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)

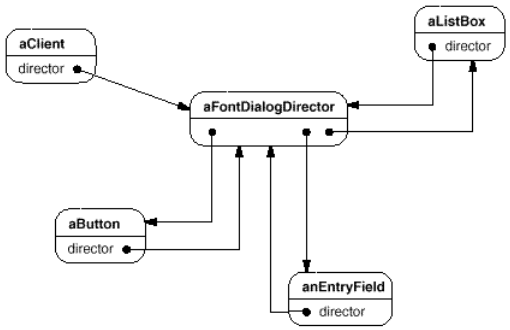
Mediator

- 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

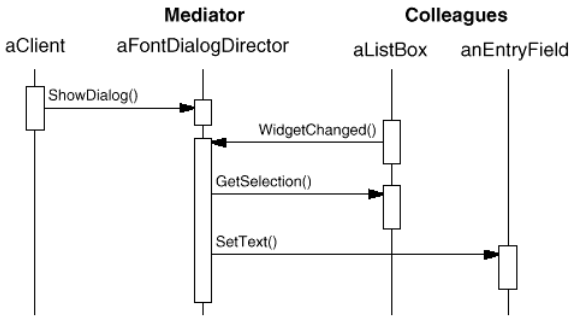
- A collection of widgets that interact with one another.
  - e.g., certain familes may not have certain weights
    - disable 'demibold' choice

Motivation



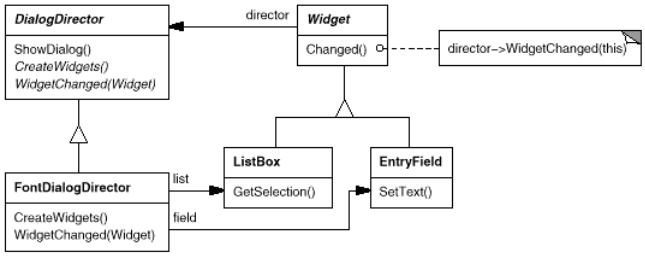
- Create a mediator to control and coordinate the interactions of a group of objects.

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*

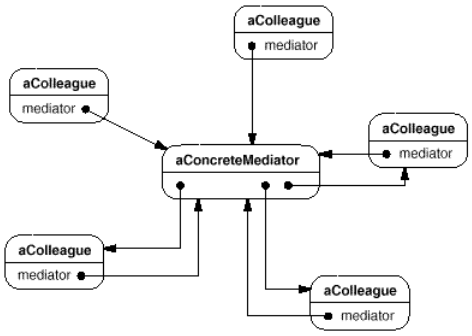
Motivation



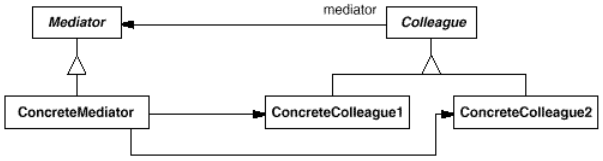
Applicability

- 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

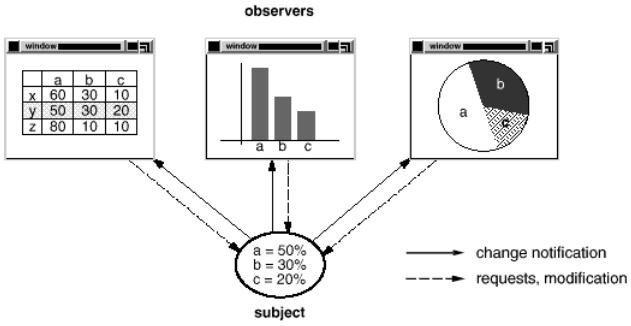
Consequences

- 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

Observer

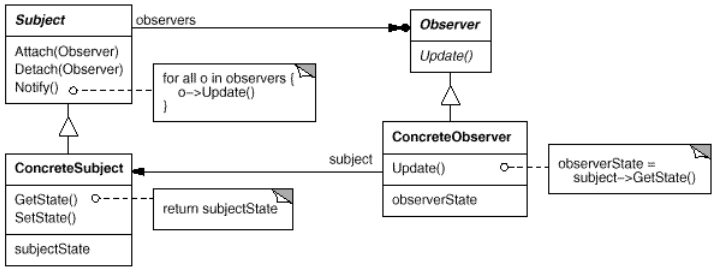
- Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
  - A common side-effect of partitioning a system into a collection of cooperating classes is
    - the need to maintain consistency between related objects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



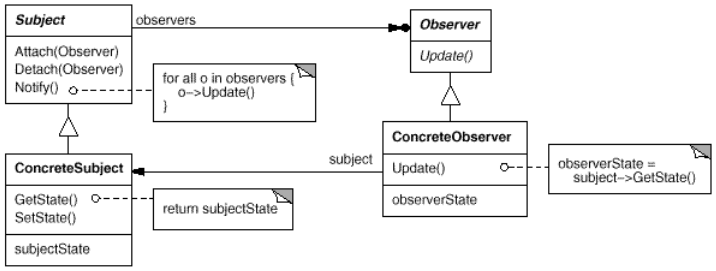
- Separate presentation aspects of the UI from the underlying application data.
  - e.g., spreadsheet view and bar chart view don't know about each other
    - they *act* as if they do: changing one changes the other.

Structure



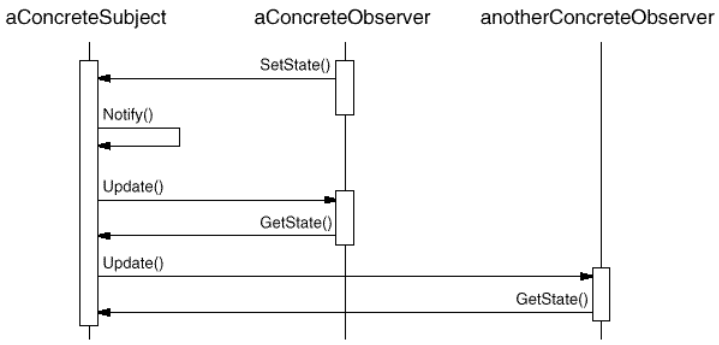
- Subject
  - knows its observers
  - any number of Observers may observe one subject
- Observer
  - defines an updating interface for objects that should be notified of changes to the subject

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