Behavioral 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)

Observer

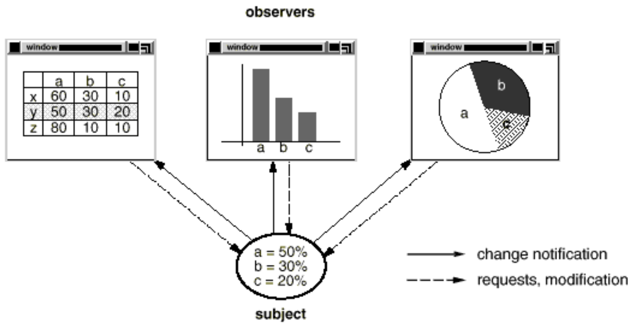
- Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
  – A common side-effect of partitioning a system into a collection of cooperating classes is
    - the need to maintain consistency between related objects

  – You don't want to achieve consistency by making the classes tightly coupled, because that reduces their reusability.

  – a.k.a. Publish-Subscribe

  – Common related/special case use: MVC
    - Model-View-Controller pattern

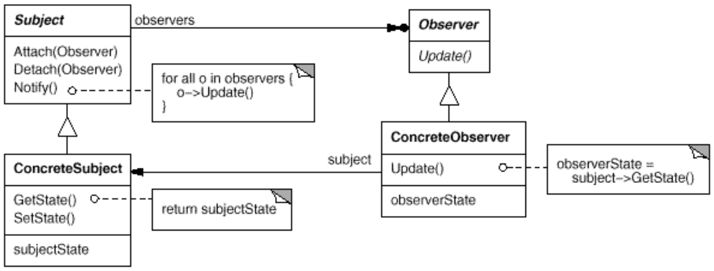
Motivation



	a	b	c
x	60	30	10
y	50	30	20
z	80	10	10

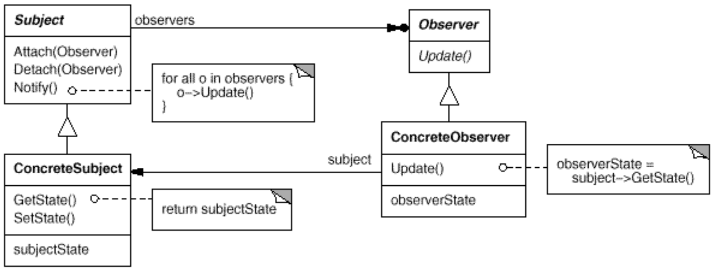
- Separate presentation aspects of the UI from the underlying application data.
  - e.g., spreadsheet view and bar chart view don't know about each other
    - they *act* as if they do: changing one changes the other.

Structure



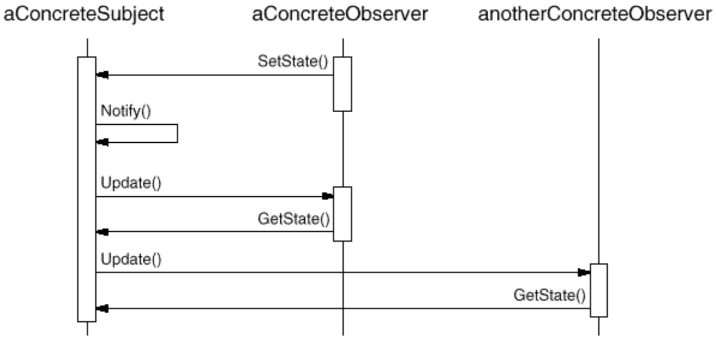
- Subject
  - knows its observers
  - any number of Observers may observe one subject
- Observer
  - defines an updating interface for objects that should be notified of changes to the subject

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
  - e.g., view and model

  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
  - problem aggravated when simple protocol that does not say what was changed is used

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

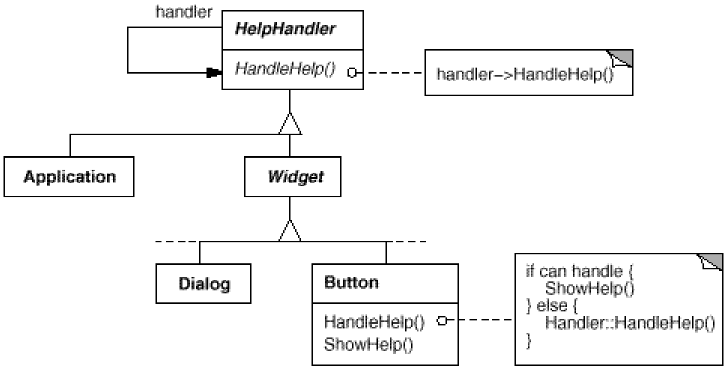
Implementation

- 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

Chain Of Responsibility

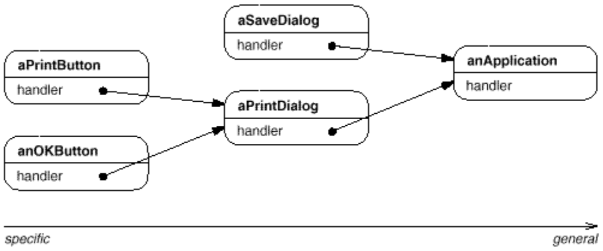
- Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request.
  - Chain the receiving objects and pass the request along the chain until an object handles it.

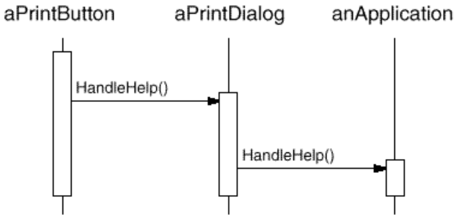
Motivation



- Context-sensitive help
  - User can obtain information on any part of a UI by clicking on it.
  - If no help available (e.g., for a button), system should display a more general help message about the context (e..g, the dialog box containing the button).

Motivation



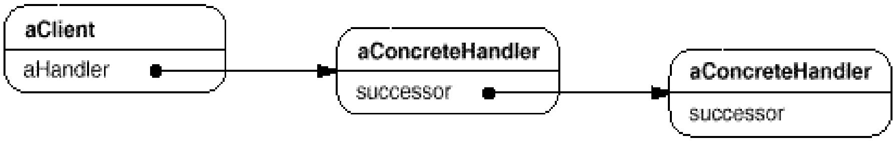


- Objects forward the request until there is one that can handle it.
- The key is that the client does not know the object that will eventually handle the request.

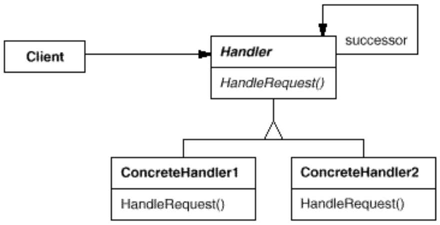
Applicability

- More than one object may handle a request, and the handler isn't known *a priori*.
  - The handler should be ascertained automatically.
- You want to issue a request to one of several objects without specifying the receiver explicitly.
- The set of objects that can handle a request should be specified dynamically.

Structure



Structure



- Handler
  - defines an interface for handling requests
  - implements the successor list (optional)
- ConcreteHandler
  - handles requests for which it is responsible
  - can access its successor
  - forward to successor if it can't handle the request
- Client
  - initiates the request to the first ConcreteHandler in the chain.

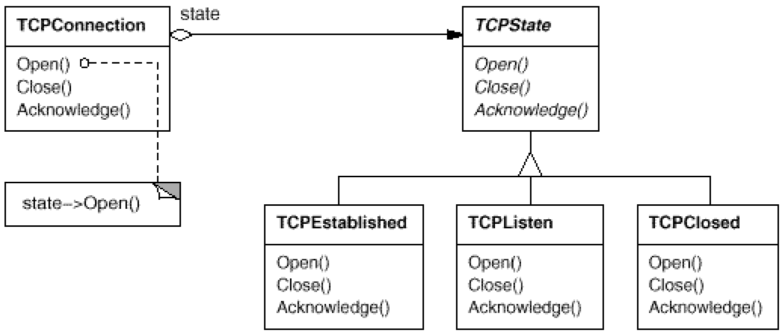
Consequences

- reduced coupling
  - receiver and sender have no explicit knowledge of each other
  - can simplify object interactions
- added flexibility
  - can add or change responsibilities by changing the chain at run-time.
- receipt is not guaranteed.
  - request may fall off the end of the chain

State

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

Motivation

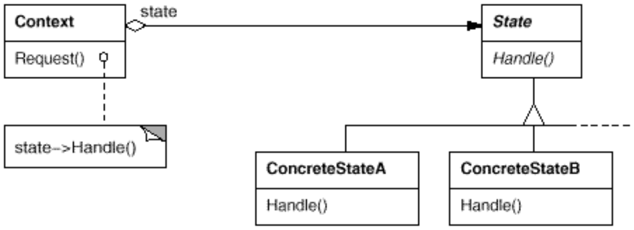


- A TCPConnection object that responds differently to requests given its current state.
- All state-dependent actions are delegated.

Applicability

- An object's behavior depends on its state, and it must change its behavior at run-time depending on that state.
- Operations have large, multipart conditional statements that depend on the object's state.
  - This state is usually represented by one or more enumerated constants.
  - Often, several operations will contain this same conditional structure.
  - The State pattern puts each branch of the conditional in a separate class.
  - This lets you treat the object's state as an object in its own right that can vary independently from other objects.

Structure



- **Context**
  - defines the interface of interest to clients.
  - maintains an instance of a ConcreteState subclass that defines the current state.
- **State**
  - defines an interface for encapsulating the behavior associated with a particular state of the Context.
- **ConcreteState subclasses**
  - each subclass implements a behavior associated with a state of the Context.

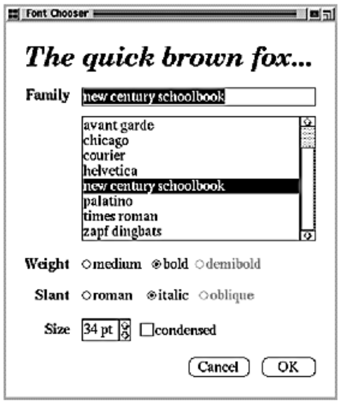
Consequences

- It localizes state-specific behavior and partitions behavior for different states.
  - The State pattern puts all behavior associated with a particular state into one object.
  - Because all state-specific code lives in a State subclass, new states and transitions can be added easily by defining new subclasses.
- It makes state transitions more explicit
  - State is represented by the object pointed to.
- It protects the object from state-related inconsistencies.
  - All implications of state changed wrapped in the atomic change of 1 pointer.
- State object can be shared
  - if no data members they can be re-used across all instances of the Context

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

Motivation



Font Chooser

The quick brown fox...

Family: new century schoolbook

avant garde
chicago
courier
helvetica
new century schoolbook
palatino
times roman
zapf dingbats

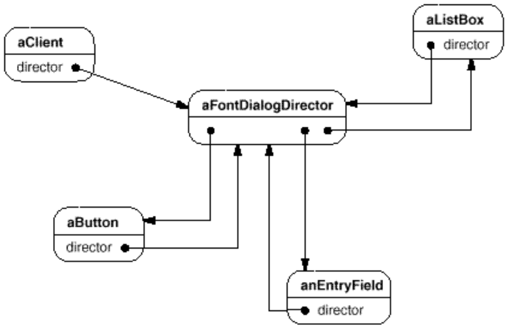
Weight ○medium ●bold ○demibold

Slant ○roman ●italic ○oblique

Size 34 pt □condensed

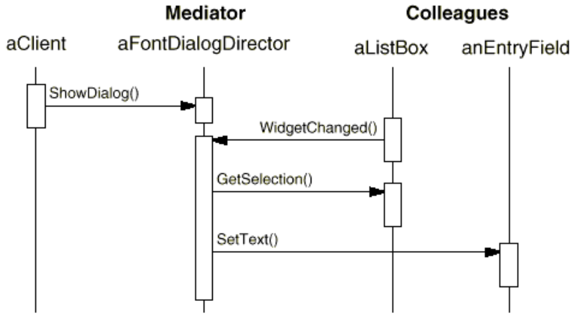
- A collection of widgets that interact with one another.
  - e.g., certain families may not have certain weights
    - disable ‘demibold’ choice

Motivation



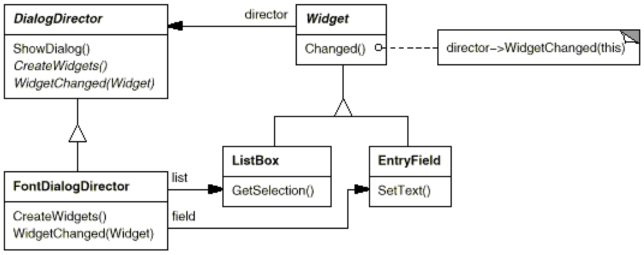
- Create a mediator to control and coordinate the interactions of a group of objects.

Motivation



- *e.g.,*
  - list box selection moving to entry field
  - entryField now calls WidgetChanged() and enables/disables
  - entry field does not need to know about list box and *vice-versa*

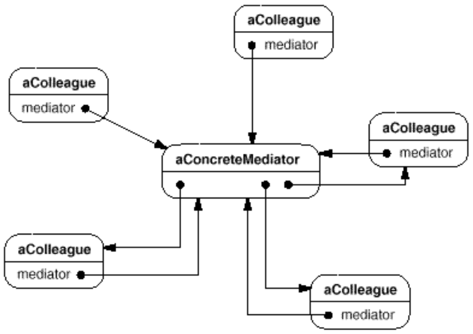
Motivation



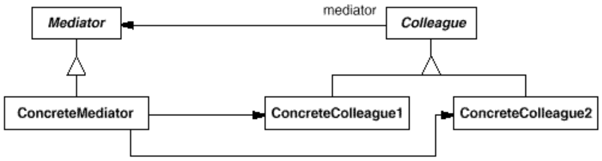
Applicability

- 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

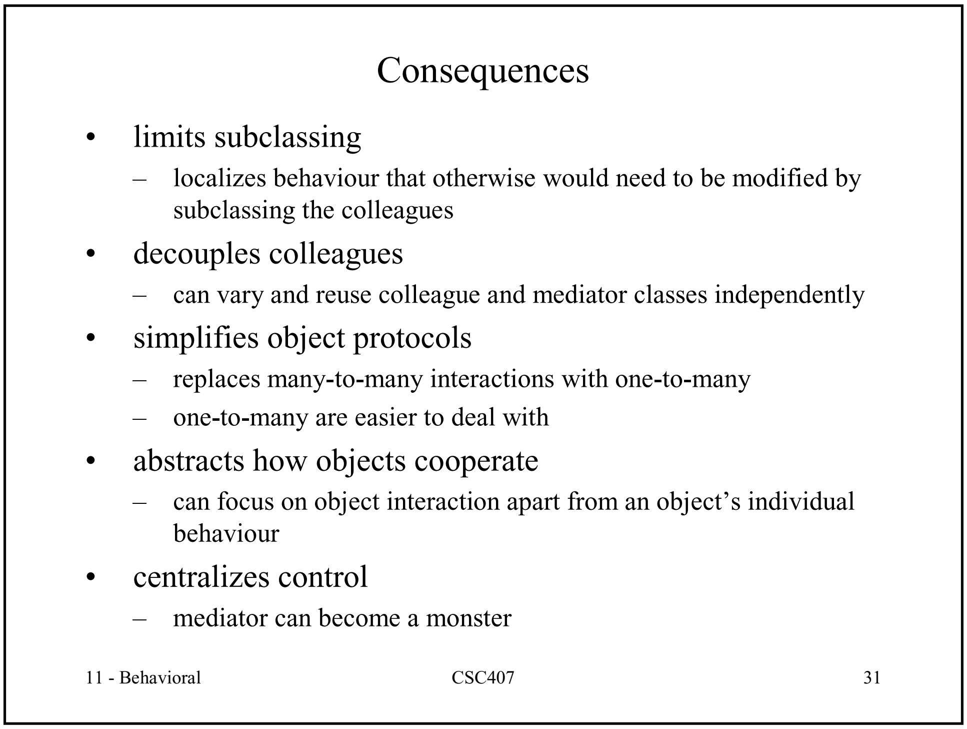
Structure



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