## Design Patterns

- Similar to analysis patterns
  - but about design issues
  - domain independent
  - but not technology-independent
- A solution to a common design problem
- More specifically:
  - as in the Gang of Four book
  - small scale (a few classes)
  - not about architecture
  - object-oriented

## Levels of Design (1): Architecture

- Overall Structure
  - Big decisions
  - Modules
  - Distribution and concurrency
- Architectural styles
  - object-oriented
  - blackboard
  - client-server
  - peer-to-peer
- Connectors
  - determine how components talk to each other
  - examples:
    - network protocols
    - message passing
    - procedure calls

## Levels of Design (2)

- Design Patterns
  - intermediate scale
  - interactions of a few classes or objects
  - often used in class libraries
    - e.g. Java Swing
- Class Design
  - design of a single class
  - which operations and attributes
  - what classes it inherits from
  - one class or two?

## Gang of Four Book

- Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides: Design Patterns, 1995
- Their definition:
  - not about things that can be represented as classes (lists, hashtables): class libraries
  - not complex design for the whole application: architecture
  - "design patterns are descriptions of communicating objects and classes customized to solve a general design problem in a particular context"
- Classification of design patterns
  - class-based versus object-based
  - creational, structural, behavioral
- Source of most patterns: Gui libraries
- Structure for describing design patterns
  - about 10 pages per pattern

## Purposes of Design Patterns

- Finding solutions
  - make design easier
  - tested and tried solutions
  - experience of master designers
- Documenting a design
  - saying “design pattern X used” makes your design and code much more understandable
  - people will know why you did what you did
  - makes changes easier
    - reduces the risk that something will accidentally be broken
- Using class libraries
  - many good class libraries rely on design patterns
    - Java, C++ standard libraries
  - need to understand the patterns to get full benefit of libraries

## Some Design Patterns

- Creational
  - Abstract Factory (o)
  - Factory Method (c)
  - Singleton (o)
- Structural
  - Adapter (o, c)
  - Composite (o)
  - Decorator (o)
  - Façade (o)
  - Proxy (o)
- Behavioral
  - Command (o)
  - Interpreter (c)
  - Iterator (o)
  - Memento (o)
  - Observer (o)
  - State (o)
  - Strategy (o)
  - Template Method (c)
  - Visitor (o)

- o = object based
- c = class based

## Design Pattern: Observer

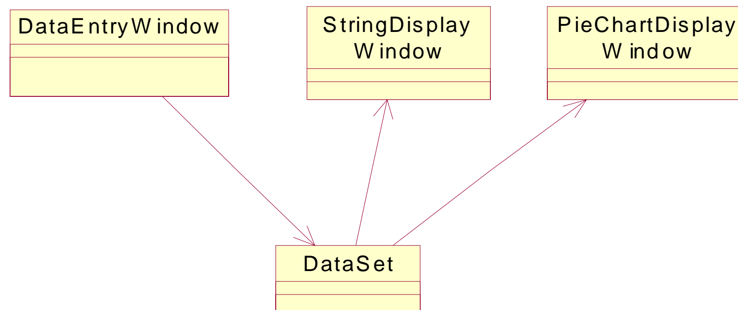
- Purpose:
  - Create a 1-to-n dependency between objects, so that if one object changes its state, all others can be notified. Then they can update themselves.
- Other names:
  - publish-subscribe system
  - listener
  - event system
- Examples:
  - Model-View-Controller architecture (MVC): each model (1) can have any numbers of views (n) that represent it on the screen
  - caching: when the cached object changes, all caches that store it need to be updated
  - file system: when someone saves a file, all directory listings in which it occurs need to update its size

## Observer Example: Requirements

- A simple application that stores some simple data
- Window 1:
  - TextField to enter a set of numbers (percentages)
- Window 2:
  - displays the numbers entered as a list of strings
- Window 3:
  - displays the numbers entered as a pie chart
- Gui should be extensible, changeable
  - it should be easy to change the windows later
  - it should be easy to add more windows later

## Observer Example: Design

- Classes
  - Subclasses of JFrame (Swing class representing a window):
    - DataEntryWindow
    - StringDisplayWindow
    - PieChartDisplayWindow
  - To store the numbers:
    - DataSet



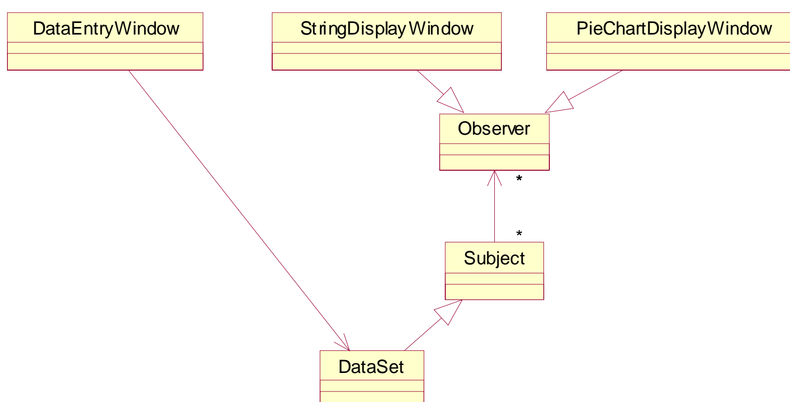

## Observer Example: Design

- What's wrong with this design?
  - bidirectional dependency between Gui and Data Model
  - contradicts the three-tier architecture
  - example: if a new display window is added, class DataSet has to be changed
- We need a solution that allows localized Gui changes
- You want to be able to add another display window without having to change the DataSet class

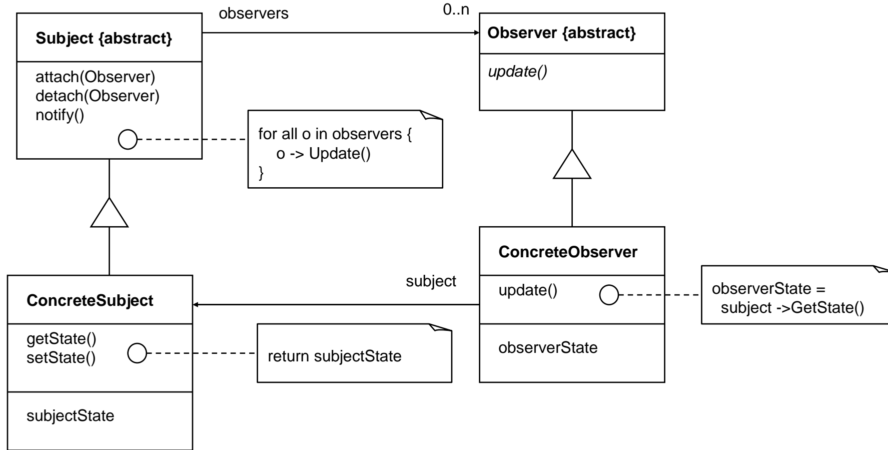
## Observer Example

- Solution: the Observer Design pattern
- Idea: the DataSet keeps a list of all the display windows
  - when a new display window is created, it adds itself to the list
  - when the DataSet needs to change the display windows, it notifies all the elements of the list
- Advantages:
  - DataSet class does not need to know about individual window classes anymore
  - It is possible to add a new display window class without having to modify the DataSet class

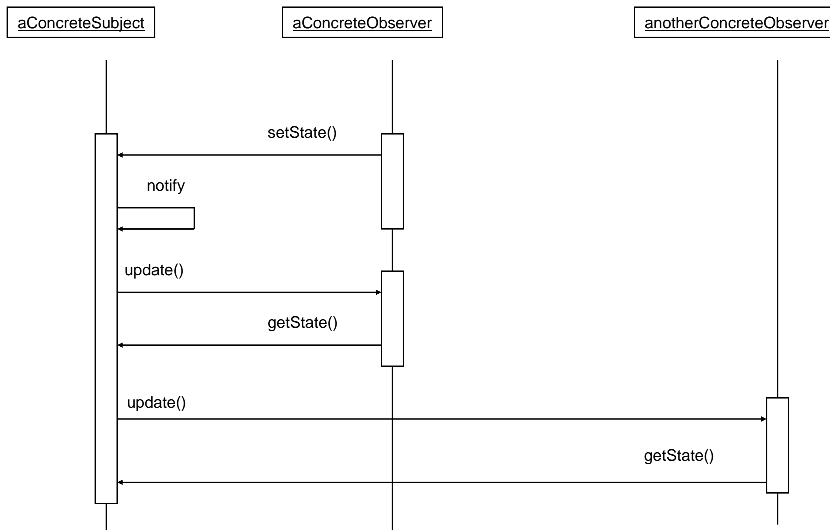
## Observer Example



# Observer: Abstract Class Diagram




# Observer: Sequence Diagram



## Observer: Sequence Diagram

- Subject has Observers
- an Observer changes Subject with setSate()
- Subject calls its own "notify()" operation
- Subject notifies all Observers
  - by sending "update()"
- Observers update themselves by asking the Subject for its new state
  - by sending "getState()"

## Observer

- Use the Observer pattern when…
  - updating one object requires updating one or more other objects
    - updating the DataSet requires updating all the display windows
    - updating a file requires updating directory listings
  - when the updated objects does not need to know the exact type of the dependent objects
    - DataSet does not know that its Observers are display windows
    - when you want to send one message to several objects at once

## Notes about Observer Pattern

- Fits very well into three-tier architecture
  - Observer pattern ensures that business logic does not need to know about the Gui
- Sometimes known as Model-View-Controller pattern (MVC)
  - Subject = Model
  - Observer = View
- Supported by Java Standard Library
  - Subject = java.util.Observable
    - Class
    - operations to add, remove, notify Observers
  - Observer = java.util.Observer
    - Interface
    - only operation: void update( Observable o, Object arg )