Software Engineering

Lesson Design Pattern 09
Observer, Mediator
v1.0

Uwe Gühl

Fall 2007/ 2008

Contents

- Observer
- Mediator

Used sources:

- [GHJ04] Gamma, Helm, Johnson, Vlissides: Design Pattern, Addison Wesley, 2004
- [Hus08] Vince Huston: Design Pattern, www.vincehuston.org/dp/, 2008

Observer

- Intent:
  - Defines a 1:n (one to many) relationship, so that if an object is changing, all dependent objects could be updated automatically
  - also known as
    - Dependents
    - Publish - Subscribe
  - ... is a Behavioral Pattern

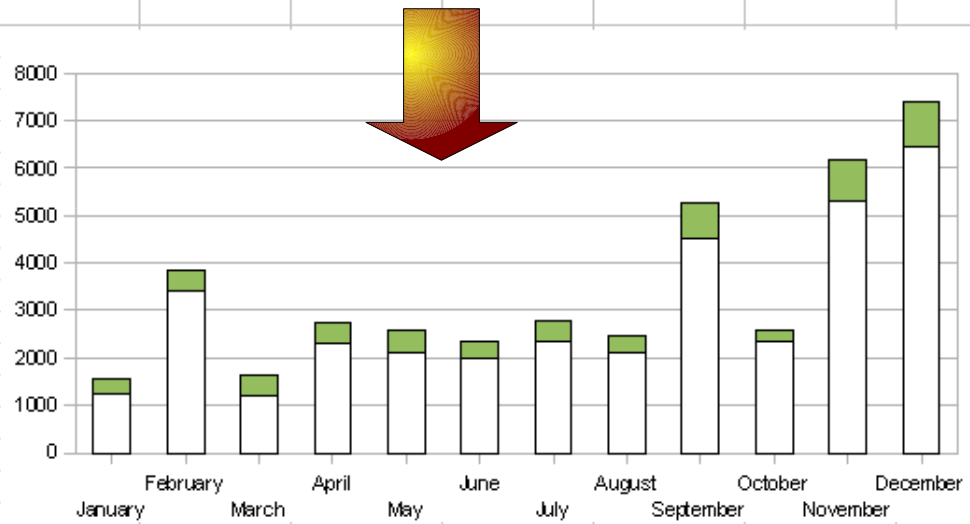
Observer

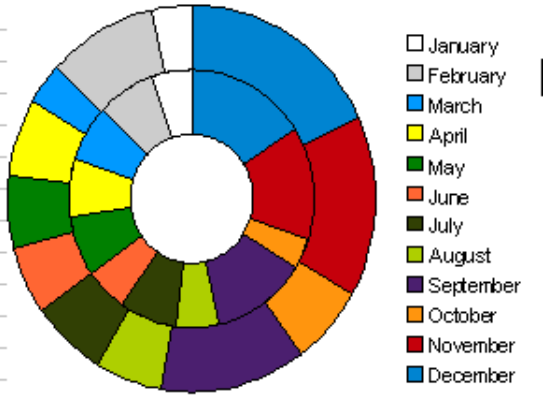
- Motivation
  - Breakdown of a system in cooperating classes:
    - Changes of an object affect in the system several different places
    - Objects should react on status changes of specific objects
  - Goal: Consistent status of depending objects without close coupling of the concerned objects → Reuse
  - GUI toolkits often differ between displaying and basic application data (layer model)

Observer

- Example
  - Different graphical presentations of numerical values

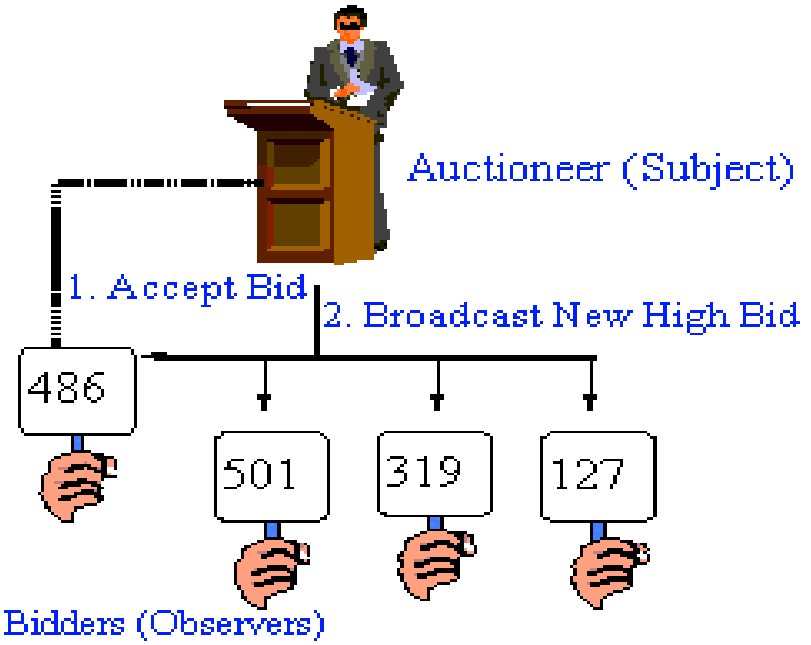
	January	February	March	April	May	June	July	August	September	October	November	December
Sales	1234	3412	1223	2322	2123	1987	2344	2123	4523	2342	5323	6433
Profit	324	453	435	435	454	343	452	345	745	235	865	956





Observer

- Example [Hus08]



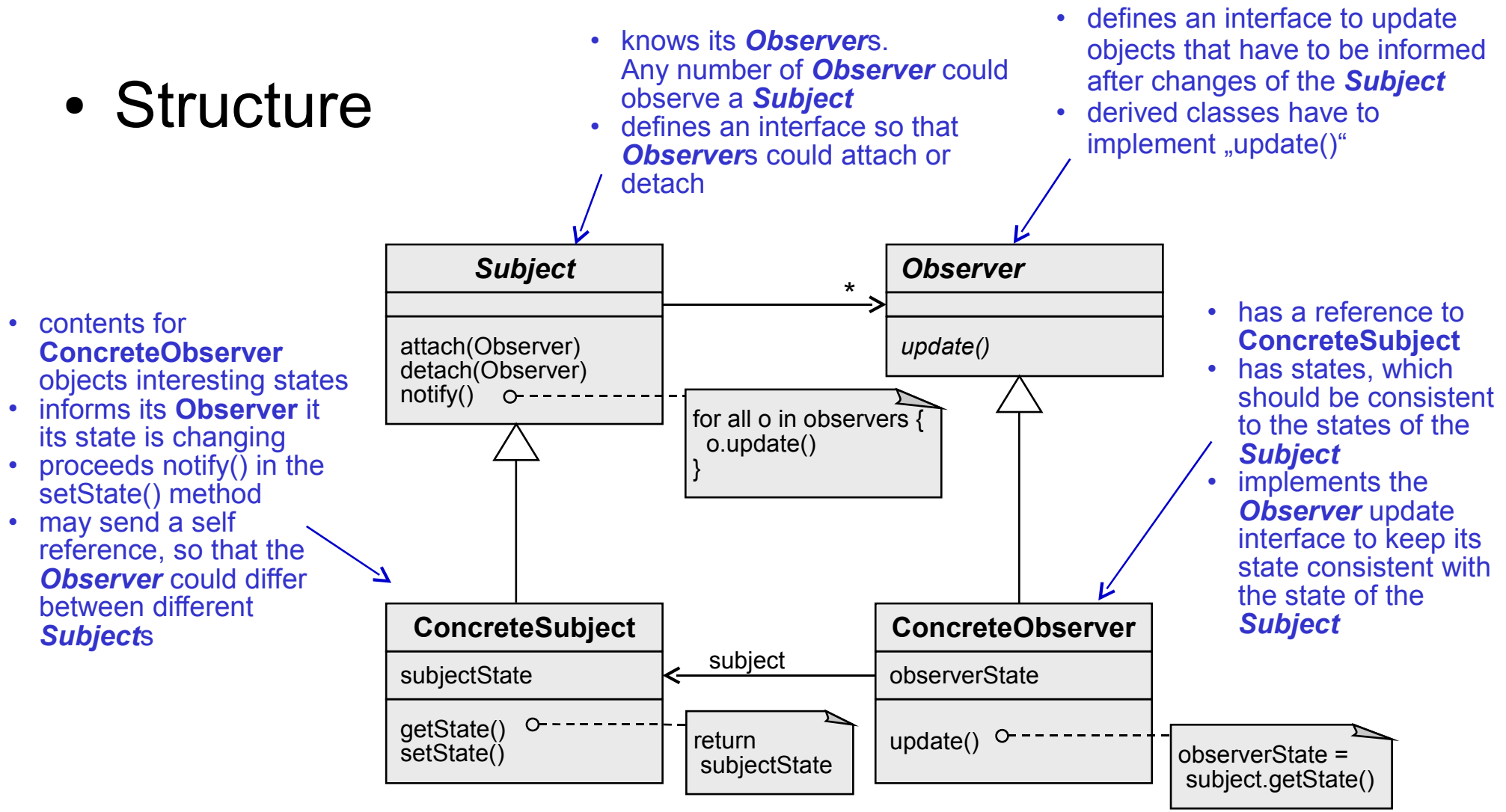
Some auctions demonstrate this pattern.
Each bidder possesses a numbered paddle that is used to indicate a bid.
The auctioneer starts the bidding, and "observes" when a paddle is raised to accept the bid.
The acceptance of the bid changes the bid price
which is broadcast to all of the bidders in the form of a new bid.
[Michael Duell, "Non-software examples of software design patterns",
Object Magazine, Jul 97, p54]

Observer

- Ideas
  - Implementation of a „publish subscribe“ interaction
  - All observer register at a sender object („subject“)
  - The subject holds a list of all registered observer
  - If the state is changing, the subject informs all registered observers, and they could react adequate

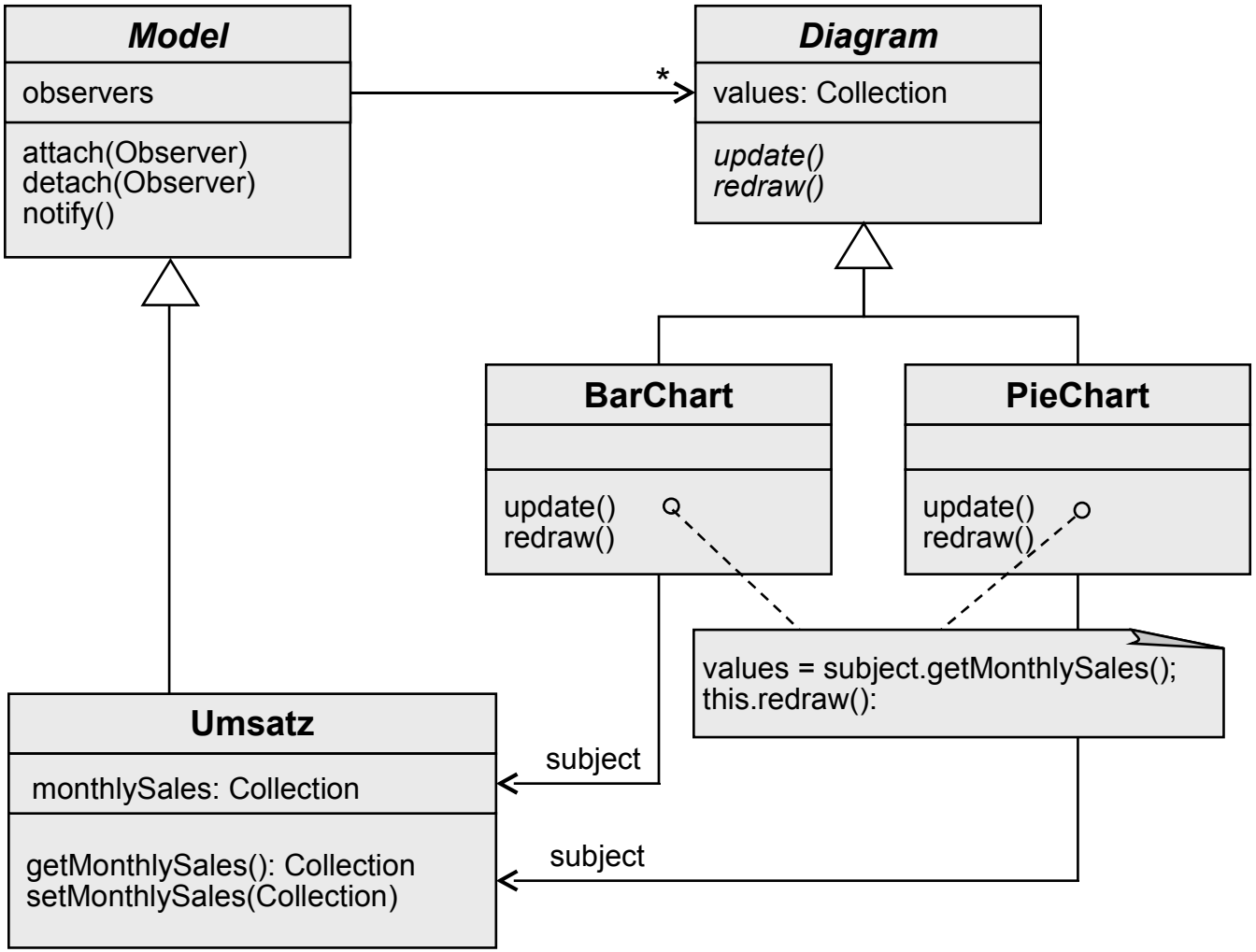
Observer

- Structure



Observer

- Example

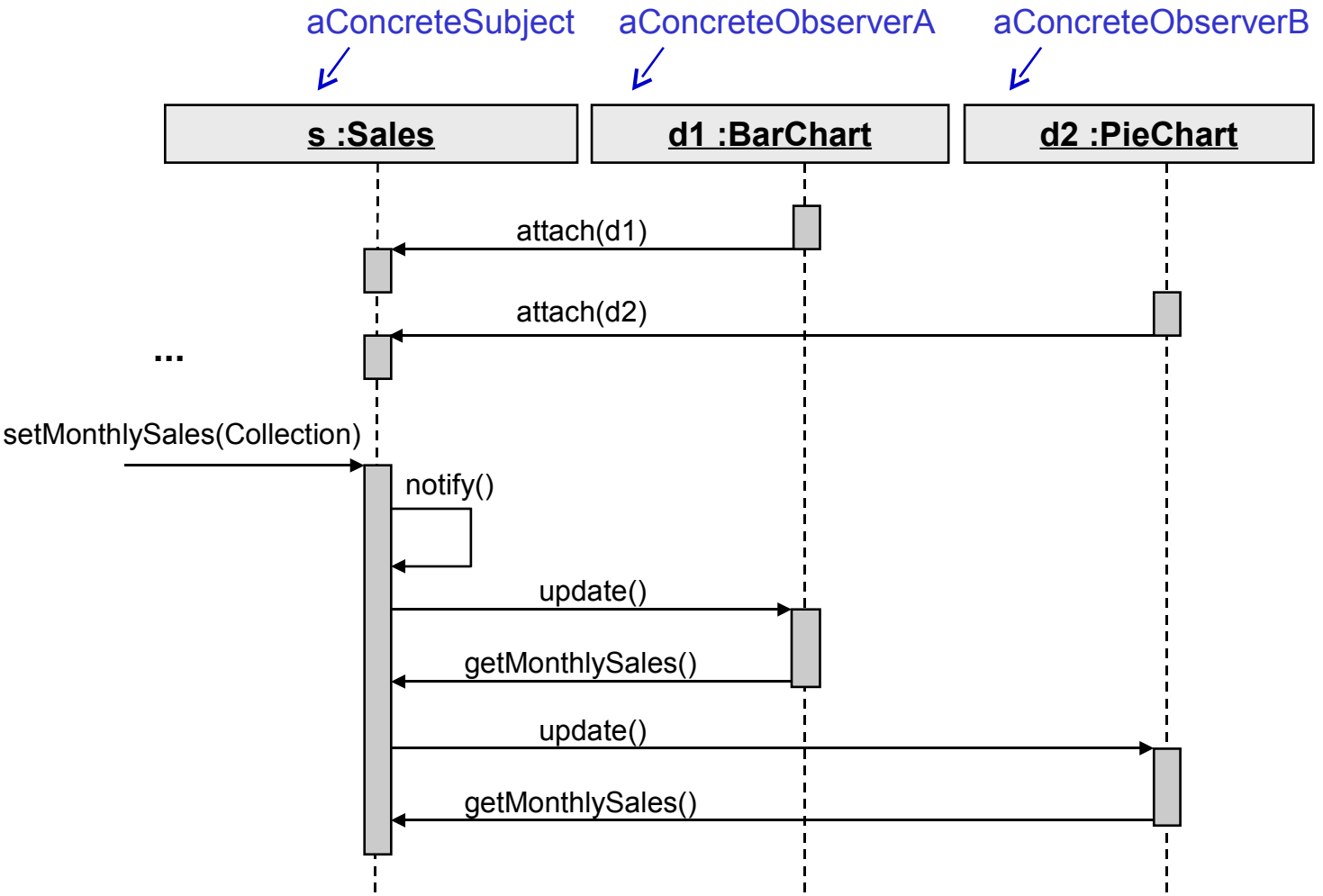


Observer

- Collaboration
  How do the objects work together**?**
  - aConcreteSubject registers all ***Observer***s to achieve consistence among them
  - aConcreteObserver requests after an advice note about a status change the corresponding information, e. g. to update the representation
  - If an Observer is responsible for information to be updated for a Subject
    - it first sends this information to the Subject and
    - waits for a message of the Subject before updating itself

Observer

- Collaboration - Example



Observer

- Applicability
  Use the Observer Pattern if
  - an abstraction has two points of view with corresponding dependencies
  - an encapsulation allows variation and independent reuse
  - the change of an object causes follow-up changes of other states; unknown how many objects are affected
  - an object should be capable to inform others without deep know-how about these objects (loose coupling)

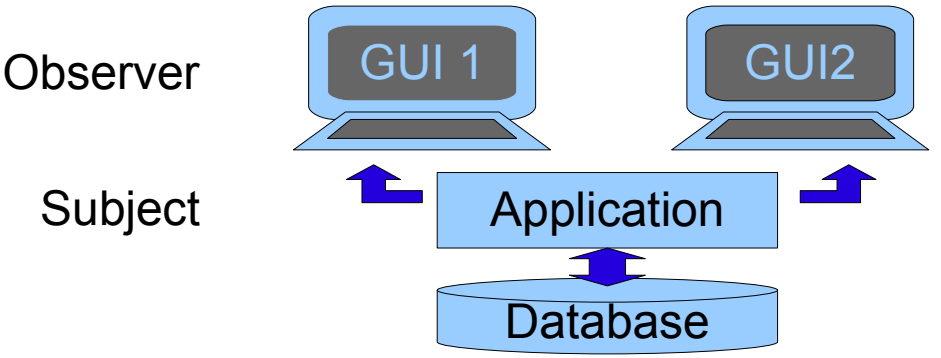
Observer

- Consequences

+ ***Subject***s and ***Observer*** are loose coupled – so they could vary independently
  → Independent reuse

+ Minimal coupling mean for the ***Subject***:
  - only one list of ***Observer*** (simple interface)
  - concrete classes of the ***Observer*** are not known

  → Level comprehensive collaboration possible



Observer

- Consequences

+ Broadcast principle
  Message of a ***Subject*** has no direct receiver
  - An update message will be sent to all registered ***Observer***
  - The only responsibility of the Subject: Sending messages
  - Observer work on the message in their own responsibility

Observer

- Consequences
  - Unexpected updates
    - ***Observer*** don't know each other
    - This could result in complex dependencies so that even harmless changes lead to update cascades
    - The minimal observer implementation does not provide information what exactly has been changed → Extensibility

Observer

- Implementation

  1)Who stores the Mapping between Subject and Observer?

  - ***Alternative 1***
    Subject: Efficient, but too expensive with many Subjects and few Observer; *example:*

```
import java.util.*; // because of Iterator, Vector

public class Subject {
  private Vector observers = new Vector();

  public void attachObserver(Observer anObserver) {
    observers.add(anObserver);
  }

  public void detachObserver(Observer anObserver) {
    observers.remove(anObserver);
  }

  public void notifyObservers() {
    Iterator elements = observers.iterator();
    while (elements.hasNext()) {
          ((Observer)elements.next()).update();
    }
  }
}
```

method has to be implemented by every Observer

Observer

- Implementation

  1)Who stores the Mapping between Subject and Observer?

  - ***Alternative 2***
    Global repository (e. g. hash table):
    Slower access, but in circumstances resource saving, if only some of the potential Subjects got observed

Observer

1)Example: Implementation using a global repository

```
public class Subject {
    private static Hashtable observeables = new Hashtable();
    public void attachObserver(Observer anObserver) {
    Set observers = (Set) observeables.get(this);
    if (observers == null) {                        // first observer for this subject
        observers = new HashSet();          // so we have to create a collection first
        observeables.put(this, observers);
    }
    observers.add(anObserver);
}
public void detachObserver(Observer anObserver) {
    Set observers = (Set) observeables.get(this);
    if (observers != null) {
        observers.remove(anObserver);
    }
}
public void notifyObservers() {
    Set observers = (Set) observeables.get(this);
    if (observers != null) {
        Iterator elements = observers.iterator();
        while (elements.hasNext()) {
            ((Observer) elements.next()).update();
        }
    }
}
```

Observer

- Implementation

  2)Observer observes many Subjects
  Question: Which Subject sent notification?
  → Extension of the `update` interface
  to identify the causing Subject

Observer

- Implementation

  2)Example: Extension of the `update` interface

Subject

```
public void notifyObservers() {
     Iterator elements = observers.iterator();
     while (elements.hasNext()) {
           ((Observer)elements.next()).update(this);
     }
}
```

Subject passes itself

Client

```
public void update(Subject aSubject) {
    if (aSubject instanceof Person) {
        // do some person related stuff
    }
    else {
        // do some other stuff
    }
}
```

Observer

- Implementation
  - 3)Who initiates `update`?
    - Subject after status change
      - All setter methods call `notify` after a change
        - \+ clear responsibilities
        - – many status changes → many updates
        - – inefficient by redundant intermediate updates
        - – „Flattering“
    - Clients control
      - Controlled `notify` request calls on demand
        - \+ More efficient, as status changes could be collected
        - – Error prone in implementation

Observer

- Implementation

  4)Hanging references to deleted Subjects
  If a Subject gets deleted you have to handle the references to the Observers

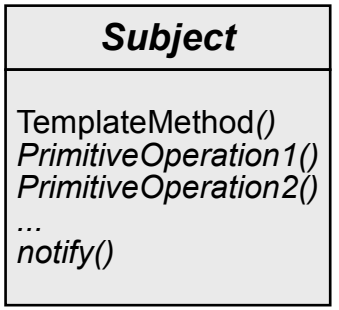
  - Bad idea: Ordinary deleting Observer
    - Dependency to other objects possible
    - Observer could listen to other Subjects
  - Better: Notify Observer, so they could decide what to do

Observer

- Implementation

  5)Ensuring the consistent state of the subject, before `notify()` gets triggered

  - Example: Identity and password
  - Potential problem in extending of setter methods in subclasses
  - Idea: Using the Template Method in the abstract Subject Class, where notify() is the last operation in the template method



Observer

- Implementation

  6) How much information does the subject offer in the `update` method?

  - Push model
    Assumption: Subject knows *something* about Observers need
    - Subject sends all detailed information to all Observer
  - Pull model
    Assumption: Subject ignores its Observers
    - Subject sends only a minimal `notify()`
    - Observer demands for more information if required

Observer

- Implementation

  7)Registration only for specific information

  - Observer registers only for special aspects
    `void Subj::Attach(Observer*, Aspect& i);`
  - Notification depending on parameter: Subject supplies changed aspect to observers in the `update` operation
    `void Obs::Update(Subject*, Aspect& i);`
  - Implementation effort is worth if there are many possible aspects

Observer

- Implementation

7)Example: Registration only for specific information

Subject

```
public void notifyObservers(String name) {
    Iterator elements = observers.iterator();
    while (elements.hasNext()) {
        ((Observer)elements.next()).update(this, name);
    }
}
```

Client

```
public void update(Subject aSubject, String attributeName) {
    if (aSubject instanceof Person) {
        if (attributeName.equals("firstName")) {
            // update first name related data
        }
    ...
    }
}
```

Observer

- Implementation

  8)Encapsulating complex update semantics

  - A Change-Manager could be established to handle complex dependency relationships between subjects and observers
  - Its responsibilities
    - Mapping of a subject to its observers
    - Defining a particular update strategy
    - Updating of all dependent observers at the request of a subject
  - The Change-Manager is an instance of the Mediator pattern

Observer

- Implementation

  9)Combination of Observer and Subject in a parent class in programming languages without multiple inheritance

  - An object could have the Subject and Observer role at the same time
  - makes Observer chains possible
  - realized in Smalltalk

Observer

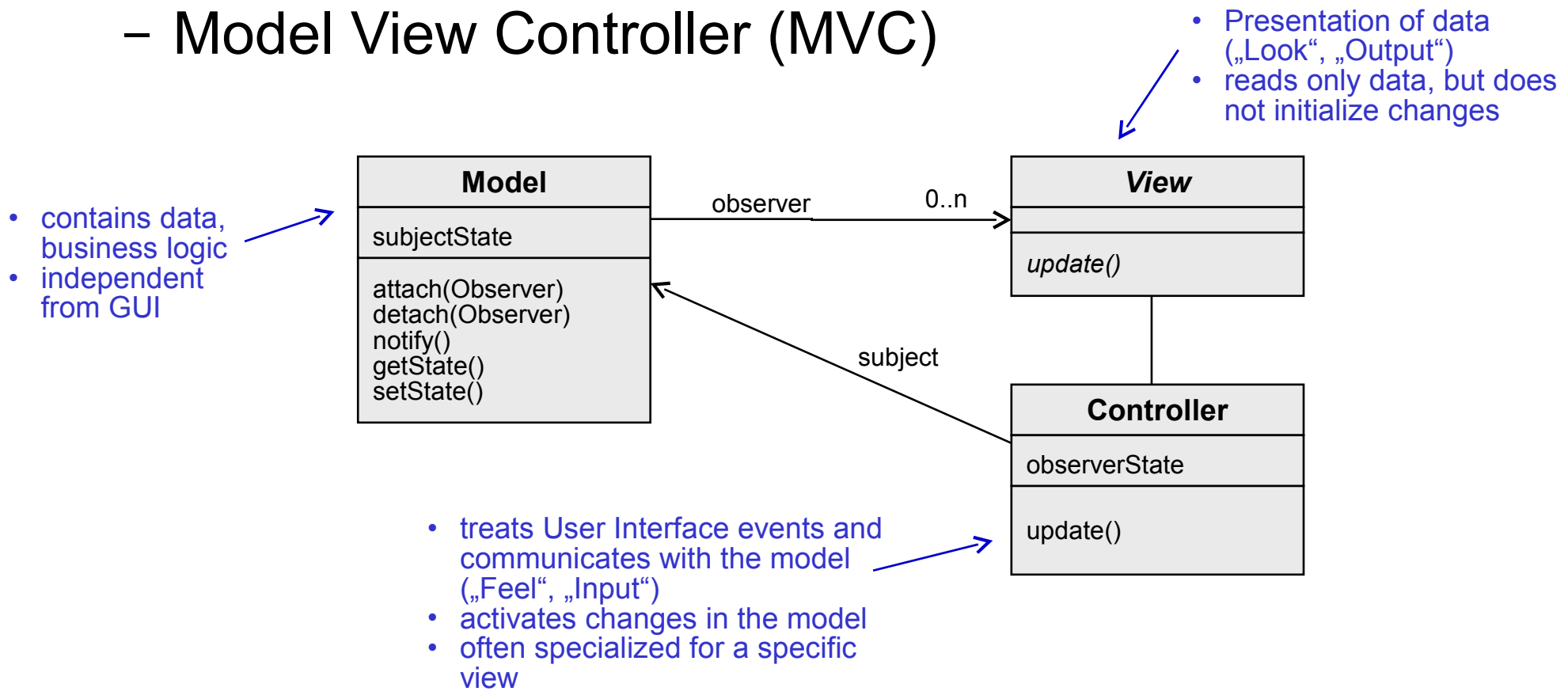
- Implementation
  9)Java offers a standard implementation with
    - Class `java.util.Observable` (meaning 'Subject') and
    - Interface `java.util.Observer`
  - Problem: Only usable, if the corresponding Subject class is not already subclass of another parent class

Observer

- Known Uses (see [GHJ+95])
  - MVC (Model-View-Controller) paradigm, e. g.
    - Smalltalk
    - Jakarta Struts
    - Java Swing
      - `JComponent`
      - `PropertyChangeListener`
  - Interviews, Andrew Toolkit, Unidraw
  - ET++, THINK class library
  - Web architectures

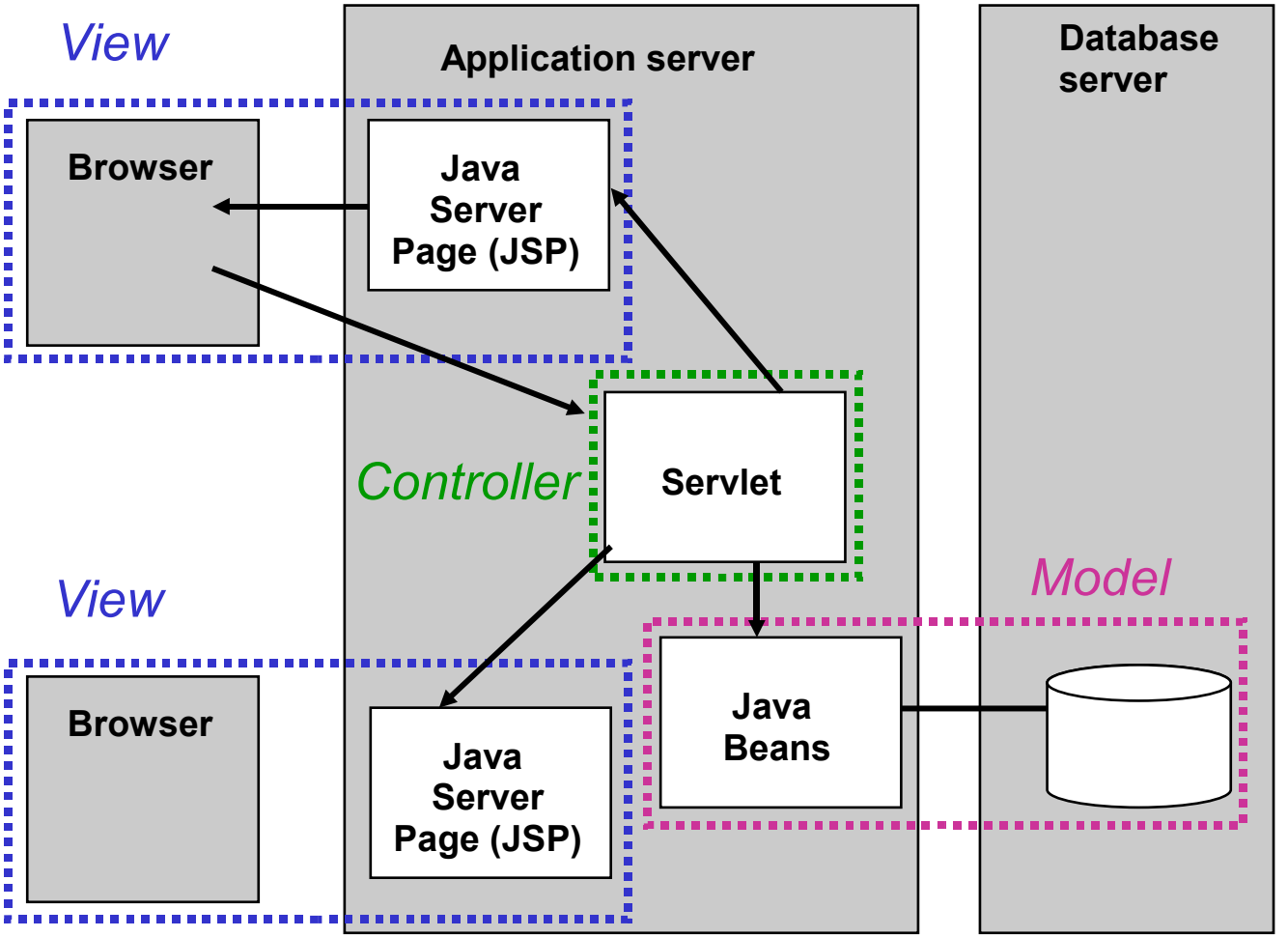
Observer

- Known Uses
  - Model View Controller (MVC)



Observer

- Known Uses
  - Typical web application



Observer

- Related Patterns
  - Mediator
    - The Change-Manager described in Implementation 8) acts as a Mediator
  - Singleton
    - The Change-Manager could be realized as a singleton to make it unique and global accessible

Mediator

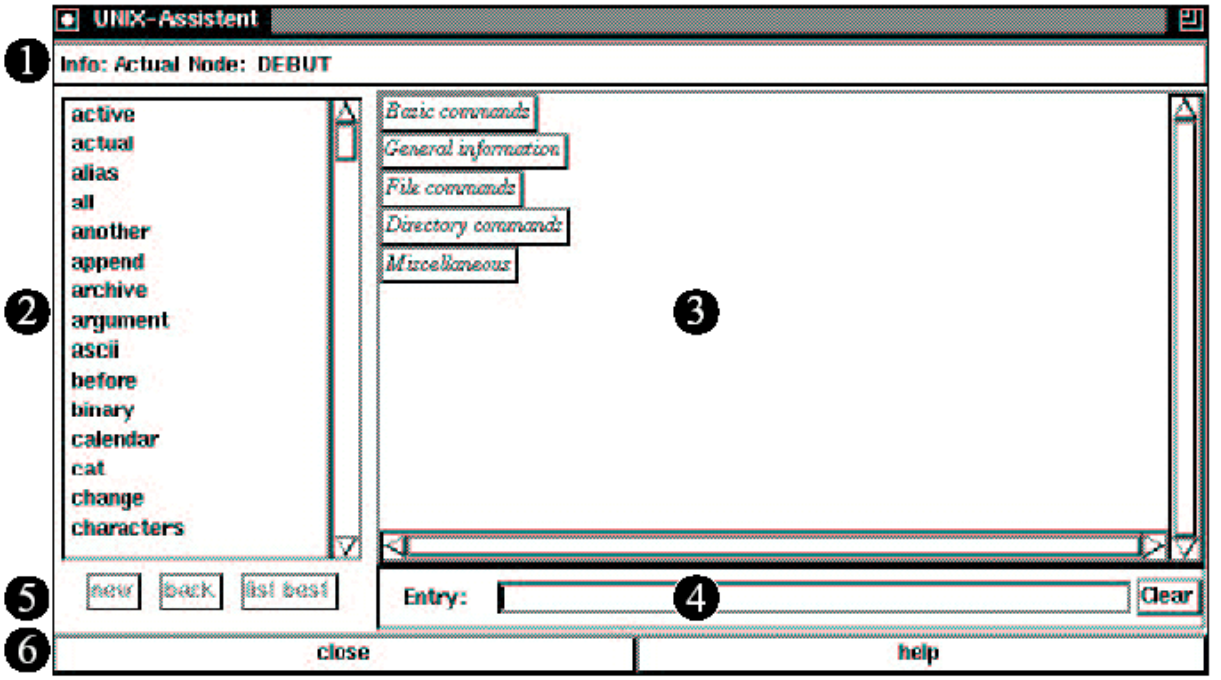
- Intent:
  - defines an object to encapsulate the interaction of several objects belonging together
  - supports loose coupling, avoiding explicit referencing of multiple objects among each other
  - lets vary interaction of objects independently
  - allows a flexible, independent interaction of the objects among themselves
  - a Mediator acts like an information broker
  - ... is a Behavioral Pattern

Mediator

- Motivation
  - Object oriented design promotes the distribution of behavior between different objects
  - A complex manifold connection structure between objects is possible – in the worst case any object is combined with any object
    - The system is acting like a monolith
    - A change of behavior is difficult to achieve, as the behavior could be distributed about many objects

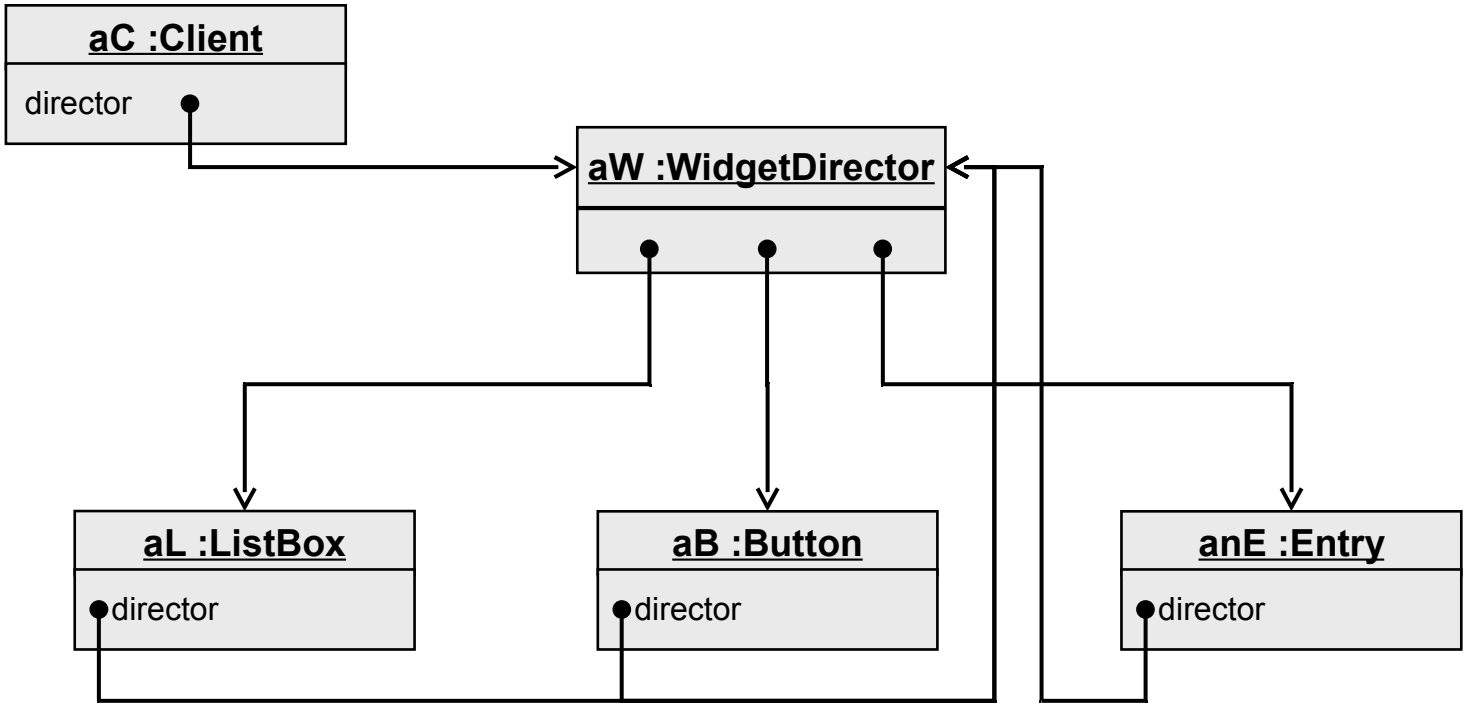
Mediator

- Example
  Implementation of dialog boxes in a graphical user interface with
  - Buttons
  - Menus, and
  - Entry fields
- Dependencies:
  - Button gets disabled under determined condition



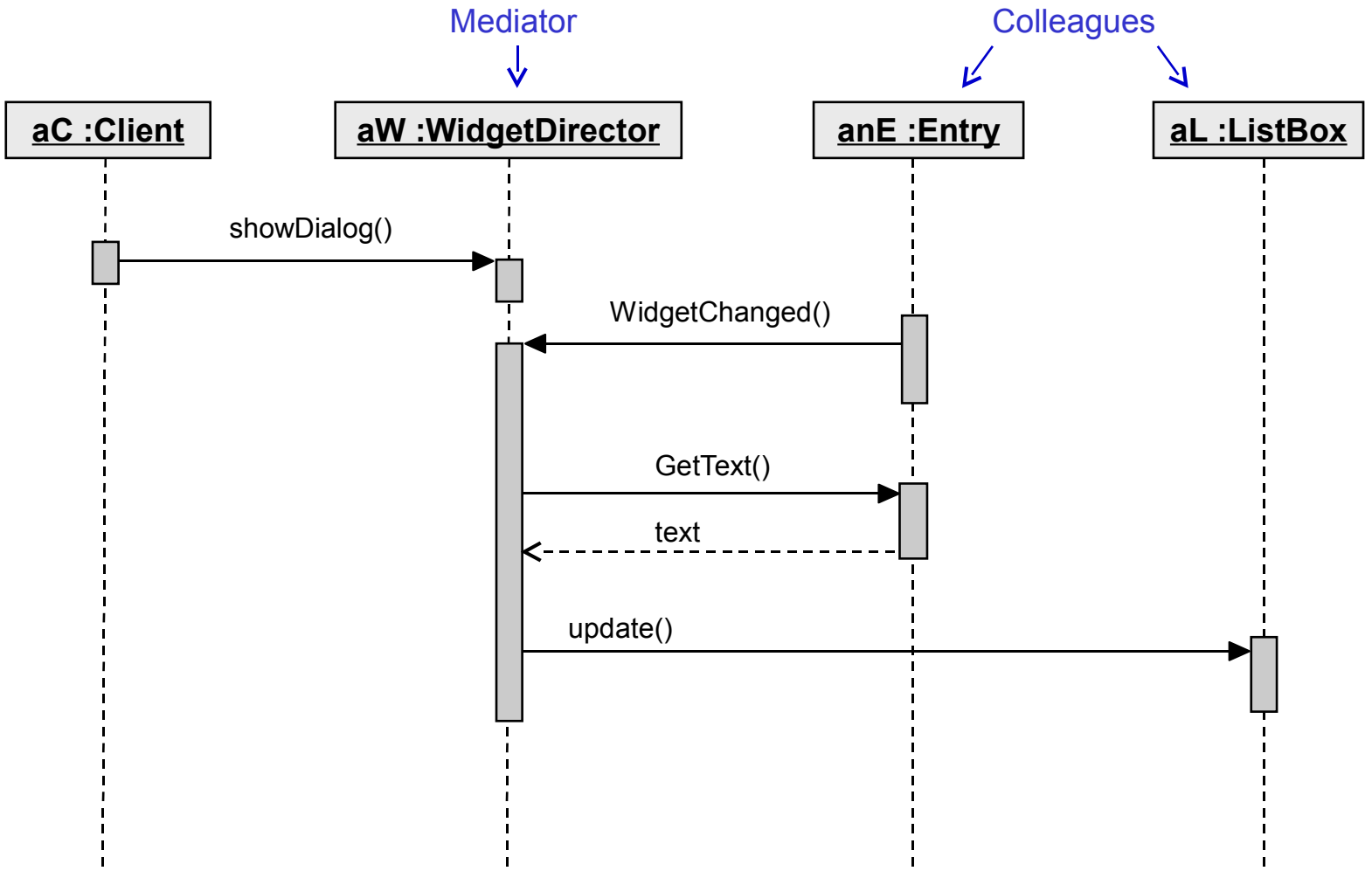
Mediator

- Example – Object Diagram



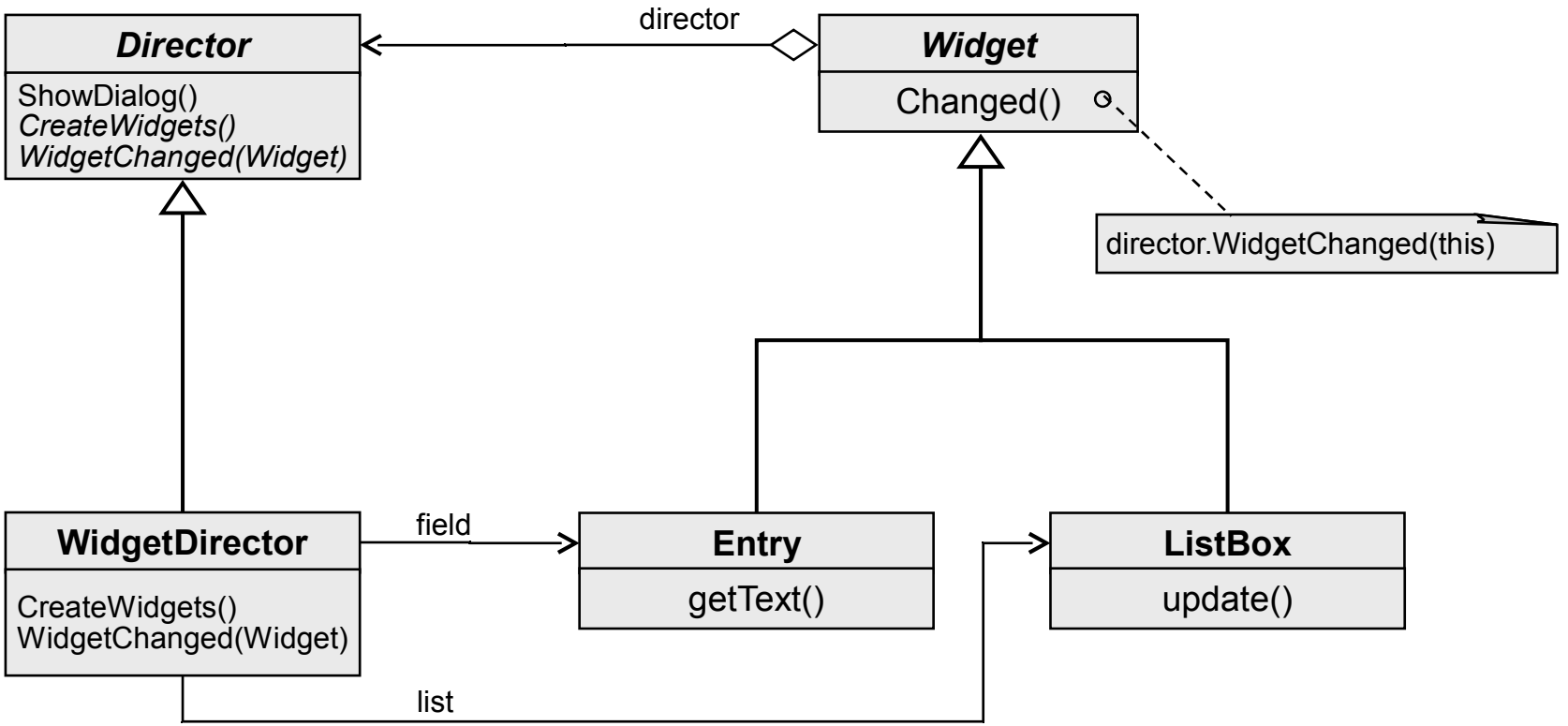
Mediator

- Example – Sequence Diagram



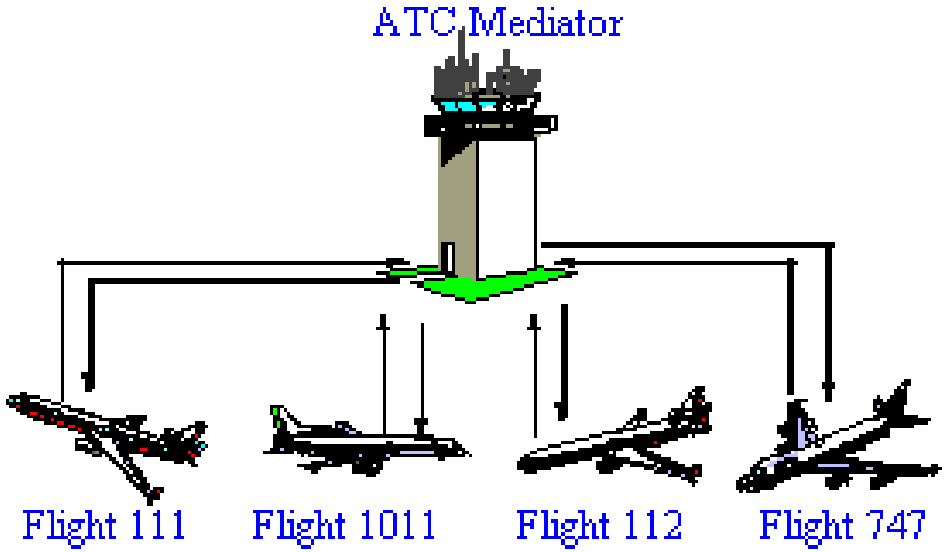
Mediator

- Example – Class Diagram



Mediator

- Example [Hus08]



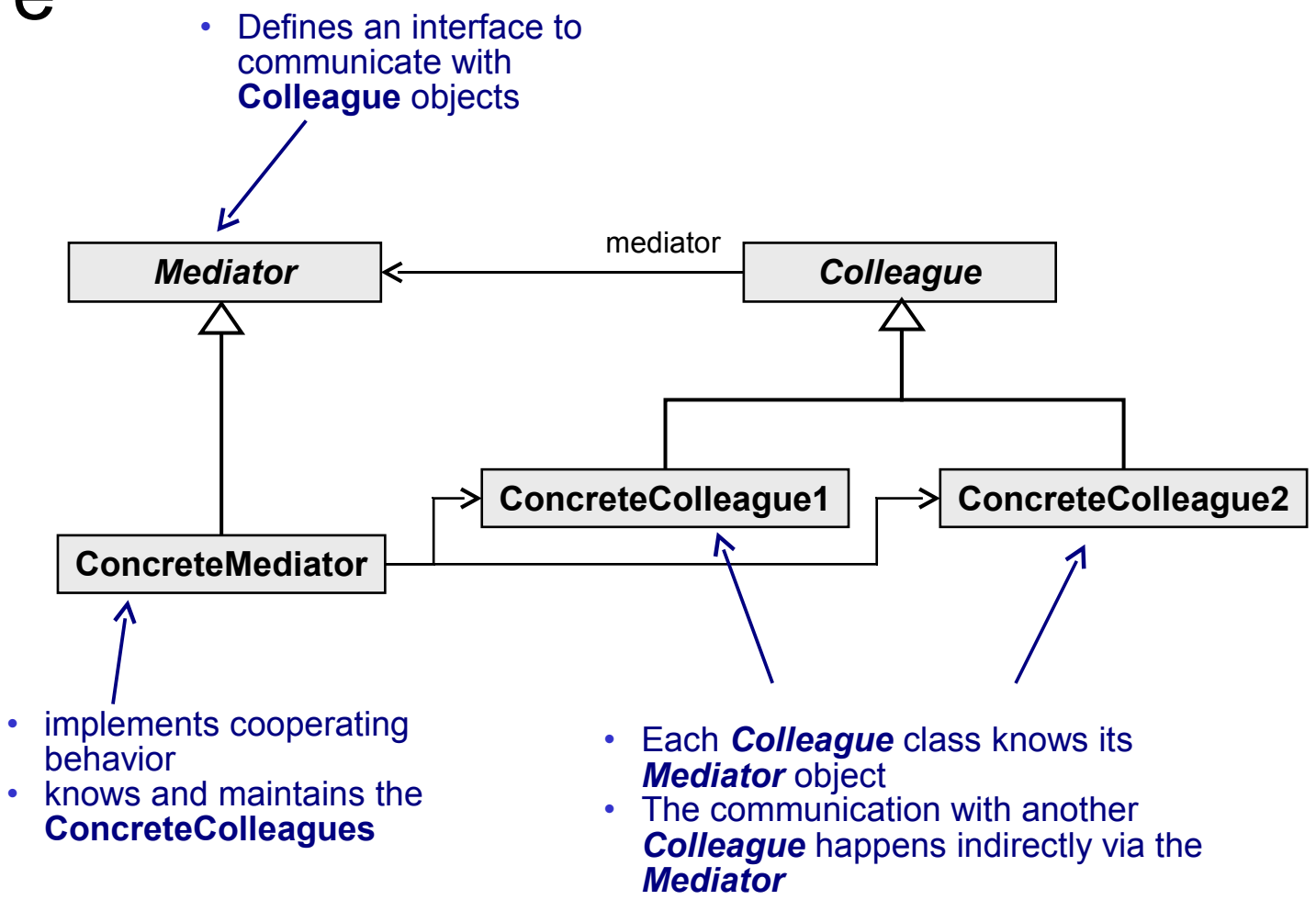
The control tower at a controlled airport demonstrates this pattern very well.
The pilots of the planes approaching or departing the terminal area communicate with the tower rather than explicitly communicating with one another.
The constraints on who can take off or land are enforced by the tower.
It is important to note that the tower does not control the whole flight.
It exists only to enforce constraints in the terminal area.
[Michael Duell, "Non-software examples of software design patterns", Object Magazine, Jul 97, p54]

Mediator

- Applicability
  Use the Mediator Patter, when ...
  - a set of objects communicates in a specified but complex kind among each other.
    The resulting dependencies are
    - not structured
    - difficult to understand
  - reuse of an object is difficult, because it is referencing to and communicating with many other objects
  - a distributed behavior between different classes should be controlled without a set of subclasses

Mediator

- Structure



Mediator

- Collaboration
  - ***Colleague***s send and get requests from a ***Mediator*** object
  - The ***Mediator*** implements the cooperating behaviour with coordinating the requests between ***Colleague***s

Mediator

- Consequences

+ Limiting subclasses
  - Localization of behavior, that otherwise would be distributed between different objects
  - A change of this behavior requires only a subclass of the ***Mediator***. ***Colleague***s could be reused as they are

Mediator

- Consequences

+ Decoupling of ***Colleague***s
  - With loose coupling ***Mediator*** and ***Colleague*** classes could be modified independently from each other

Mediator

- Consequences

+ Simplification of the objects protocols.
  Only the ***Mediator*** knows the ***Colleague***s, so
  - 1:n relationship (***Mediator*** ⇔ ***Colleague***) instead of
  - n:n relationship (***Colleague***s among each other)

  The number of interconnections is reduced.
  As a results, the system behavior is easier to ...
  - understand
  - maintain
  - extend

Mediator

- Consequences

+ Abstraction of the cooperation of the objects. Focus is how the objects interact with each other – independent from their individual behavior

+/– Centralization of control

  - Complexity of the communication gets transferred from the system into the ***Mediator***
  - Typically the ***Mediator*** gets more complex as every individual ***Colleague***
  - The ***Mediator*** itself could get monolithic and for this reason complex and difficult to maintain

Mediator

- Implementation
  - Omit the abstract ***Mediator***
    - If the ***Colleague***s work only with one **ConcreteMediator** an additional abstraction is not necessary

Mediator

- Implementation
  - Communication between ***Mediator*** and ***Colleague***s ... is necessary, if an ‚interesting' event happens; ideas:
    - Implementation of the Mediator as Observer in using the Observer Pattern
    - Realization of a specific communication interface inside the Mediator to achieve a ‚more direct' communication to the Colleagues; e. g. with hand-off of a self-reference

Mediator

- Known Uses (see [GHJ+95])
  - ET++ and the THINK C library use „Director“ objects, acting as Mediators between widgets
  - Smalltalk/V for Windows – the application architecture is based on a Mediator structure
  - Coordination of complex updates; example:
    - In using of the observer pattern a „change manager“ could mediate between Subject and Observer to avoid redundant actualizations
    - The change manager gets informed as soon as an object is changing: He coordinates all the necessary updates in informing only the objects depending on this object

Mediator

- Related Patterns
  - Different trade-offs how to decouple senders and receivers [p347, GHJ04]
    - Chain of Responsibility
      passes a sender request along a chain of potential receivers
    - Command
      normally specifies a sender-receiver connection with a subclass.
    - Mediator
      has senders and receivers reference each other indirectly
    - Observer
      defines a very decoupled interface that allows for multiple receivers to be configured at run-time

Mediator

- Related Patterns
  - Mediator and Observer are competing patterns [p346, GHJ04]
    - Observer distributes communication by introducing "observer" and "subject" objects
    - Mediator encapsulates the communication between other objects
    - It seems to be easier to make reusable Observers and Subjects than to make reusable Mediators
  - Mediator can leverage Observer for dynamically registering colleagues and communicating with them [p282, GHJ04]

Mediator

- Related Patterns
  - Difference between Mediator and Facade [p193, GHJ04]
    - Mediator is similar to Facade in that it abstracts functionality of existing classes.
    - Mediator allows cooperative behavior between objects and the protocol is multi directional
    - Facade abstracts a subsystem of objects to offer an easier interface. It is not known by subsystem classes. The protocol is unidirectional (communication only to the subsystem, not vice versa)