

Advanced Object-Oriented Features

- Abstract class
- Interface
- Multiple inheritance
 - example
 - a BSU student is a student
 - a BSU student is a citizen of Indiana
 - but:
 - not every student is a citizen of Indiana
 - not every citizen of Indiana is a student
 - problematic
 - why?

Polymorphism Example

- abstract class Shape
 - double getArea() // return the area
- class Rectangle extends Shape
 - Rectangle(int upperLeftX, int upperLeftY, int width, int height) {...}
 - double getArea() { return width * height; }
- class Circle extends Shape
 - Circle(int centerX, int centerY, int radius) {...}
 - double getArea() { return Math.PI * radius * radius ; }
- getArea is specified by Shape
 - and implemented in two different ways by its subclasses
 - each implementation fulfills the spec
 - each implementation is different

6.4 More UML

- Just a few things we didn't have time to cover before
- Additional elements of class diagrams
- Extension mechanisms
- Packages
- Notes
- Deployment diagrams

Aggregation



- Aggregation is a specialized association
- "is part of" relation
 - a book is part of a library catalog
 - a library catalog has any number of books
- No really clear meaning
 - may make a diagram more readable
 - maps to the same code as any other association

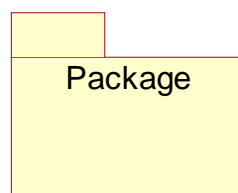
Composition



- Composition is a specialized aggregation
- But with a well-defined meaning!
- Example

```
class Rectangle {
    private Point[] p = new Point[ 4 ];
    //the elements of p are never given out
    Rectangle( int x, int y, int height, int width ) {
        p[ 0 ] = new Point( x, y );
        ... }
}
```
- Meaning: each part belongs to exactly one composite
 - When the composite is destroyed, the part is destroyed
 - Parts cannot be shared among composites

Packages



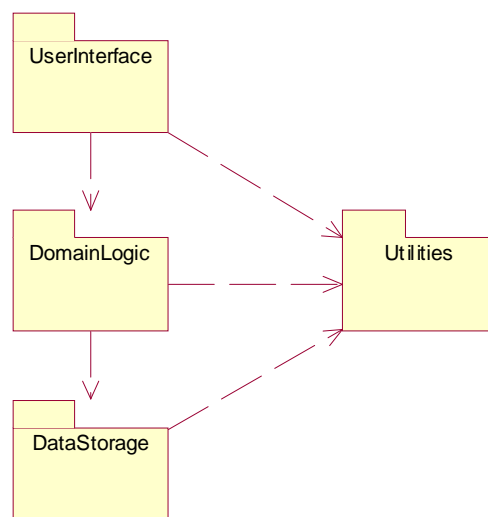
- A grouping mechanism
- A package can contain any kind of UML elements
- Two uses:
 - to hide part of a diagram
 - to show the relationship between large numbers of elements

Dependency

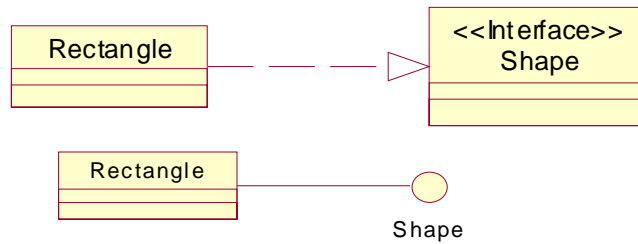


- A depends on B
- Any kind of “uses” relation between classes that is not an association or a generalization
- Examples:
 - A instantiates B (but does not keep the reference)
 - A has operations that have objects of type B as parameters
- Generally:
 - Class A cannot be used without class B
- Can also be used with packages

Example: Three-Tier Architecture

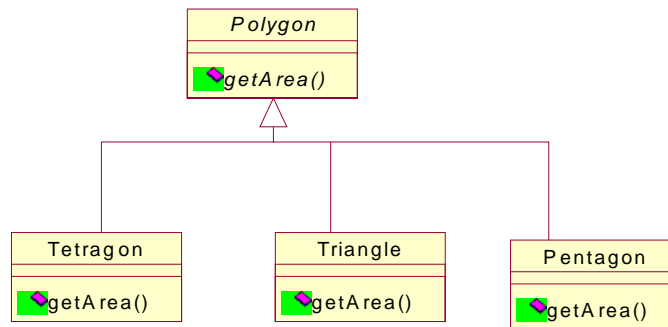


Interfaces



- Class Rectangle implements interface Shape
- Top: full representation
- Bottom: elided representation
 - no operations of the interface can be shown
- The relation between class and interface is a realization
- Interfaces are a kind of class in UML

Abstract Classes



- Abstract classes and operations
 - have names in italics
- By the way...
 - this is an other way to show generalizations
 - several arrows made into one

UML Extension Mechanisms

- Stereotypes
 - allow user-defined UML elements
 - for example:
 - «interface»
 - «user interface class»
 - those thingies are called guillemets
 - can also be represented through their own icons
 - such as a circle for interfaces
- Profiles
 - set of stereotypes for a purpose
 - for example: USDP profile
- Constraints
 - we already discussed those
 - {abstract}

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

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

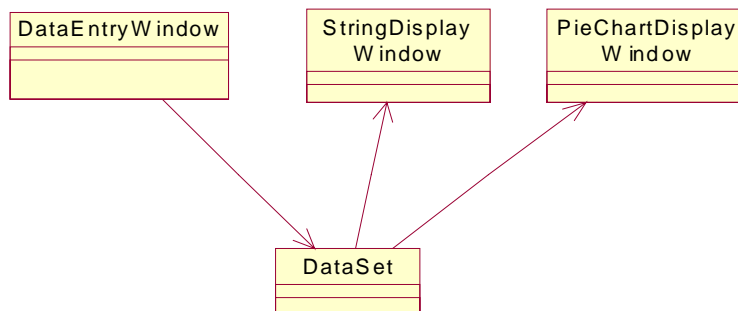
- Purpose:
 - Create a 1-to-n dependency between objects, so that if one object changes its state, all others can be notified. Then they can update themselves.
- Other names:
 - publish-subscribe system
 - listener
 - event system
- Examples:
 - Model-View-Controller architecture (MVC): each model (1) 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 (Java class representing a window):
 - DataEntryWindow
 - StringDisplayWindow
 - PieChartDisplayWindow
 - To store the numbers:
 - DataSet



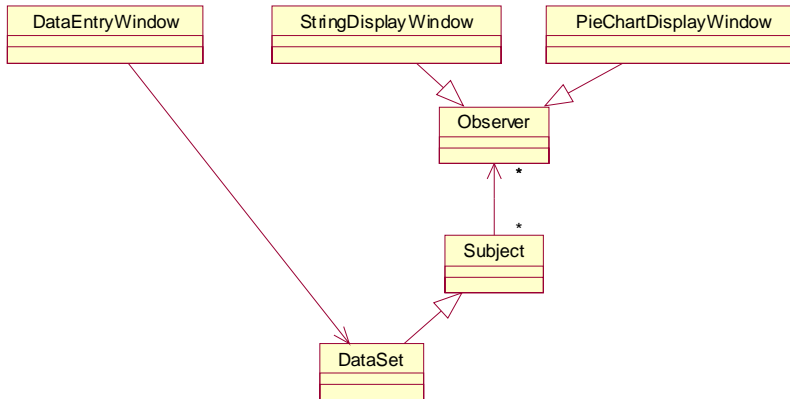
Observer Example: Design

- 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

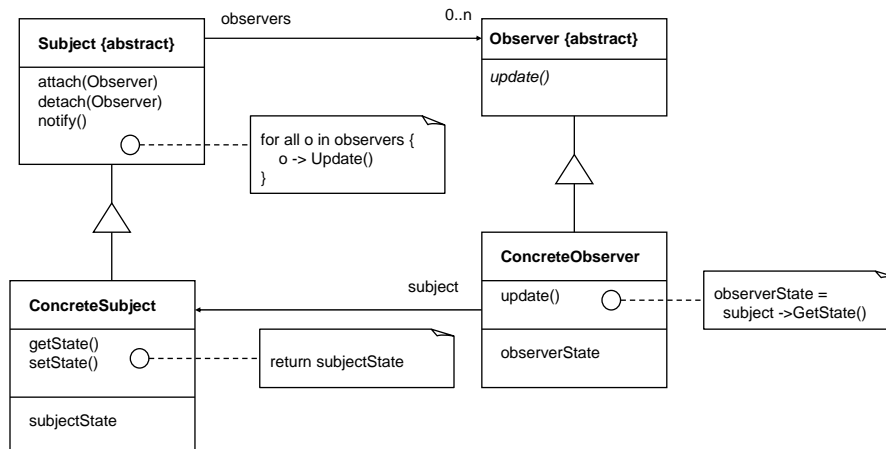
Observer Example

- 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

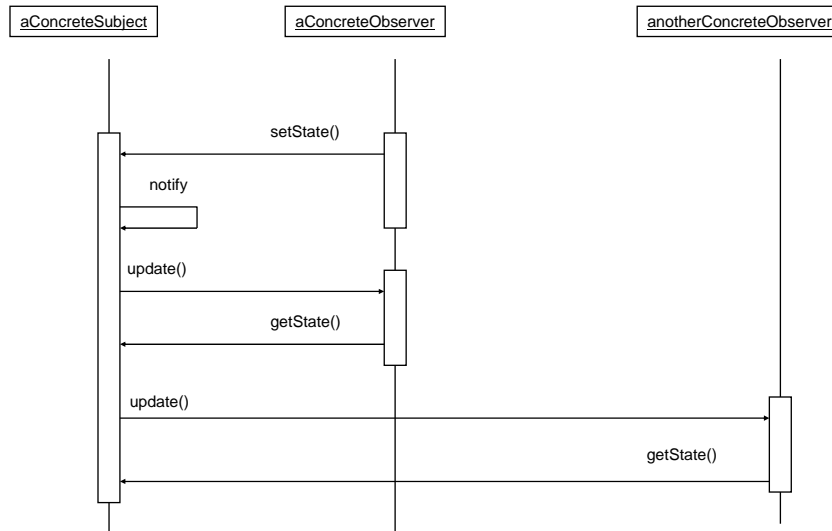
Observer Example



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)