it

Structure



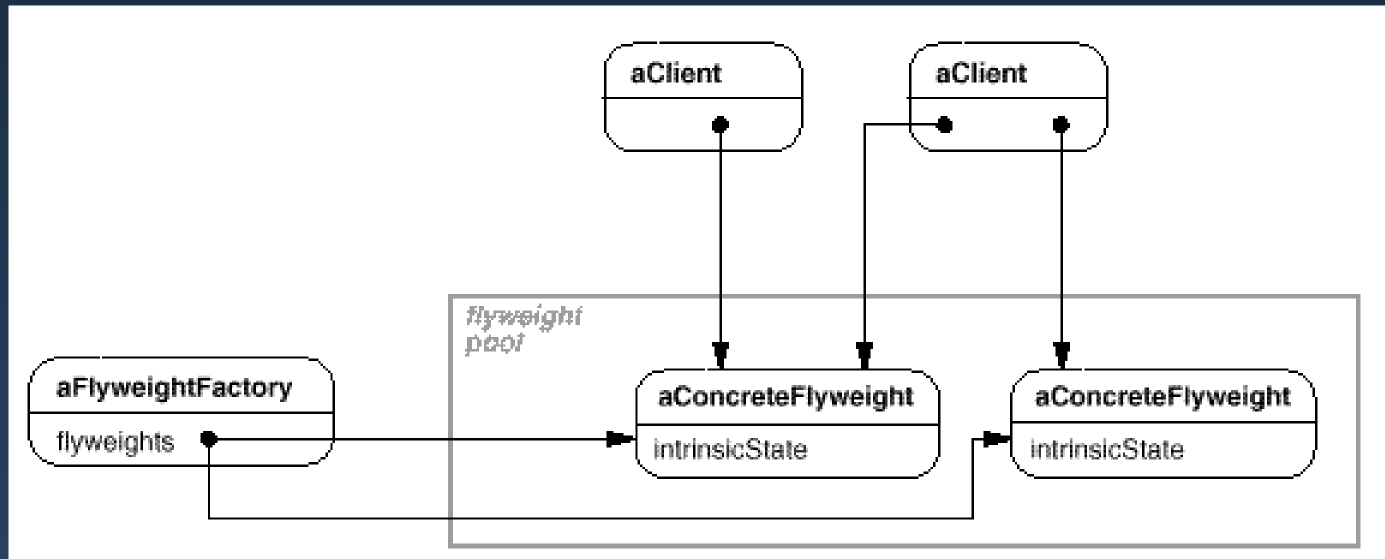
- FlyweightFactory
  - creates and manages flyweight objects
  - ensures that flyweights are shared properly
    - when a client requests a flyweight, the FlyweightFactory object supplies an existing instance or creates one, if none exists

Structure



- Client
  - maintains a reference to flyweights
  - computes or stores the extrinsic state of flyweights

Structure



- Clients should not instantiate ConcreteFlyweights directly
- Clients must obtain ConcreteFlyweight objects exclusively from the FlyweightFactory object to ensure they are shared properly

Consequences

- Flyweights introduce run-time costs associated with transferring, finding, and/or computing extrinsic state
- Costs are offset by space savings
  - (which also save run-time costs)
  - depends on
    - the reduction in the total number of instances that comes from sharing
    - the amount of intrinsic state per object
    - whether extrinsic state is computed or stored
- Often coupled with Composite to represent a hierarchical structure as a graph with shared leaf nodes
  - flyweight leaf nodes cannot store a pointer to their parent
  - parent pointer is passed to the flyweight as part of its extrinsic state
    - profound effect on object collaboration

Implementation

- Extrinsic State *e.g.,* Document editor
  - character font, *type style,* and colour.
  - store a map that keeps track of runs of characters with the same typographic attributes
- Shared Objects
  - FlyweightFactory can use an associative array to find existing instances
  - need reference counting for garbage collection

Behavioural Patterns

- Chain of Responsibility (requests through a chain of candidates)
- Command (encapsulates a request)
- Interpreter (grammar as a class hierarchy)
- Iterator (abstracts traversal and access)
- Mediator (indirection for loose coupling)
- Memento (externalize and re-instantiate object state)
- Observer (defines and maintains dependencies)
- State (change behaviour according to changed state)
- Strategy (encapsulates an algorithm in an object)
- Template Method (step-by-step algorithm w/ inheritance)
- Visitor (encapsulated distributed behaviour)

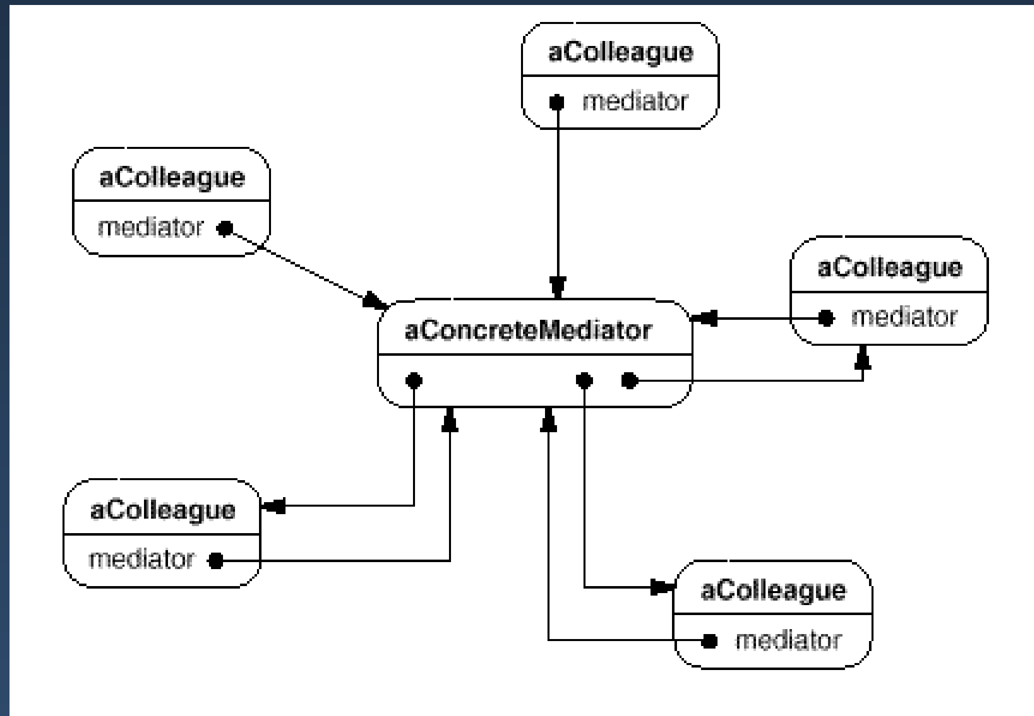
Mediator

- Defines an object that encapsulates how a set of objects interact
  - promotes loose coupling by keeping objects from referring to each other explicitly
  - lets you vary their interaction independently
- Create a mediator to control and coordinate the interactions of a group of objects

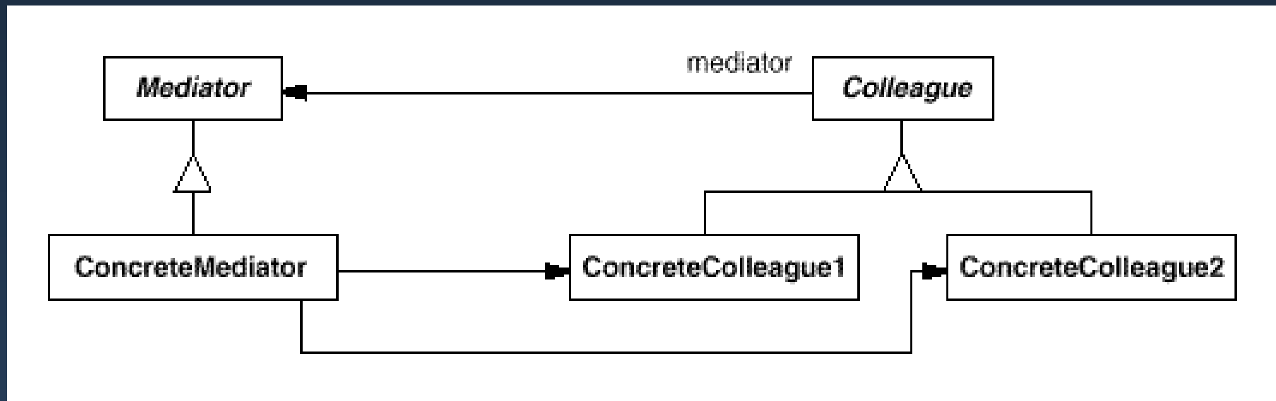
Applicability

- A set of objects communicate in a well-defined but complex manner
- Reusing an object is difficult because it refers to and communicates with many other objects
- A behavior that's distributed between several classes should be customizable without a lot of subclassing

Structure



Structure



- Mediator
  - defines an interface for communicating with Colleague objects
- ConcreteMediator
  - knows and maintains its colleagues
  - implements cooperative behavior by coordinating Colleagues
- Colleague classes
  - each Colleague class knows its Mediator object
  - each colleague communicates with its mediator whenever it would have otherwise communicated with another colleague

Consequences

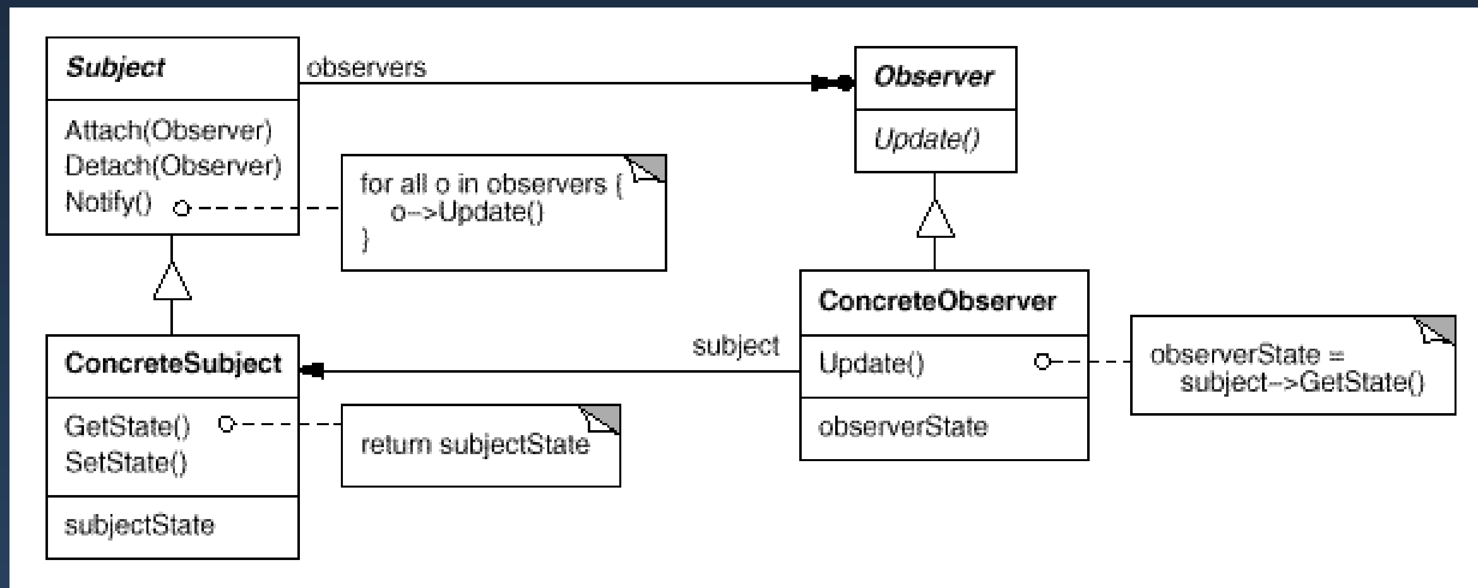
- Limits subclassing
  - localizes behaviour that otherwise would need to be modified by subclassing the colleagues
- Decouples colleagues
  - can vary and reuse colleague and mediator classes independently
- Simplifies object protocols
  - replaces many-to-many interactions with one-to-many
  - one-to-many are easier to deal with
- Abstracts how objects cooperate
  - can focus on object interaction apart from an object's individual behaviour
- Centralizes control
  - *mediator can become a monster…*

Observer

- Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically
  - A common side-effect of partitioning a system into a collection of cooperating classes is

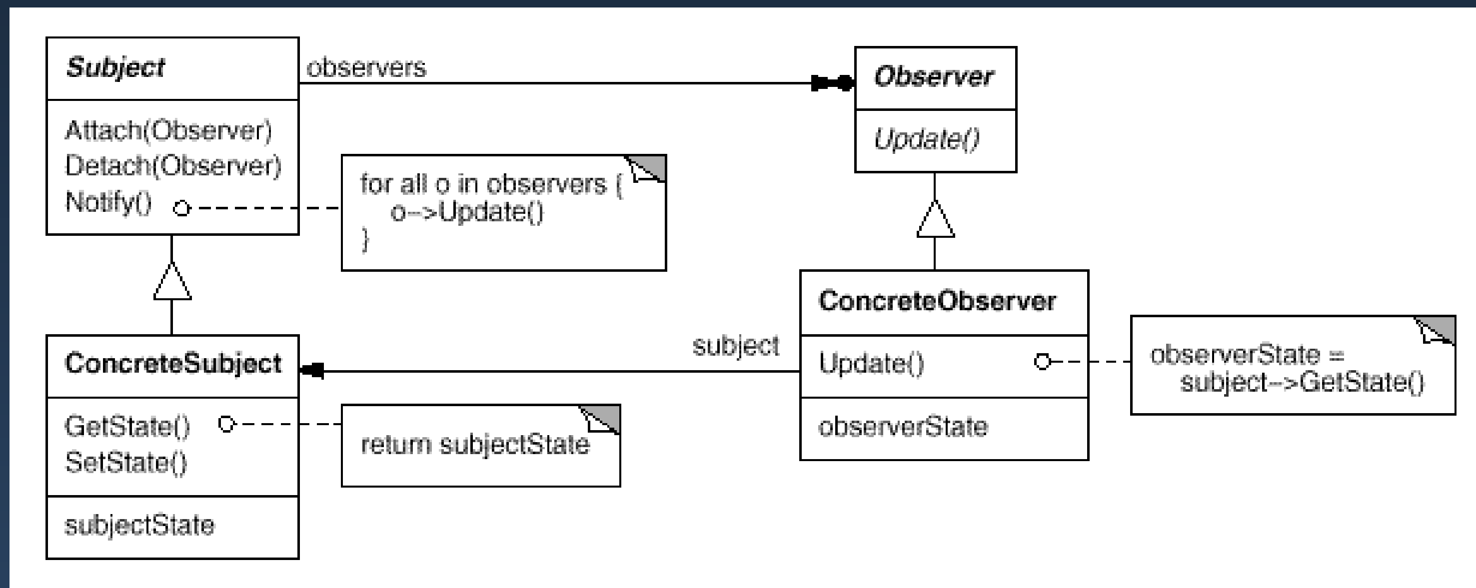
    the need to maintain consistency between related objects
  - You don't want to achieve consistency by making the classes tightly coupled, because that reduces their reusability
- a.k.a. Publish-Subscribe

Structure



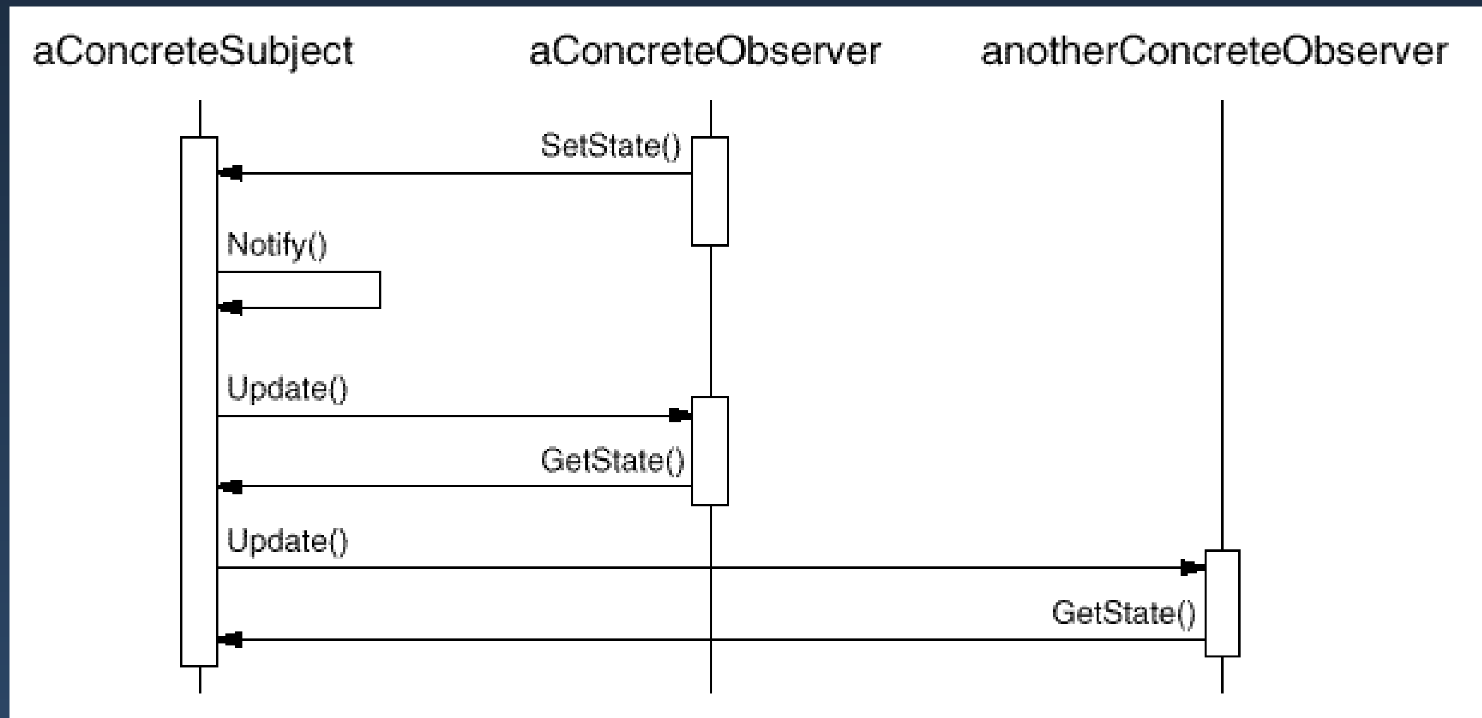
- Subject
  - knows its observers
  - any number of Observers may observe one subject
- Observer
  - defines an updating interface for objects that should be notified of changes to the subject

Structure



- Concrete Subject
  - stores the state of interest to ConcreteObservers
  - send notification when its state changes
- Concrete Observer
  - maintains a reference to the ConcreteSubject objects
  - stores state that should remain consistent with subject's
  - implements the Observer updating interface

Collaborations



- Subject notifies its observers whenever a change occurs that would make its observers' state inconsistent with its own
- After being informed, observer may query subject for changed info
  - uses query to adjust its state

Applicability

- When an abstraction has two aspects, one dependent upon the other
  - Encapsulating these aspects into separate objects lets you vary them independently

- When a change to one object requires changing others, and you don't know ahead of time how many there are or their types
  - when an object should be able to notify others without making assumptions about who these objects are,
  - you don't want these objects tightly coupled

Consequences

- Abstract coupling
  - no knowledge of the other class needed
- Supports broadcast communications
  - subject doesn’t care how many observers there are
- Spurious updates a problem
  - can be costly
  - unexpected interactions can be hard to track down

Implementation

- Mapping subjects to observers
  - table-based or subject-oriented
- Observing more than one subject
  - interface must tell you which subject
  - data structure implications (e.g., linked list)
- Who triggers the notify()
  - subject state changing methods
    - > 1 update for a complex change
  - clients
    - complicates API & error-prone
    - can group operations and send only one update
  - transaction-oriented API to client

Implementation

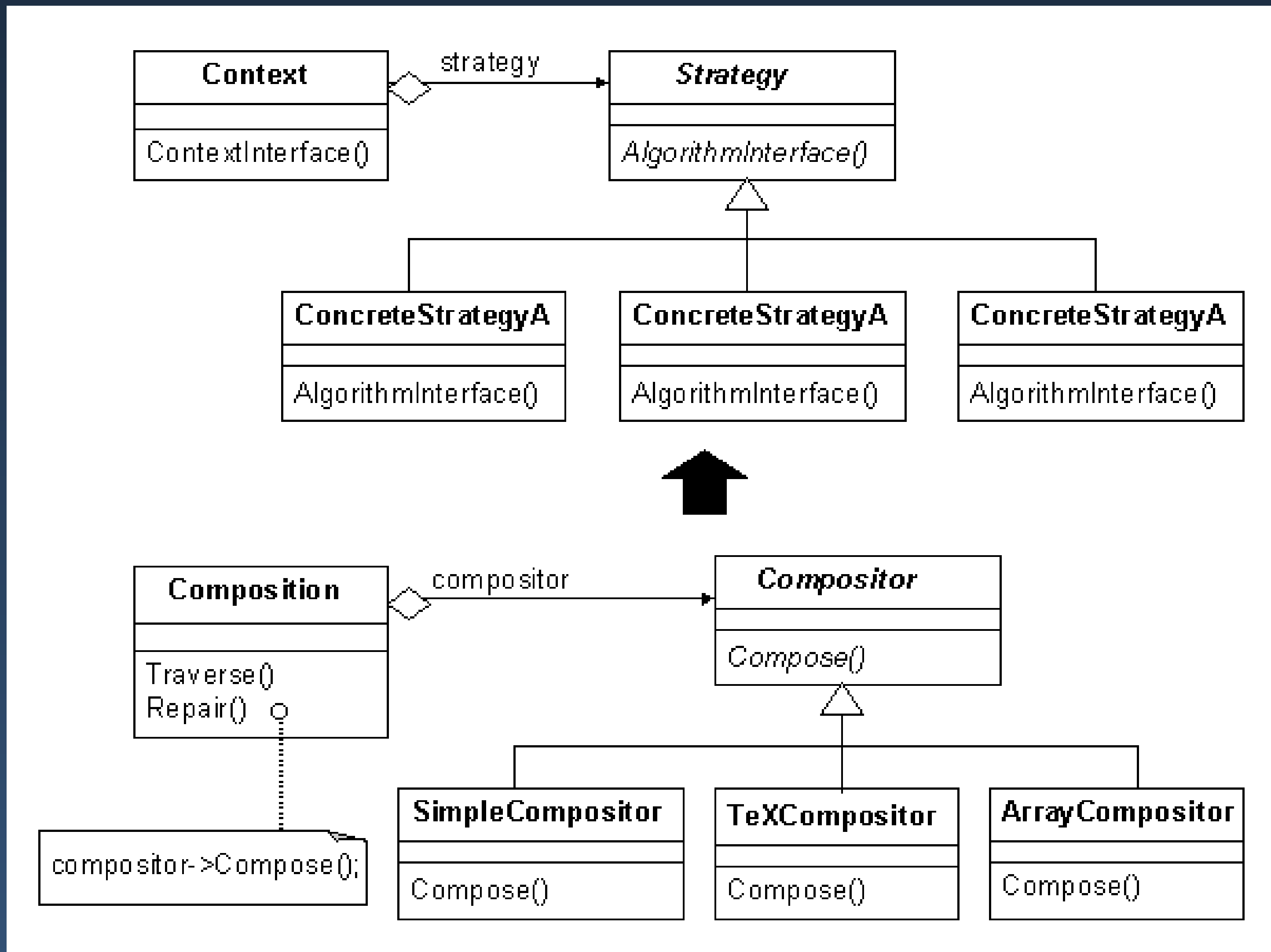
- Dangling references to deleted subjects
  - send 'delete message'
  - complex code
- Must ensure subject state is self-consistent before sending update
- Push versus pull
  - push: subject sends info it thinks observer wants
  - pull: observer requests info when it needs it
  - registration: register for what you want
    - when observer signs up, states what interested in
- ChangeManager
  - if observing more than one subject to avoid spurious updates
- Can combine subject and observer

Strategy

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

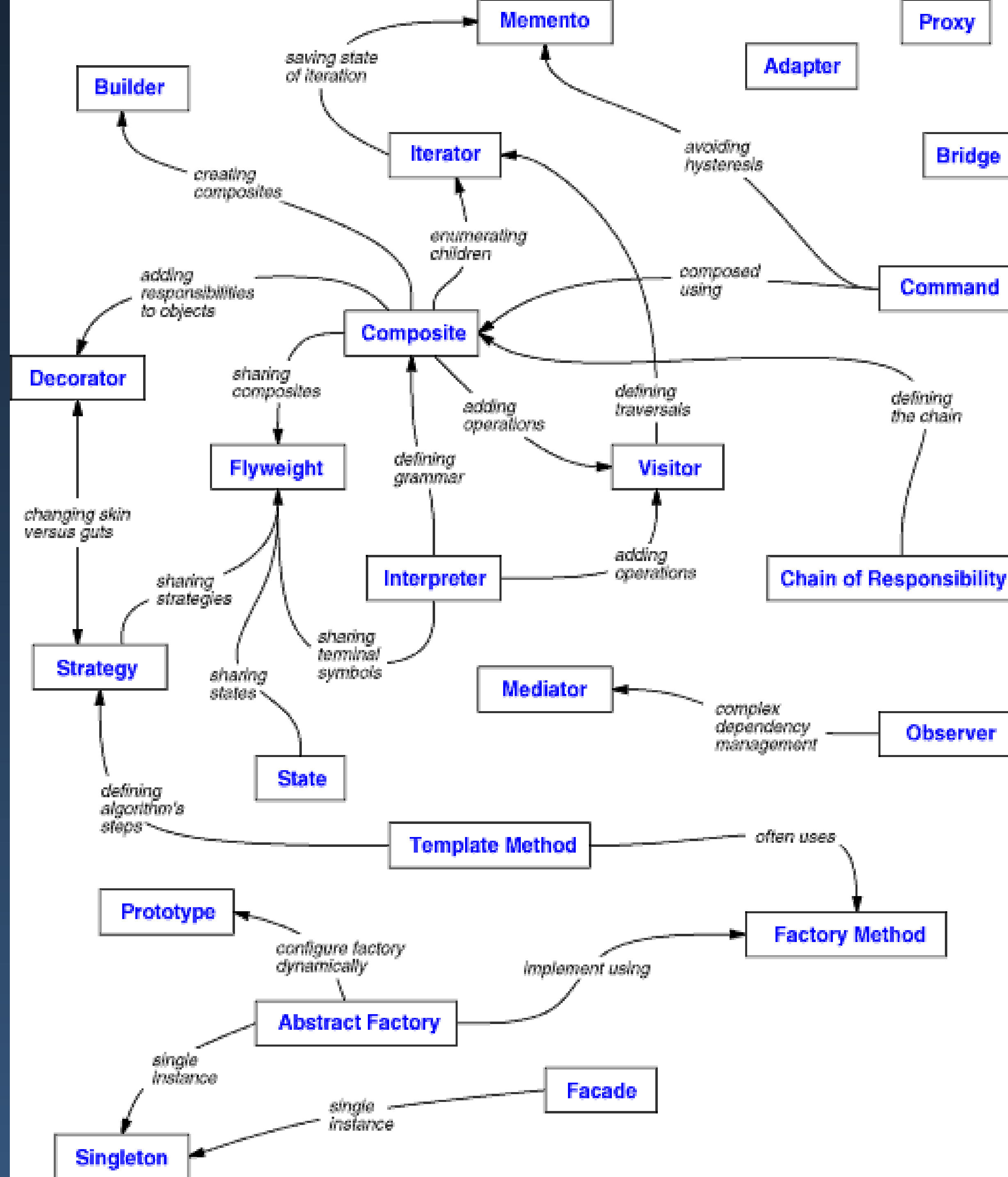
- Many related classes differ only in their behaviour
- You need different variants of an algorithm
  - Strategies can be used as a class hierarchy of algorithms
- An algorithm use data structures that clients shouldn't know about
- A class defines many behaviors, and these appear as multiple conditionals in its operations

Strategy



Pattern relations

Patterns collaborate to model more complex designs



How to select a design pattern

- Consider how design patterns solve design problems
- Scan Intent sections of pattern documentation
- Study how patterns interrelate
  - See pattern relations slide
- Study patterns of like purpose
  - Understand similarities and differences of patterns grouped together in pattern categories slide
- Consider what should be variable in your design
  - Encapsulate the concept that varies instead of redesign
  - Understand what design aspects that patterns let you vary

How to use a design pattern

- Read the pattern once through for an overview
- Go back and study the structure, participants and collaborations sections
- Look at the sample code to see a concrete example of the pattern in code
  - Learn how to implement the pattern

- Choose names for pattern participants that are meaningful in the application context
- Define the classes
  - Including identifying existing classes in application that the pattern will affect
- Define application-specific names for operations

- Implement the operations to carry out responsibilities and collaborations in the pattern

Conclusion

- Design patterns
  - Provide solutions to common problems
  - Lead to extensible models and code
  - Can be used as is or as examples of interface inheritance and delegation

Designing object-oriented software is hard
Designing reusable object-oriented software is even harder

- Design patterns DO NOT solve all software engineering problems
  - patterns are a starting point, not a destination
  - models are not right or wrong, they are more or less useful

Homework

- Next lecture: describe a design pattern (not illustrated in the lectures)
- Geant4 Low Energy Electromagnetic Physics
- Look at the design diagrams on the web
  - http://www.ge.infn.it/geant4/lowE/
- Identify
  - a singleton *(creational pattern)*
  - a composite *(structural pattern)*
  - a strategy *(behavioural pattern)*
- Further homework in medical modeling exercises…