Gang of Four Design Patterns – Two-Liners

Accompanying material for the "Gang of Four Design Patterns in C#" course by Wincubate and Teknologisk Institut.

Creational Patterns

Abstract Factory

"Provide an interface for creating families of related or dependent objects without specifying their concrete classes."

- Create a "factory" class responsible for creating the concrete object instances needed by clients.
- Concrete factories share a common interface abstraction such that client does not rely on concrete factory classes but only on the factory abstraction.
- Ideally, the concrete instances created by the factories also share a common interface of abstract class such that the clients are completely free of dependencies on concrete instances altogether.

Builder

"Separate the construction of a complex object from its representation so that the same construction process can create different representations."

- Used when creating similar but distinct objects where either object construction process or construction data vary.
- Use a Builder class responsible for the construction data for the instances created.
- Use a Director class responsible for the construction process itself.

Factory Method

"Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses."

- Equip an abstract base class of a class hierarchy with a virtual or abstract "factory method" which can be overridden by concrete subclasses to provide details of object creation.
- This provides a family of related objects with a built-in internal factory (method) essentially playing the role of a Builder (see Builder Pattern) as each override builds a distinct object type.

Prototype

"Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype."

- Allow clients to create a new object by creating a clone of a designated master object (called the "prototype") when clients cannot otherwise create the desired object directly.
- The Prototype object exposes a method returning a clone of itself instead of exposing internal state directly in order to avoid breaking the encapsulation of the Prototype object.

Singleton

"Ensure that a class only has one instance, and provide a global point of access to it."

- Ensure that (at most) a single instance of a class is created.

- Such a Singleton class has a static property or method giving clients access to the (one and only) instance created.
- This instance is usually created by some static machinery of the Singleton class itself upon first access by a client – often in a static Singleton.Instance property getter.

Structural Patterns

Adapter

“Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn’t otherwise because of incompatible interfaces.”

- Adapt Client interface to Adaptee interface
- Adapter implements Target interface and invokes Adaptee
- Potentially: Also loosely couple or future-proof Client

Bridge

“Decouple an abstraction from its implementation so that the two can vary independently.”

- Separate abstraction and its implementation avoiding “combinatorial explosion” of classes and concepts
- Implement the abstraction by delegating to an Implementor object
- Prefer Composition over Inheritance…!

Composite

“***Compose objects into tree structures to present part/whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.”***

- Define the elements of a recursive tree-like structure
- Treats elements and groups of elements alike

Decorator

“***Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.”***

- Extend functionality without modifying existing classes
- Avoid “explosion” in number of subclasses
- Create add-on classes adding “aspect”

Façade

“***Provide a unified interface to a set of interfaces in a subsystem. Façade defines a higher-level interface that makes the subsystem easier to use.”***

- Isolate subsystems’ intricacies from client
- Make subsystem functionality reusable by clients
- Loosely couple clients and subsystems

Flyweight

“***Use sharing to support large numbers of fine-grained objects efficiently.”***

- Creating a large number of objects should be avoided

- Store *"intrinsic"* (i.e. invariant) state that can be shared
- Allow *"extrinsic"* (i.e. variant) state to be passed in methods

Proxy

"***Provide a surrogate or place-holder for another object to control access to it.***"

- Define a substitute object with the same interface
- Implement additional functionality or restriction in substitute object
- Clients cannot tell whether they interact with the real object or a proxy

Behavioural Patterns

Iterator

"Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation."

- Facilitate iteration through a read-only collection of elements of the aggregate using foreach
- Facilitate LINQ for querying elements of the aggregate
- Implement IEnumerable<T> for element type T

Chain of Responsibility

"***Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.***"

- Senders of messages are decoupled from the sequence of receivers of messages
- It is possible for more than one receiver to handle a request as it is processed in a chain-like sequence with the output of one receiver relaying the output to the next in the Chain of Responsibility.

Template Method

"***Define the skeleton of an algorithm in a method, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.***"

- Encapsulate general algorithm process in a template method
- Use template method for multiply variations of same algorithm
- Subclasses customize details of the individual steps
- Base class template method always calls subclass methods

Strategy

"***Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.***"

- Avoid unnecessary coupling
- Configure a class with one of a family of algorithms at run-time
- Strategy object implements algorithm

Memento

"***Without violating encapsulation, capture and externalize an object's internal state so that it can be restored to this state later."***

- Make object itself responsible for saving its internal state to a memento object.
- Make object itself responsible for restoring its internal state from a memento object

Command

“***Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.”***

- A Client delegates a request to some Receiver to a Command object instead of invoking Receiver directly
- Decouples the Invoker of a Command request from how it is executed.

State

“***Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.***”

- Encapsulate logic of distinct states of some State machine into separate classes
- Owner class will act a as proxy to State objects
- Makes program maintainable (and testable!) as each concrete State class is developed and tested in a loosely-coupled manner

Interpreter

“***Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.***”

- Define a grammar as a Composite IExpression class hierarchy
- Represent sentences as abstract syntax trees of IExpression objects
- Interpret sentences (recursively) by calling the Interpret() method of IExpression with a specified Context

Visitor

“***Represent a method to be performed on the elements of an object structure. Visitor lets you define a new method without changing the classes of the elements on which it operates.”***

- Define a visitor object implementing an operation on each type of elements of the object structure
- Traverse the object structure by calling accept on an element
  - Request is dispatched back to the accepted visitor’s appropriate method

Observer

“***Define a one-to-many dependency relation between objects so that when one object changes state, all its dependents are notified and updated automatically.***”

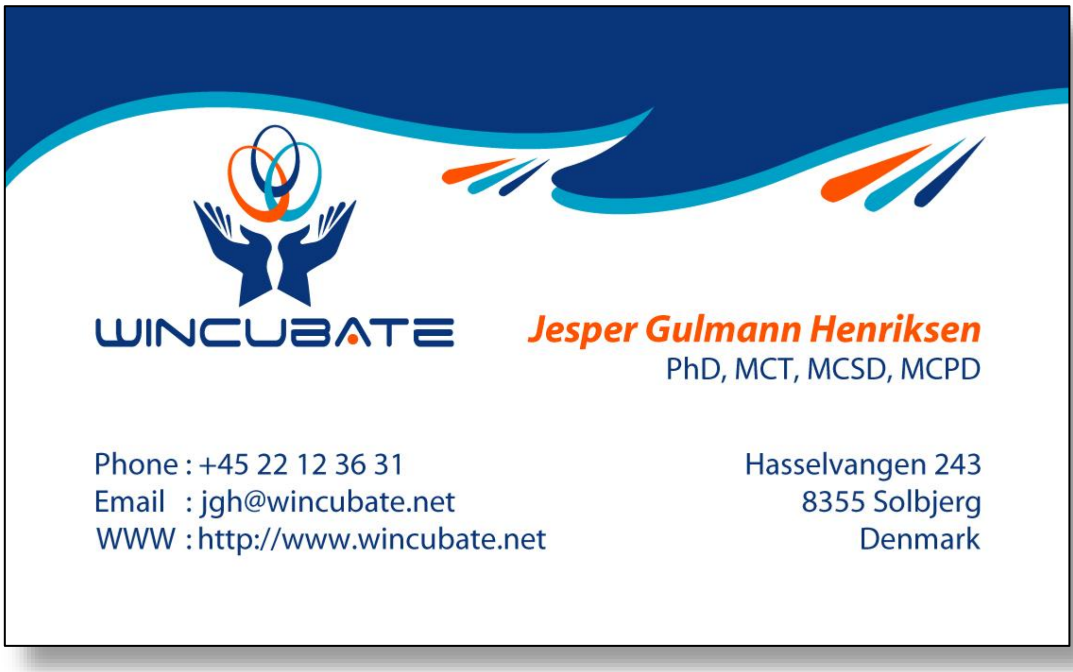
- Define Subject (emitting data) and Observer objects (consuming data)
- Let Observer objects register and deregister with Subject
- Ensure that when a Subject changes state, it will notify all registered Observers

Mediator

“***Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interactions independently.”***

- Define a separate Mediator object that encapsulates the interactions between other Colleague objects
- All Colleague objects (which may be of distinct types) interact with the Mediator instead of interacting with each other directly
- Colleagues have no explicit knowledge of other objects than the Mediator.

WINCUBATE



WINCUBATE

Jesper Gulmann Henriksen
PhD, MCT, MCSD, MCPD

Phone : +45 22 12 36 31
Email : jgh@wincubate.net
WWW : http://www.wincubate.net

Hasselvangen 243
8355 Solbjerg
Denmark