Design Patterns

When reusing software components, the developer is constrained by the design decisions that have been made by the implementers of these components.

These design decisions range from algorithm design to interface design of the component.

If these design decisions conflict with your particular requirements then reusing the components is either impossible or introduce significant inefficiencies into your system.

http://www.dofactory.com/Patterns/Patterns.aspx

Design Patterns

One way to solve this problem is to reuse more abstract designs that do not include implementation details. These are then implemented specifically to fit your application requirements.

These are certain common design patterns which can be reused when designing many systems.

In software design, design patterns have been inevitably associated with object oriented design. They often rely on object characteristics such as inheritance and polymorphism to provide generality.

Elements of a Design Pattern

Pattern Name – A name that is meaningful reference to the pattern.

Problem description – A description of the problem area that explains when the pattern may be applied.

Solution description - A solution description that describes the different parts of the design solution, their relationships and responsibilities.

Consequences – The results and trade-offs of applying the pattern. This is used to help designers understand whether or not a pattern can be effectively applied in a particular situation.

Design Patterns – An Example

General Hierarchy

Name of the pattern - General Hierarchy

Problem decription-

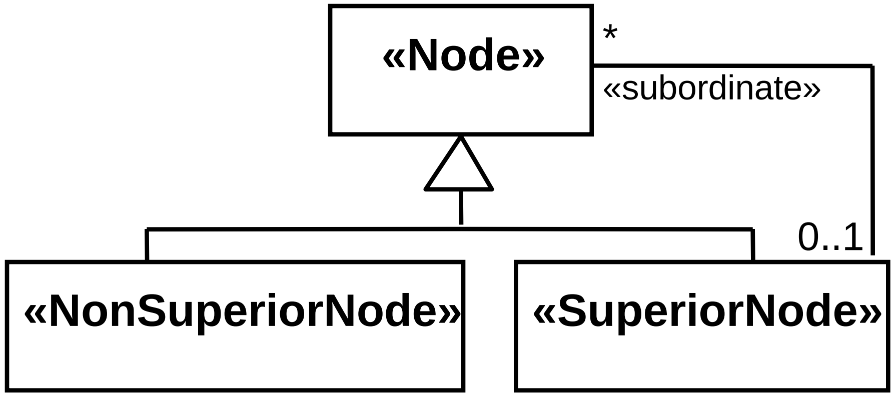
- One child class has an additional relationship with the parent class.

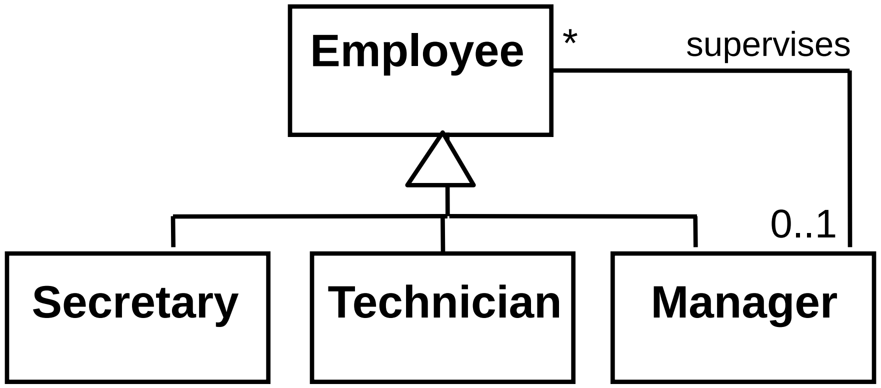
Design description-

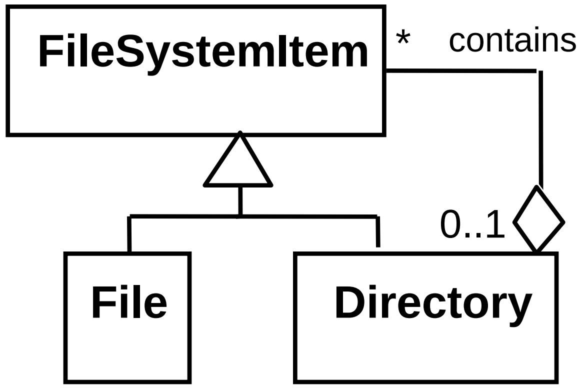
- You want a flexible way of representing the hierarchy
  - that prevents certain objects from having subordinates
- All the objects have many common properties and operations
- Objects in a hierarchy can have one or more objects above them (superiors),
  - and one or more objects below them (subordinates).

design description - General Hierarchy

– ***Solution:***







Consequences – General Hierarchy

You have to account for the fact that all the objects share common properties and operations.

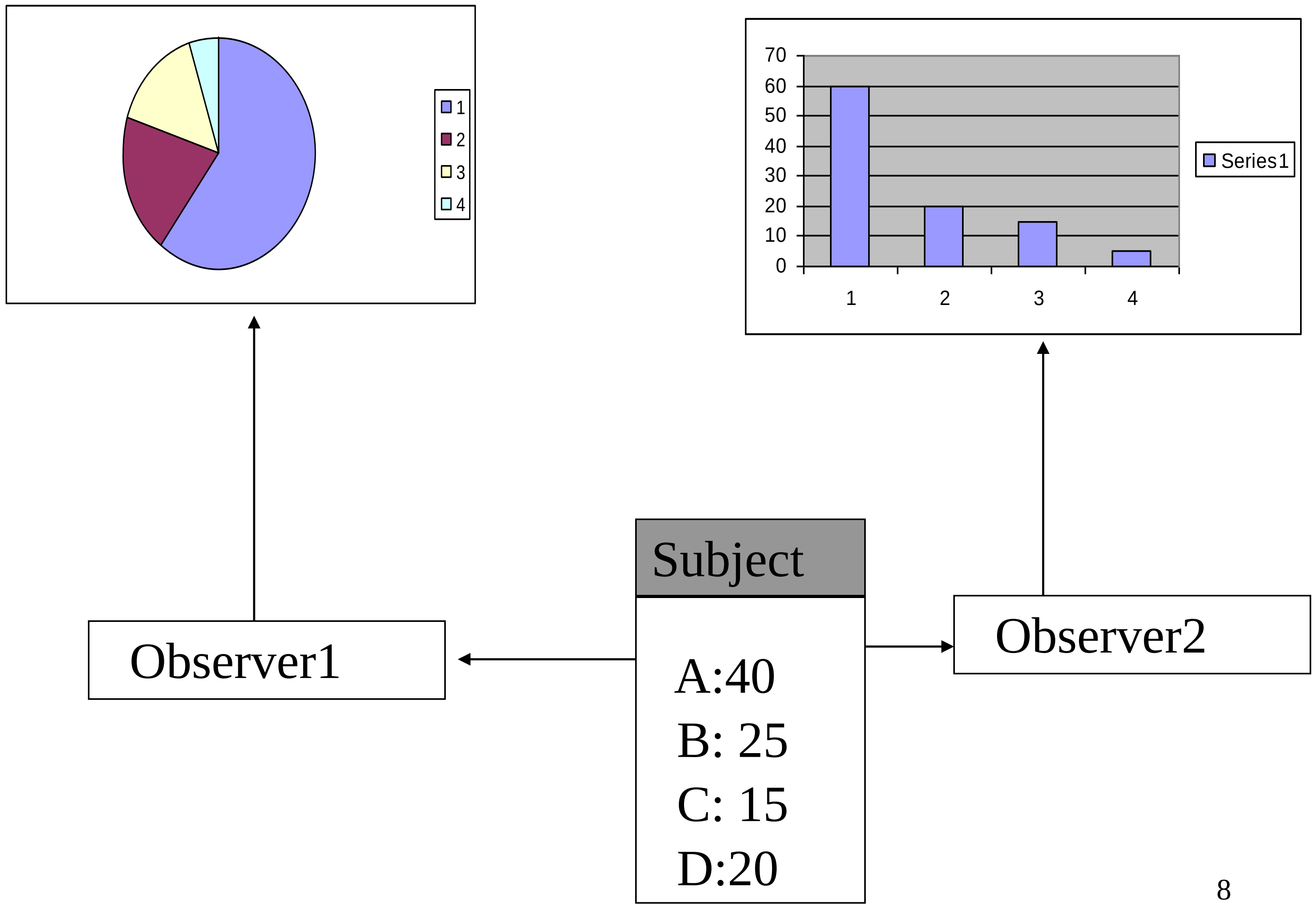
Another example of a standard design patterns

Pattern Name - Observer

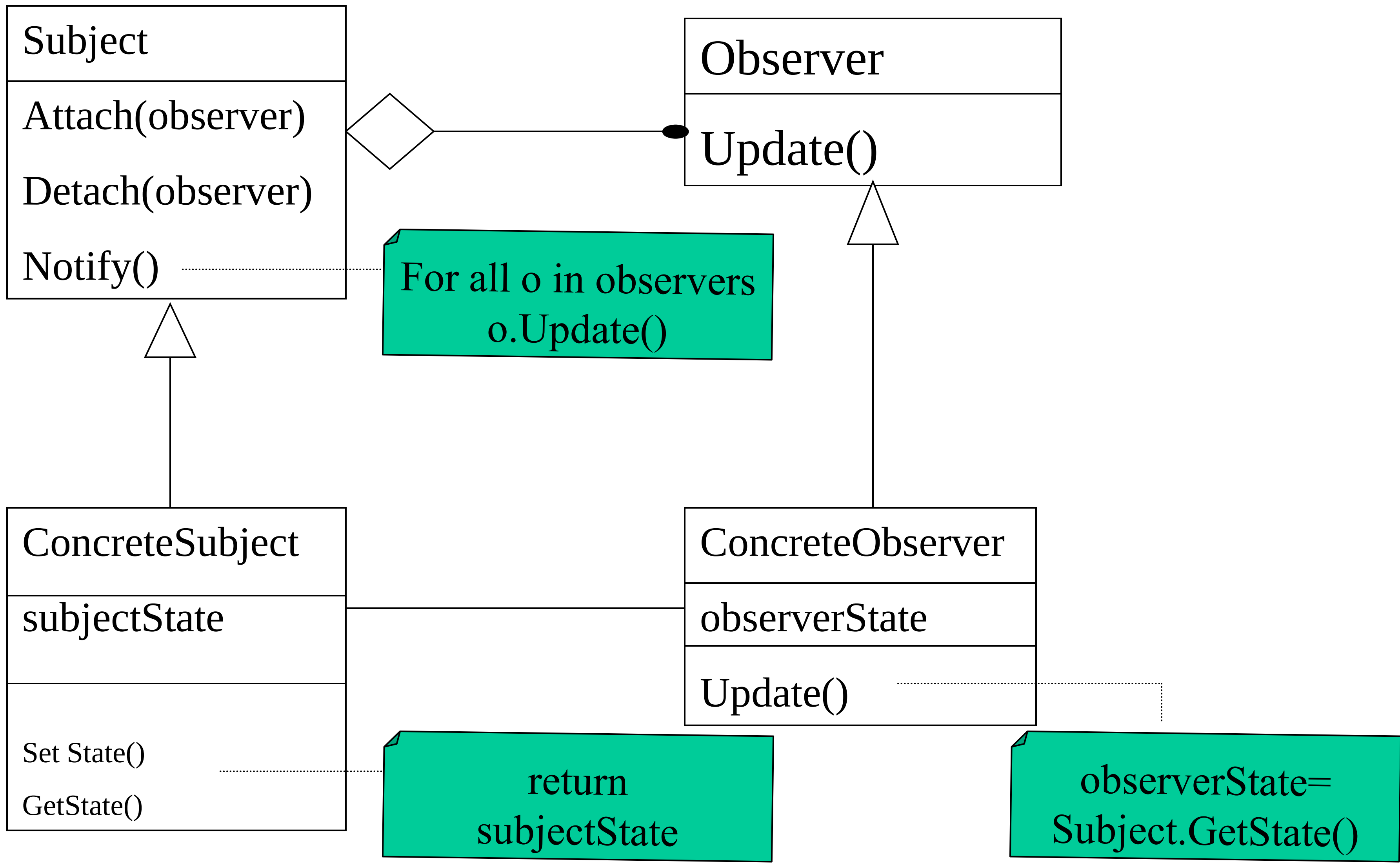
Problem description - In many situations, it is necessary to provide multiple displays of some state information such as a graphical display and a tabular display. Not all of these may be known when the information is specified. All alternative presentations may support interaction and , when the state is changed, all displays must be updated.

This pattern may be used in all situations where more than one display format for state information may be required and where it is not necessary for the object that maintains the state information to know about the specific display formats used.

Multiple Displays



Design description -



Design description –

The structure of the pattern is shown in the above figure. This defines two abstract objects – Subject and Observer, and two concrete objects - ConcreteSubject and ConcreteObserver, which inherit the attributes of the related abstract objects. The state to be displayed is maintained in ConcreteSubject which also inherits operations from Subject allowing it to add and remove Observers and to issue a notation when the state has changed.

The ConcereteObserver maintains the copy of the state of ConcereteSubject and implements the Update() interface of Observer which allows these copies to be kept in step. The ConcreteObserver automatically displays its state – this is not normally an interface operation.

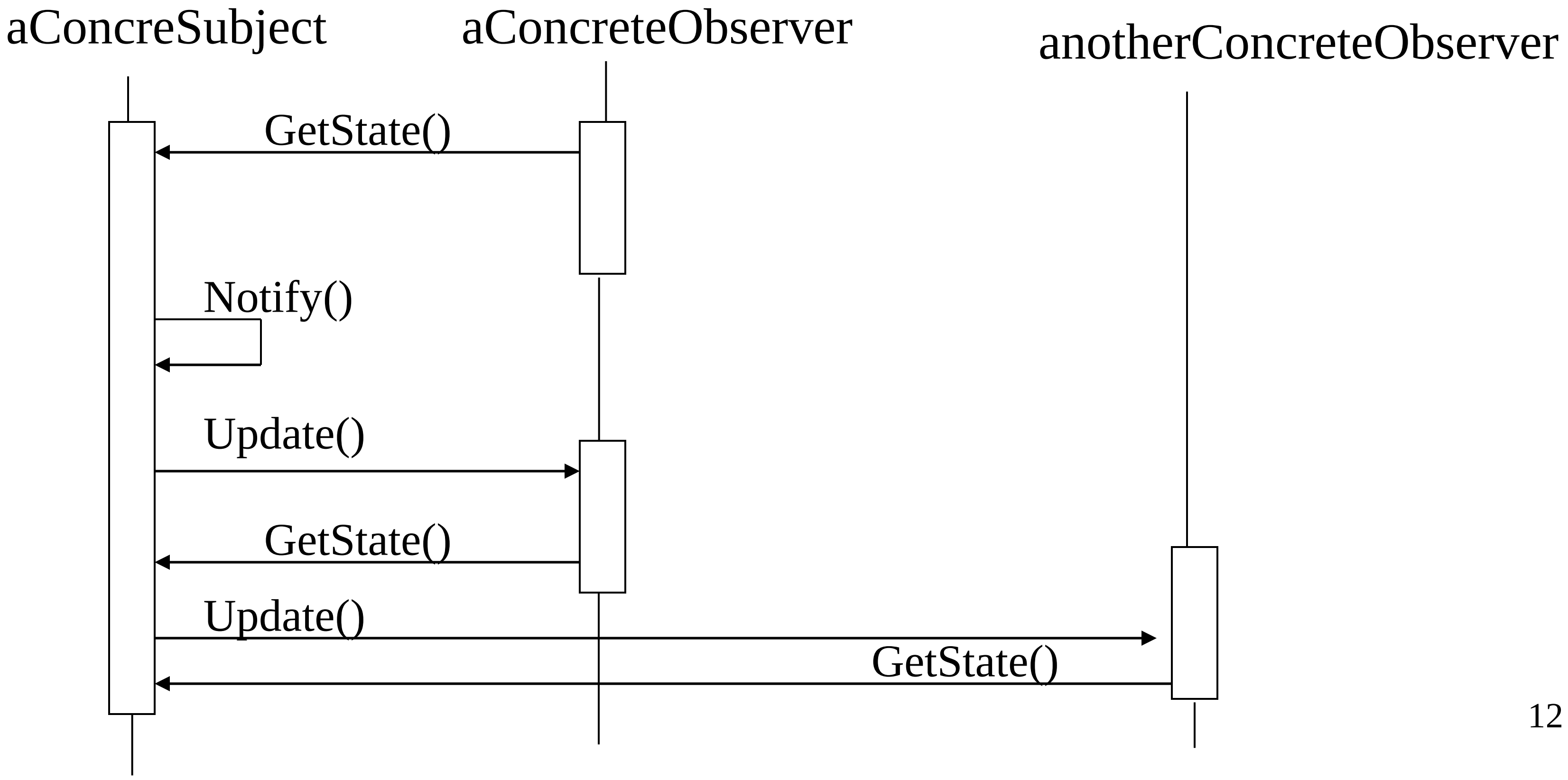
Consequences – Observer

The subject only knows the abstract observer and does not know details of the concrete class. Therefore, there is minimal coupling between these objects. Because of this lack of knowledge, optimisations that enhance display performance are impractical. Changes to the subject may cause a set of linked updates to observers to be generated, some of which may not be necessary.

Collaborations - Observer

ConcreteSubject notifies its observers whenever a change occurs that could make its observers' state inconsistent with its own.

After being informed of a change in the concrete subject, a ConcreteObserver object may query the subject for information. ConcreteObserver uses this information to reconcile its state with that of the subject.

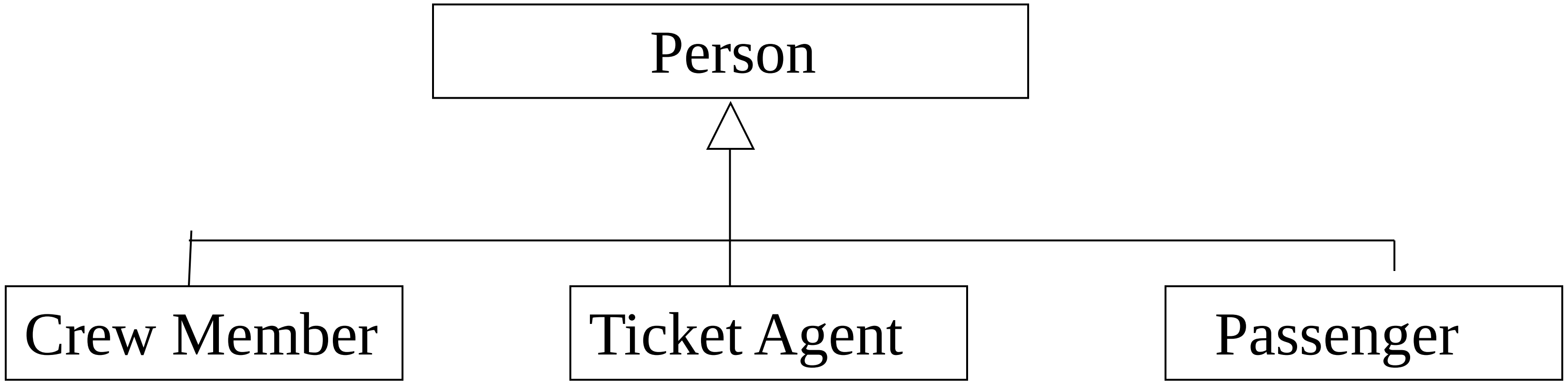


Design Pattern - Delegation

Delegation may be more suitable than **Inheritance** in some situations.

is-a-kind-of ➔ inheritance

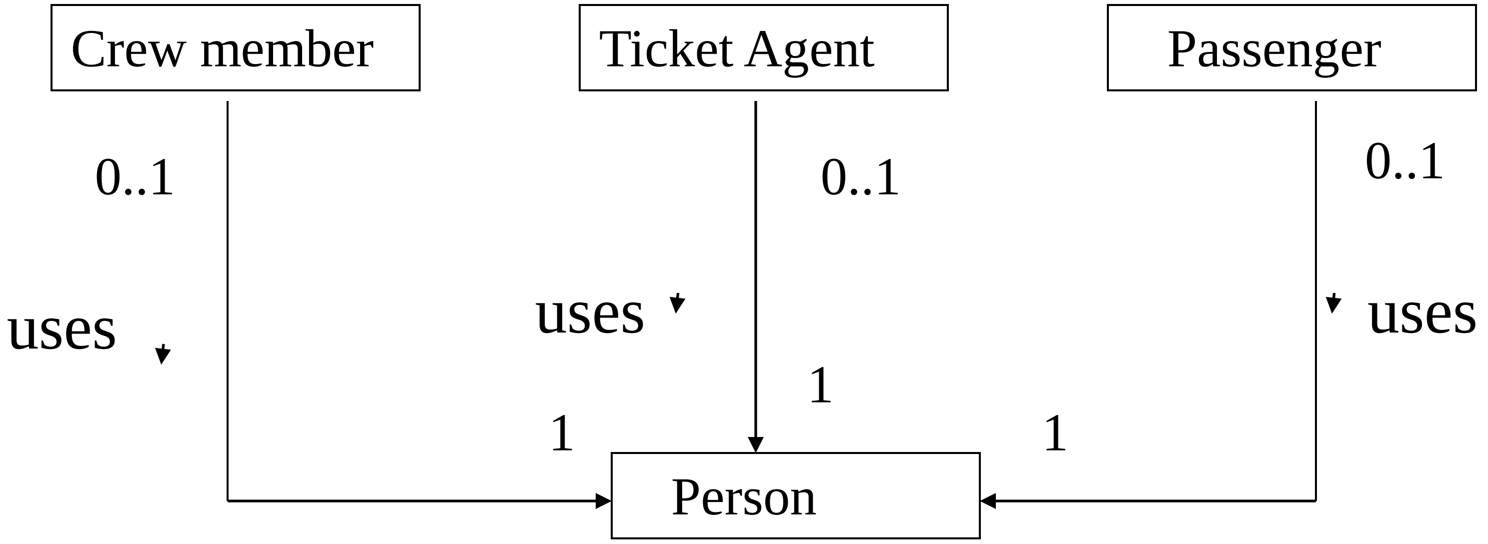
is-a-role-played-by ➔ delegation



Delegation

A sub class can play multiple roles. Eg. Crew member is a passenger as well. A more serious problem is that the same person may play different combinations of roles at different times.

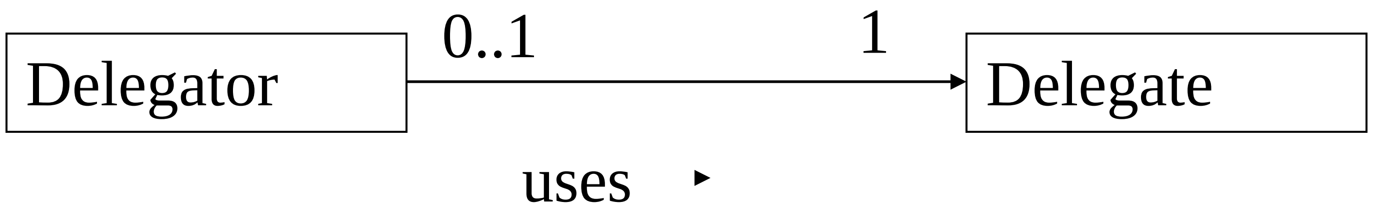
Modeling roles with Delegation

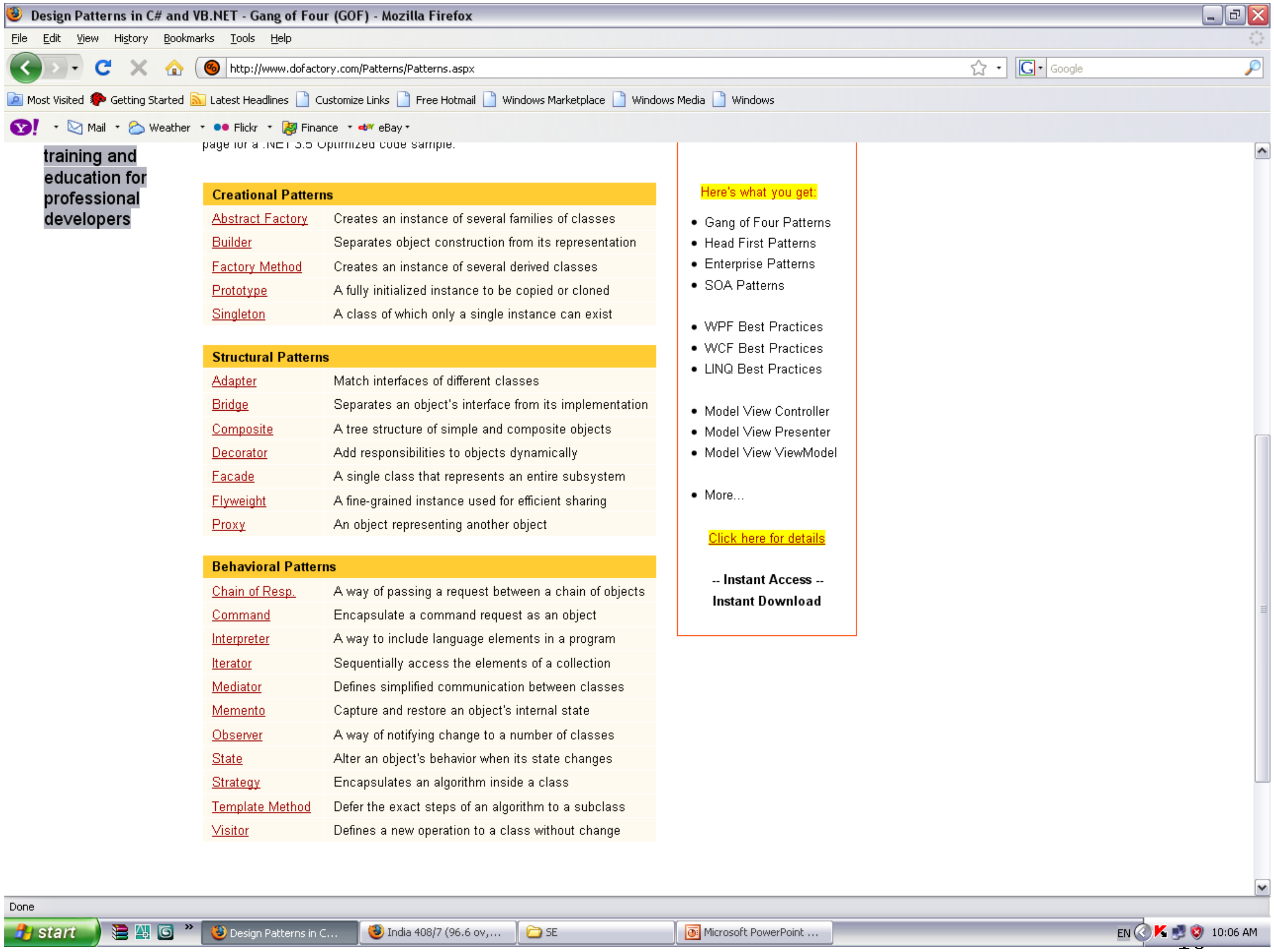


Delegation

Using the delegation model, a **Person** object delegates the responsibility of filling a particular role to an object that is specific to that role.

Since the delegation objects are dynamic, the roles objects can be added or removed as a person fills different roles.




Design Patterns in C# and VB.NET - Gang of Four (GOF) - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://www.dofactory.com/Patterns/Patterns.aspx
Google
Most Visited Getting Started Latest Headlines Customize Links Free Hotmail Windows Marketplace Windows Media Windows
Mail Weather Flickr Finance eBay
Done
start
Design Patterns in C...
India 408/7 (96.6 ov,...
SE
Microsoft PowerPoint ...
EN
10:06 AM

training and education for professional developers

page for a .NET 3.5 Optimized code sample.

Creational Patterns	
Abstract Factory	Creates an instance of several families of classes
Builder	Separates object construction from its representation
Factory Method	Creates an instance of several derived classes
Prototype	A fully initialized instance to be copied or cloned
Singleton	A class of which only a single instance can exist

Structural Patterns	
Adapter	Match interfaces of different classes
Bridge	Separates an object's interface from its implementation
Composite	A tree structure of simple and composite objects
Decorator	Add responsibilities to objects dynamically
Facade	A single class that represents an entire subsystem
Flyweight	A fine-grained instance used for efficient sharing
Proxy	An object representing another object

Behavioral Patterns	
Chain of Resp.	A way of passing a request between a chain of objects
Command	Encapsulate a command request as an object
Interpreter	A way to include language elements in a program
Iterator	Sequentially access the elements of a collection
Mediator	Defines simplified communication between classes
Memento	Capture and restore an object's internal state
Observer	A way of notifying change to a number of classes
State	Alter an object's behavior when its state changes
Strategy	Encapsulates an algorithm inside a class
Template Method	Defer the exact steps of an algorithm to a subclass
Visitor	Defines a new operation to a class without change

Here's what you get:

- Gang of Four Patterns
- Head First Patterns
- Enterprise Patterns
- SOA Patterns

- WPF Best Practices
- WCF Best Practices
- LINQ Best Practices

- Model View Controller
- Model View Presenter
- Model View ViewModel

- More...

Click here for details

-- Instant Access --
Instant Download