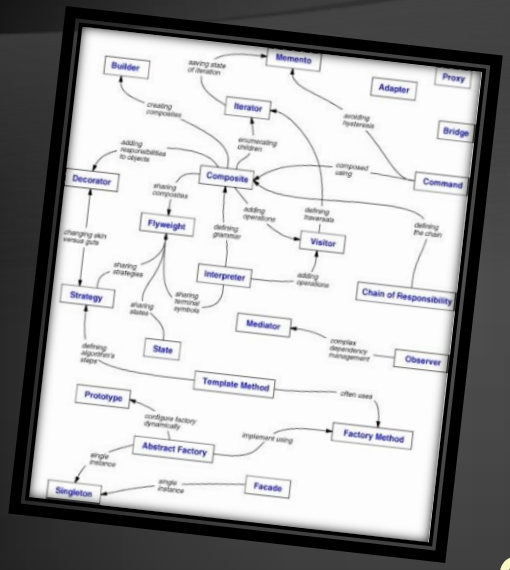
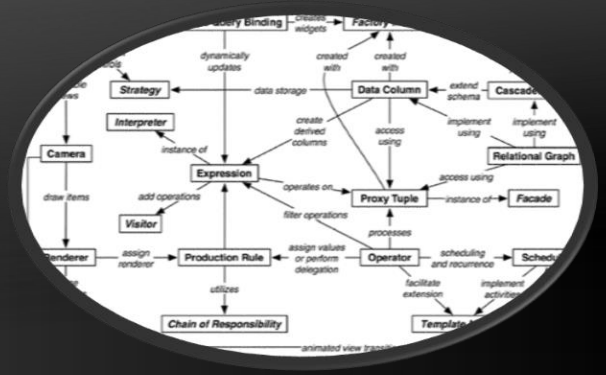
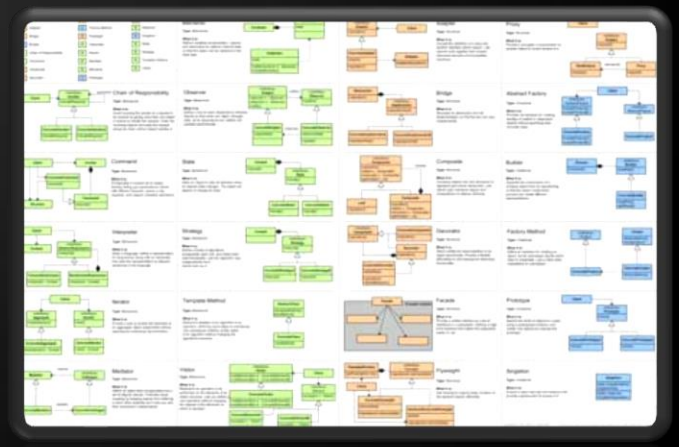
Telerik Academy





Design Patterns

General and reusable solutions to common problems in software design



Telerik Software Academy

Learning & Development

http://academy.telerik.com

Telerik Academy

Table of Contents

Telerik Academy

What is a Design Pattern?

Name, Problem, Solution and Consequences

What are Design Patterns?

- General and reusable solutions to common problems in software design
  - Problem/solution pairs within a given context
- Not a finished solution
- A template or recipe for solving certain problems
- With names to identify and talk about them

Telerik Academy

What are Design Patterns? (2)

- Patterns deal with
  - Application and system design
  - Abstractions on top of code
  - Relationships between classes or other collaborators
  - Problems that have already been solved
- Patterns are not concerned with
  - Algorithms
  - Specific implementation classes

Origins of Design Patterns

> "Each pattern describes a problem which occurs over and over again in our environment and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it in the same way twice".

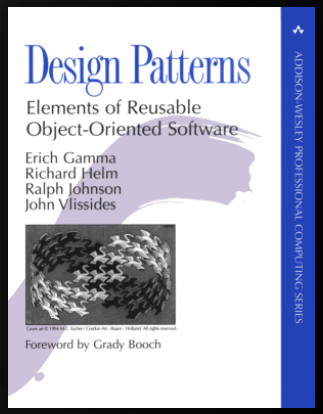
- Christopher Alexander
  - Very successful architect
  - A Pattern Language: Towns, Buildings, Construction, 1977

- Context:
  - City Planning and Building architectures

Telerik Academy

Origins of Design Patterns (2)

- Search for recurring successful designs
  - Emergent designs from practice
- Supporting higher levels of design reuse is quite challenging
- Described in Gama, Helm, Johnson, Vlissides 1995 (i.e., "Gang of Four Book")



- Based on work by Christopher Alexander
  - An Architect on building homes, buildings and towns

Describing Design Patterns

- Graphical notation is generally not sufficient
- In order to reuse design decisions the alternatives and trade-offs that led to the decisions are critical knowledge
- Concrete examples are also important
- The history of the why, when, and how set the stage for the context of usage

Pattern name
Motivation
Structure
Collaborations
Applicability
Intent
Participants
Known Uses
Implementation
Sample Code
Also Known As
Consequences
Related Patterns

Elements of Design Patterns

- Design patterns have four essential elements:
  - Pattern Name
    - Increases vocabulary of designers
  - Problem
    - Intent, context, when to apply
  - Solution
    - UML-like structure, abstract code
  - Consequences
    - Results and tradeoffs

Pattern Name

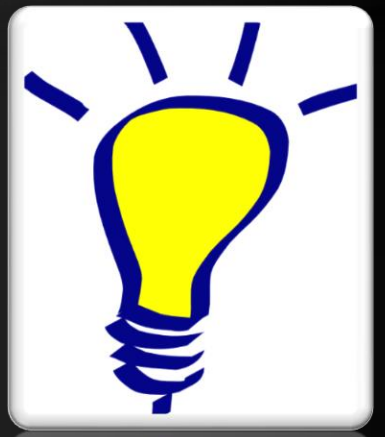
- Used to describe:
  - A design problem
  - Its solutions
  - Its consequences
- Increases design vocabulary
- Design at a higher level of abstraction
- Enhances communication
  - "The hardest part of programming is coming up with good variable names"

Problem

- Describes when to apply the pattern
- Explains the problem and its context
- May describe specific design problems and/or object structures
- May contain a list of preconditions that must be met before it makes sense to apply the pattern

Solution

- Describes the elements that make up the
  - Design
  - Relationships
  - Responsibilities
  - Collaborations



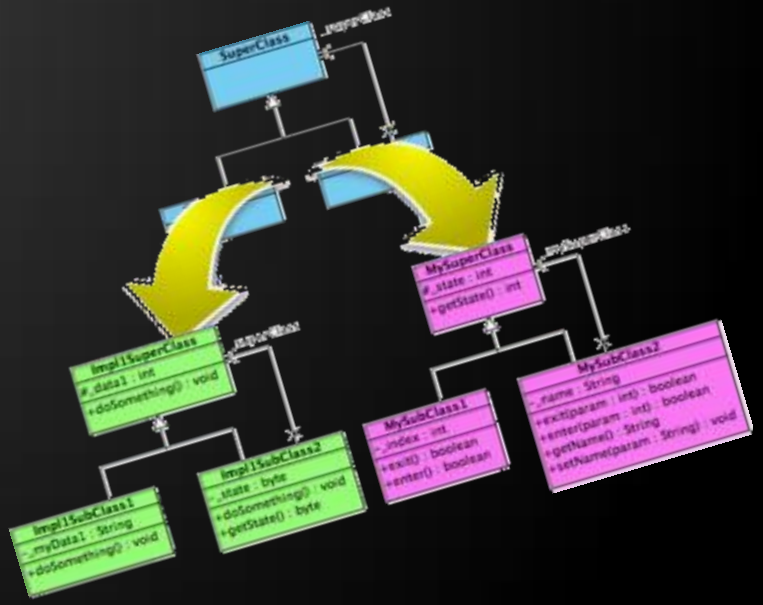
- Does not describe specific concrete implementation
  - Abstract description of design problems and how the pattern solves it

Consequences

- Results and trade-offs of applying the pattern
- Critical for:
  - Evaluating design alternatives
  - Understanding costs
  - Understanding benefits
- Includes the impacts of a pattern on a system's:
  - Flexibility, Extensibility, Portability

Telerik Academy

Why Design Patterns?



Benefits of Design Patterns

- Design patterns enable large-scale reuse of software architectures
  - Help document how systems work
- Patterns explicitly capture expert knowledge and design trade-offs
- Patterns help improve developer communication (shared language)
- Pattern names form a common vocabulary
- Patterns help ease the transition to OO technology

When to Use Patterns?

- Solutions to problems that recur with variations
  - No need for reuse if problem only arises in one context
- Solutions that require several steps:
  - Not all problems need all steps
  - Patterns can be overkill if solution is a simple linear set of instructions!
- Do not use patterns when not required
  - Overdesign is evil!

Telerik Academy

Drawbacks of Design Patterns

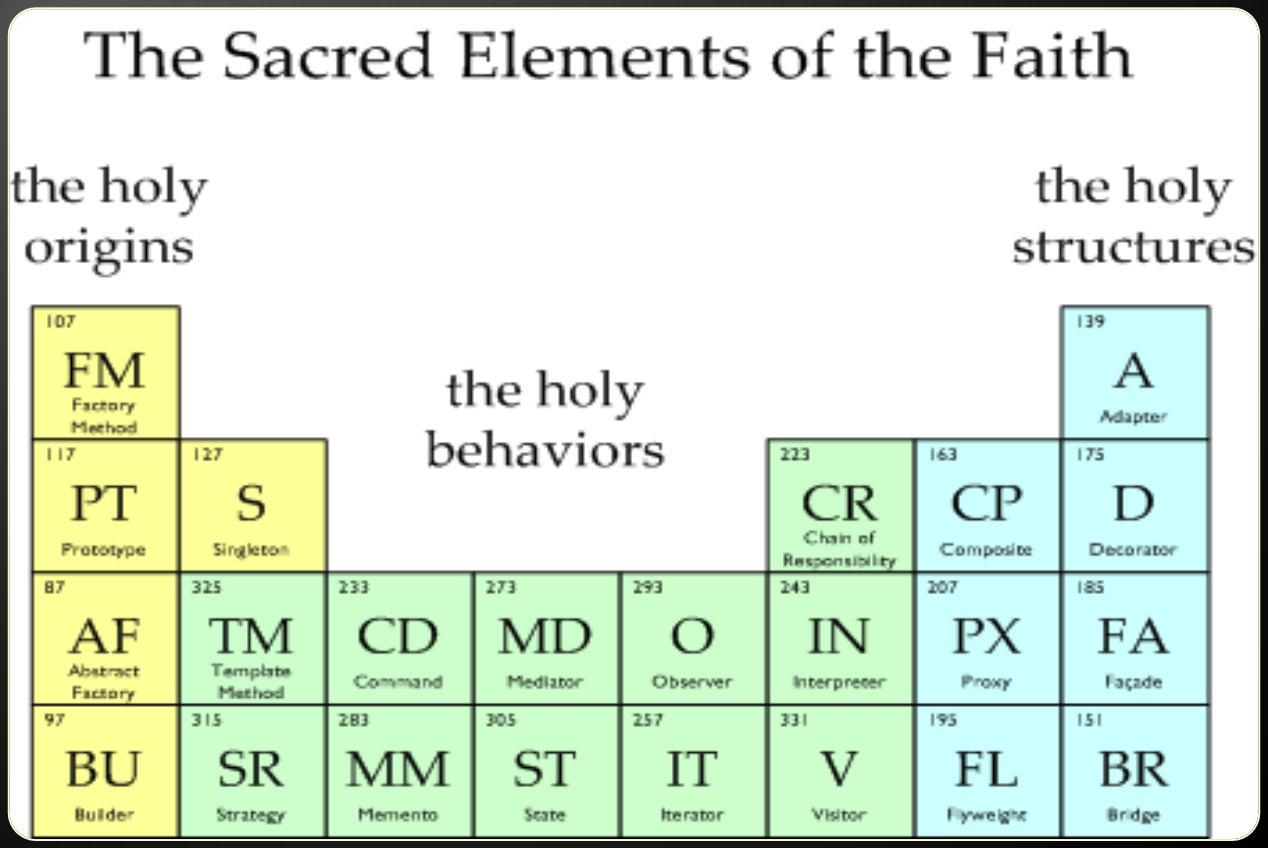
- Patterns do not lead to a direct code reuse
- Patterns are deceptively simple
- Teams may suffer from pattern overload
- Patterns are validated by experience and discussion rather than by automated testing
- Integrating patterns into the software development process is a human-intensive activity
- Use patterns if you understand them well

Criticism of Design Patterns

- Targets the wrong problem
  - The design patterns may just be a sign of some missing features of a given programming language
- Lacks formal foundations
  - The study of design patterns has been excessively ad-hoc
- Leads to inefficient solutions
- Does not differ significantly from other abstractions

Telerik Academy

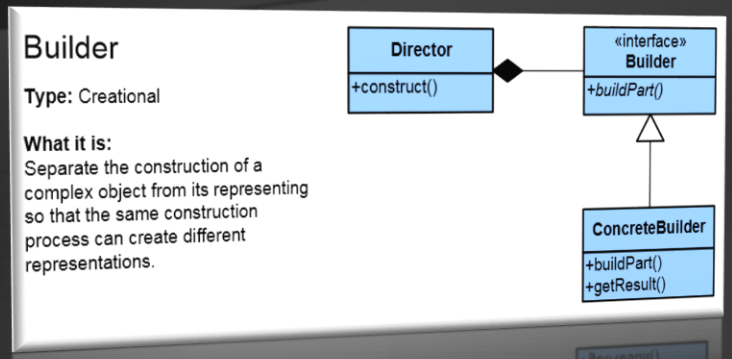
Types of Design Patterns

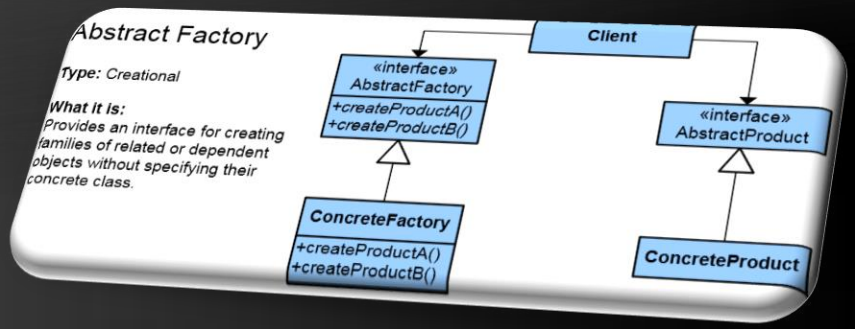


Three Main Types of Patterns

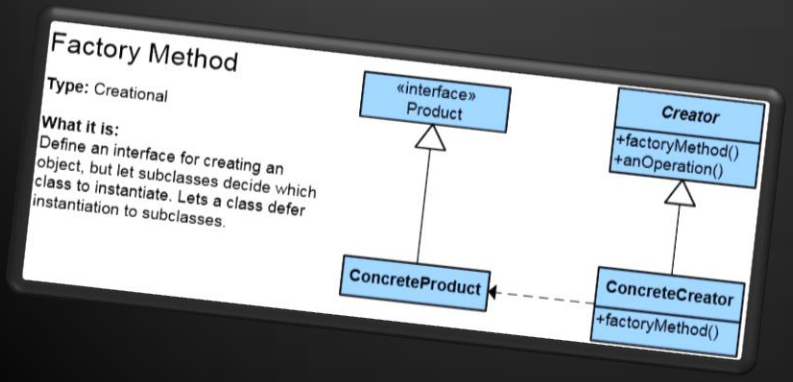
- Creational patterns
  - Deal with initializing and configuring classes and objects
- Structural patterns
  - Describe ways to assemble objects to implement a new functionality
  - Composition of classes or objects
- Behavioral patterns
  - Deal with dynamic interactions among societies of classes and objects
  - How they distribute responsibility

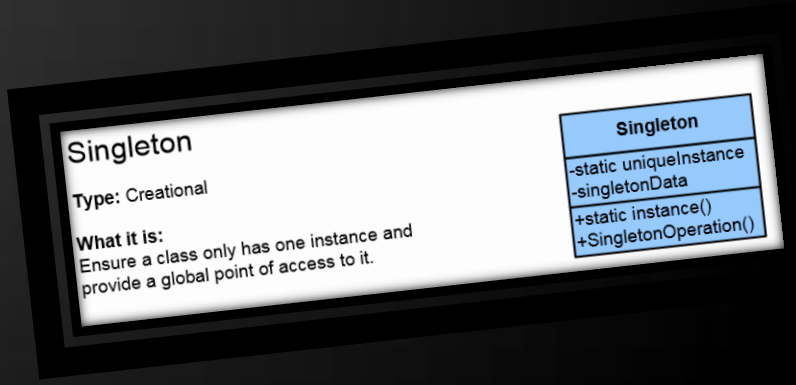
Telerik Academy





Creational Patterns



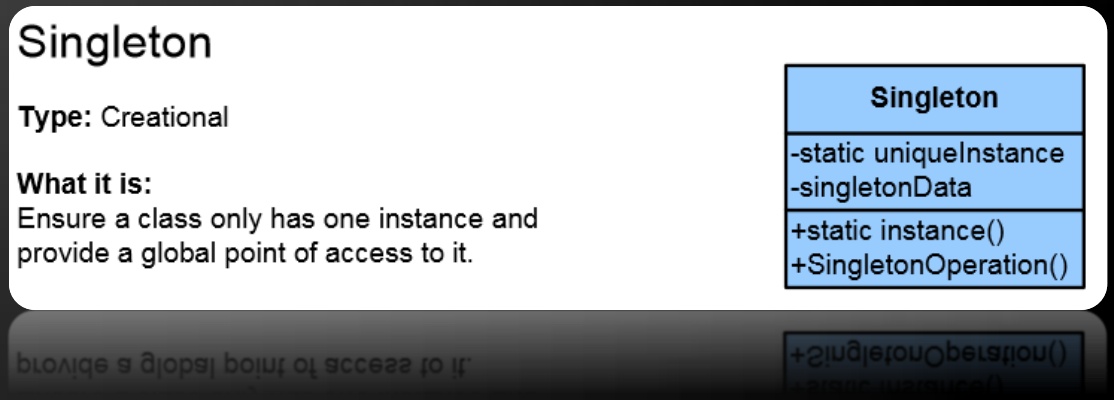


Creational Patterns

- Deal with object creation mechanisms
- Trying to create objects in a manner suitable to the situation
- Composed of two dominant ideas
  - Encapsulating knowledge about which concrete classes the system uses
  - Hiding how instances of these concrete classes are created and combined

Singleton

- The Singleton class is a class that is supposed to have only one (single) instance
- Sometimes Singleton is wrongly thought of as a global variable – it is not!
- Possible problems:
  - Lazy loading
  - Thread-safe



Simple Factory

- This is not a Pattern
  - Often mistaken with the Factory Pattern
- It is used quite often
- This is the preparation for the real Pattern
- Export the object creation in one place
  - If we making changes, we make them in one place
- We can hide complex object creation
- Higher level of abstraction

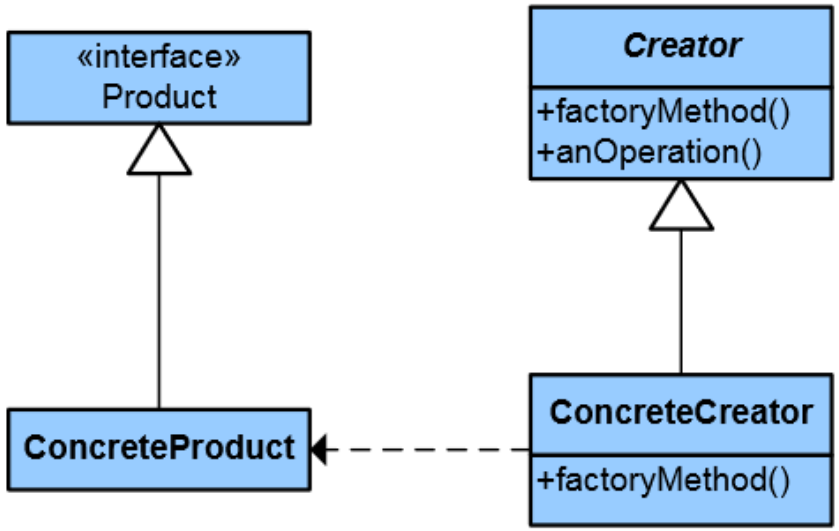
Factory Method

- Objects are created by separate method
- Produces objects as normal Factory
- This allows achieving higher reusability and flexibility in the changing applications

Factory Method

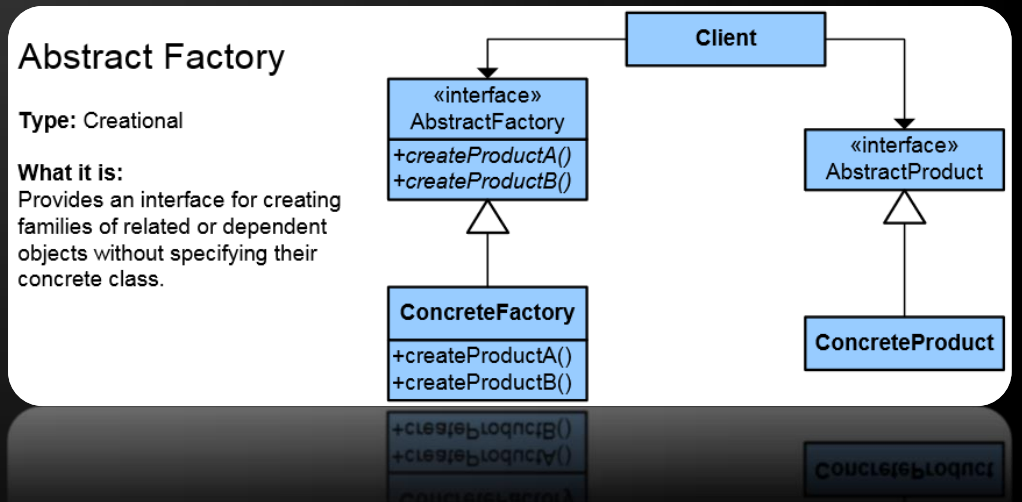
Type: Creational

What it is:
Define an interface for creating an object, but let subclasses decide which class to instantiate. Lets a class defer instantiation to subclasses.



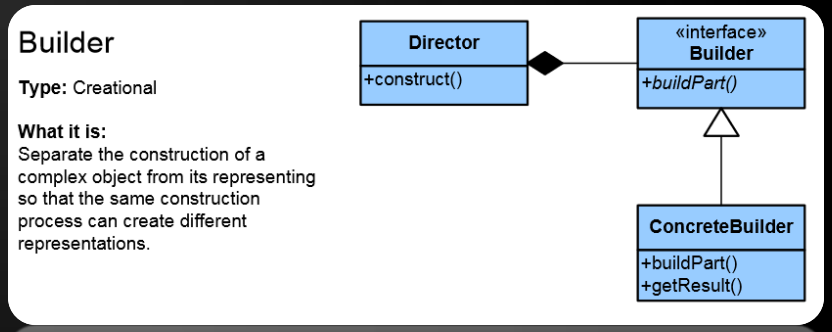
Abstract Factory

- Abstraction in object creation
  - Create a family of related objects
- The Abstract Factory Pattern defines interface for creating sets of linked objects
  - Without knowing their concrete classes
- Used in systems that are frequently changed
- Provides flexible mechanism for replacement of different sets

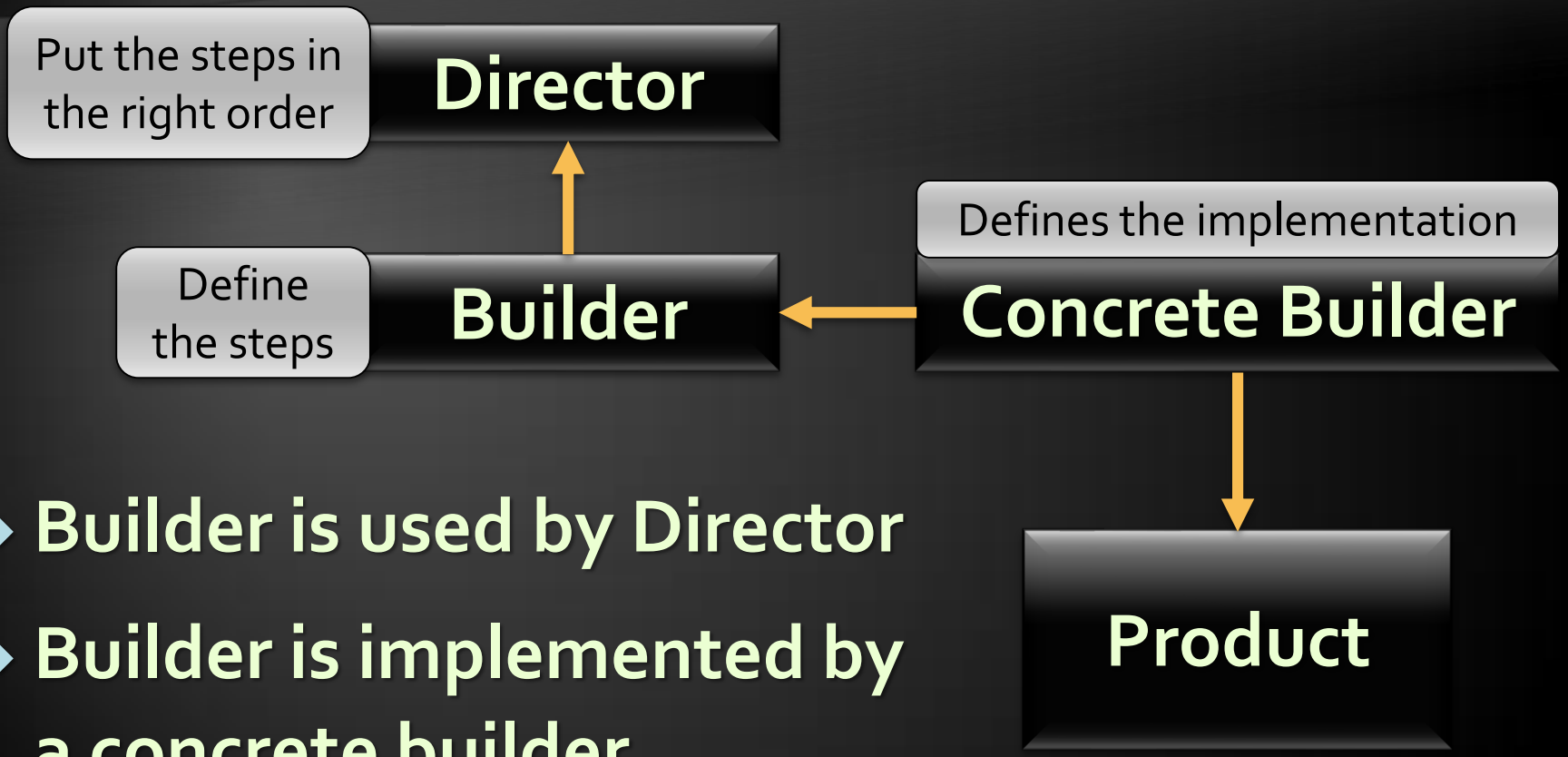


The Builder Pattern

- Separates the construction of a complex object from its representation so that the same construction process can create different representations
- Separation of logic and data
- Solves 3 types of problems
  - Too many parameters
  - Order dependent
  - Different constructions



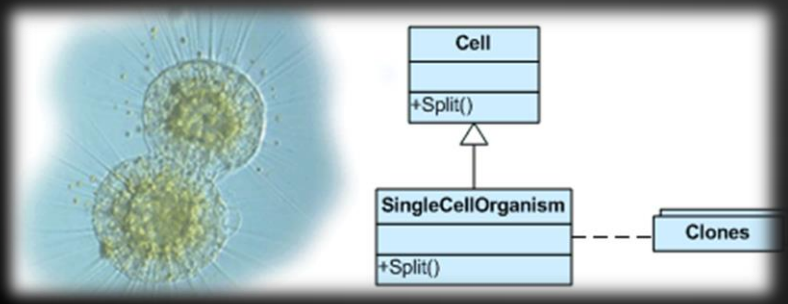
The Builder Pattern (2)

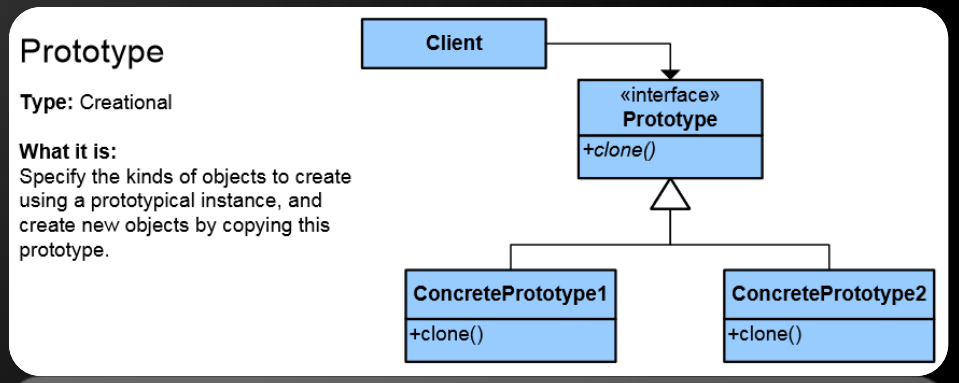


- Builder is used by Director
- Builder is implemented by a concrete builder
- Product is produced by the concrete builder

Prototype Pattern

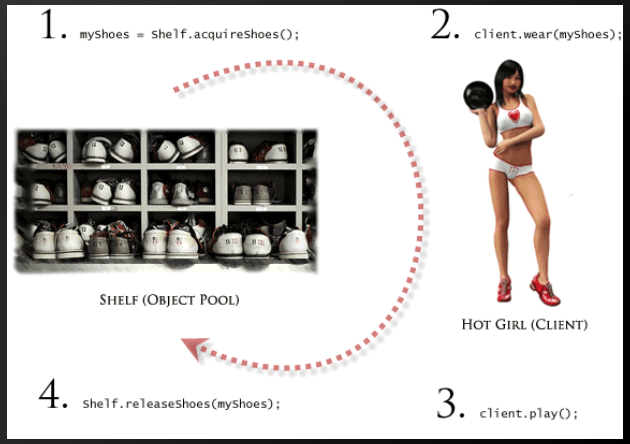
- Factory for cloning new instances from a prototype
  - Create new objects by copying this prototype
  - Instead of using "new" keyword
- ICloneable interface acts as Prototype



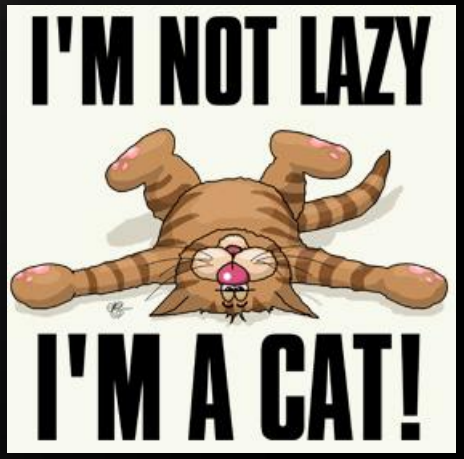


Other Creational Patterns

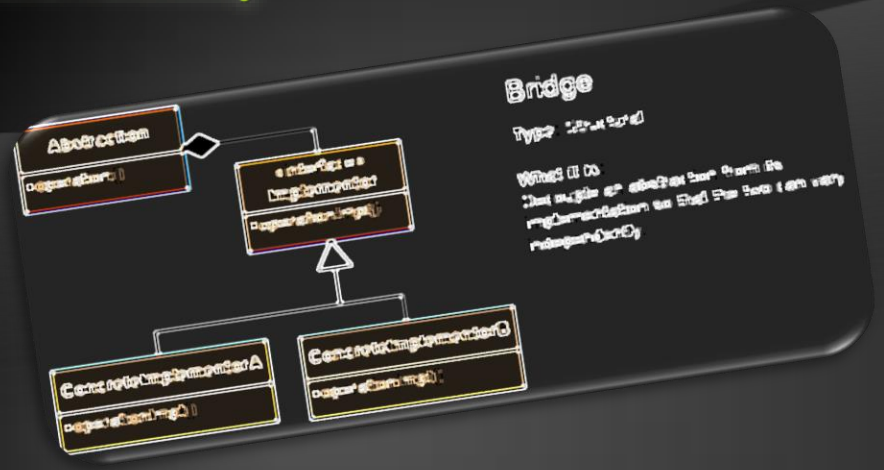
- Object Pool
  - Avoid expensive acquisition and release of resources by recycling unused objects

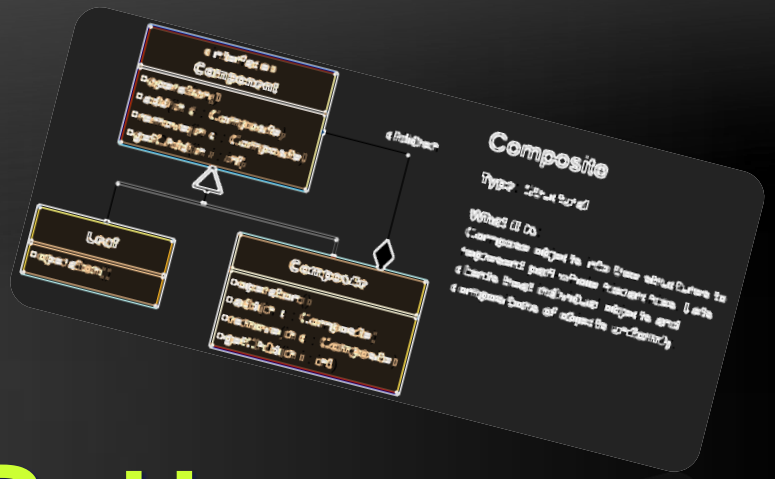


- Lazy initialization
  - Tactic of delaying the creation of an object, the calculation of a value, or some other expensive process until the first time it is needed

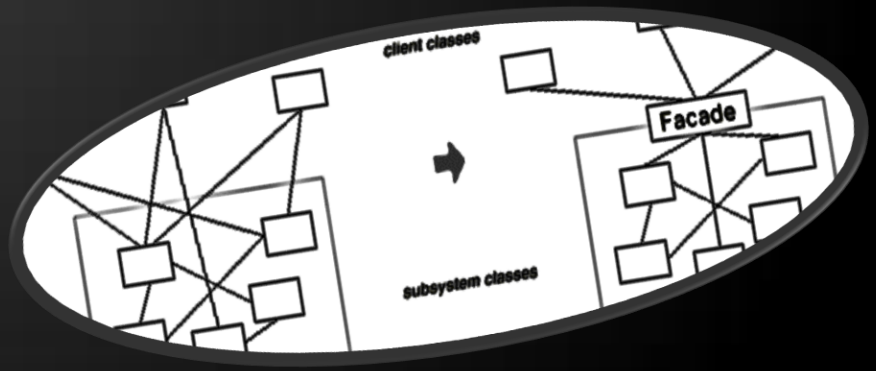


Telerik Academy





Structural Patterns

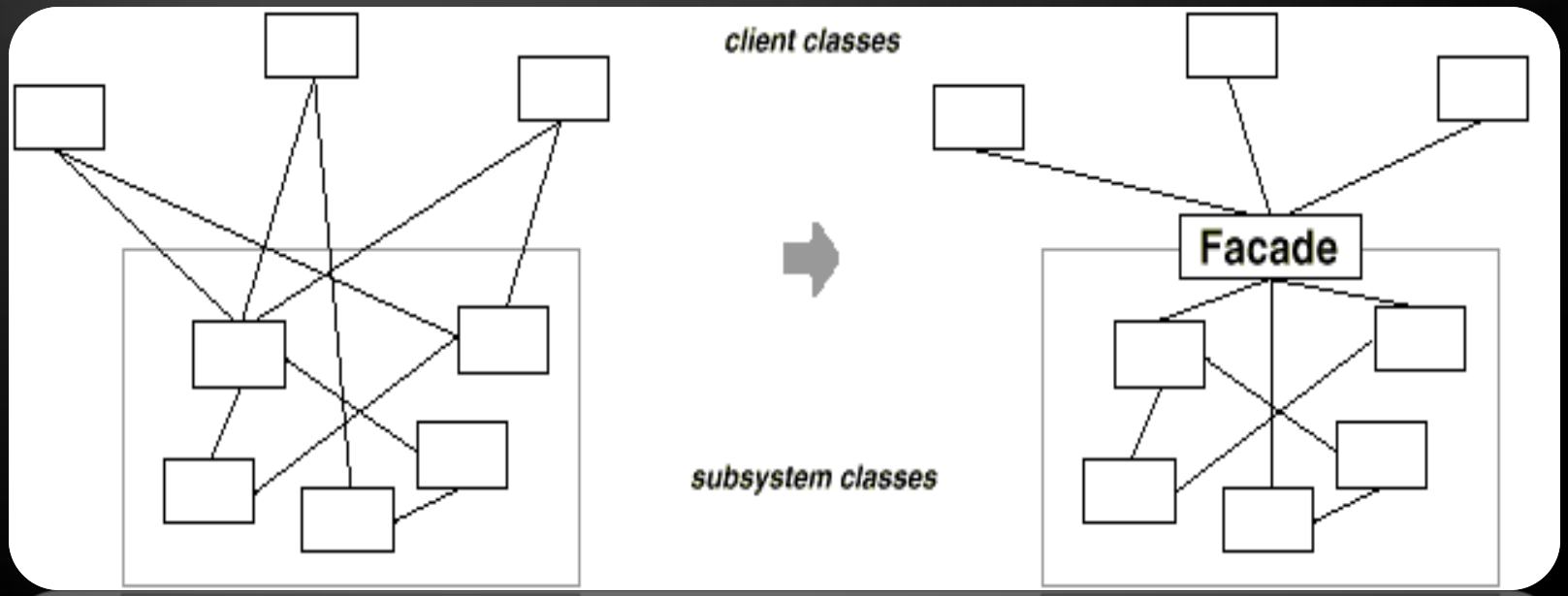


Structural Patterns

- Describe ways to assemble objects to implement a new functionality
- Ease the design by identifying a simple way to realize relationships between entities
- This design patterns is all about Class and Object composition
  - Structural class-creation patterns use inheritance to compose interfaces
  - Structural object-patterns define ways to compose objects to obtain new functionality

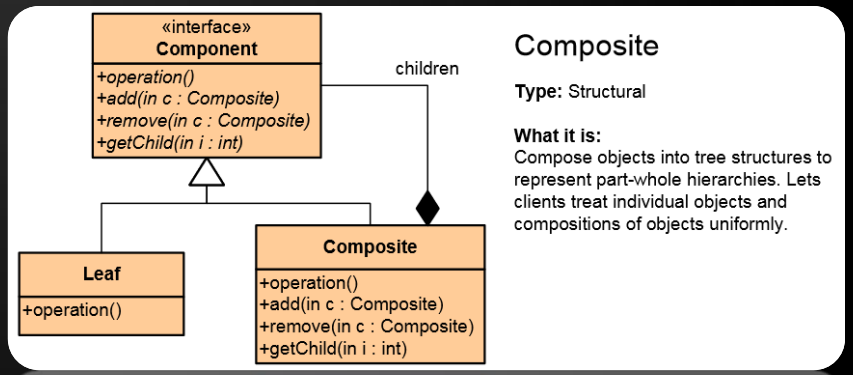
Facade Pattern

- To deliver convenient interface from higher level to group of subsystems or single complex subsystem
- Façade pattern used in many Win32 API based classes to hide Win32 complexity



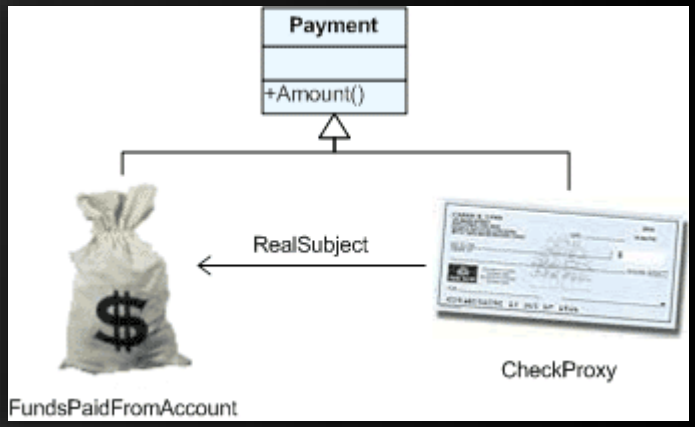
Composite Pattern

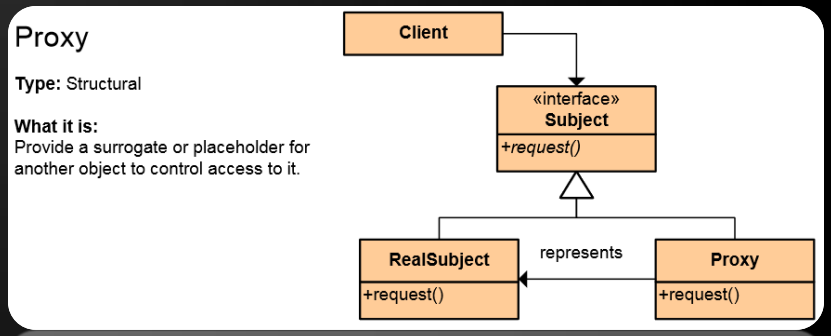
- Composite Pattern allows to combine different types of objects in tree structures
- Gives the possibility to treat the same individual objects or groups of objects
- Used when
  - You have different objects and you want to treat them the same way
  - You want to present hierarchy of objects



The Proxy Pattern

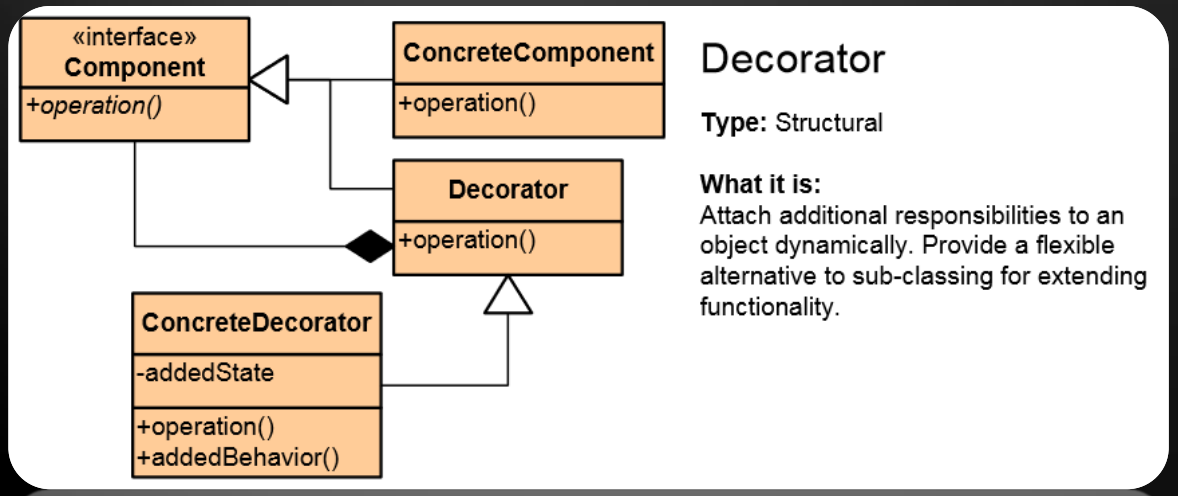
- An object representing another object
  - Provide a surrogate or placeholder for another object to control access to it
  - Use an extra level of indirection to support distributed, controlled, or intelligent access
  - Add a wrapper and delegation to protect the real component from undue complexity
- Example: Web Service

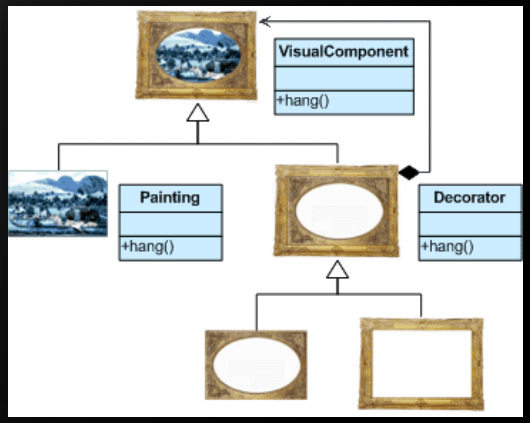




Decorator Pattern

- Add responsibilities to objects dynamically
  - Wrapping original component
  - Alternative to inheritance (class explosion)
  - Support Open-Closed principle
- In .NET: `CryptoStream` decorates `Stream`

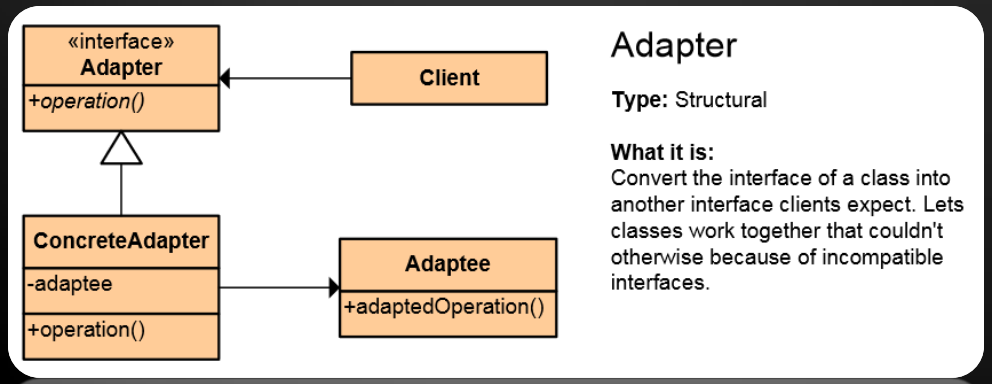


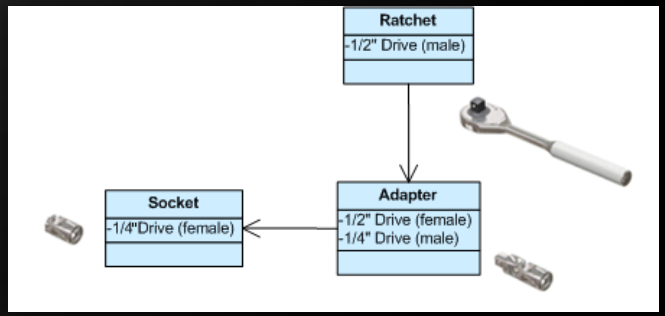


Adapter Pattern

- Converts the given class' interface into another class requested by the client
  - Wrap an existing class with a new interface
  - Impedance match an old component to a new system

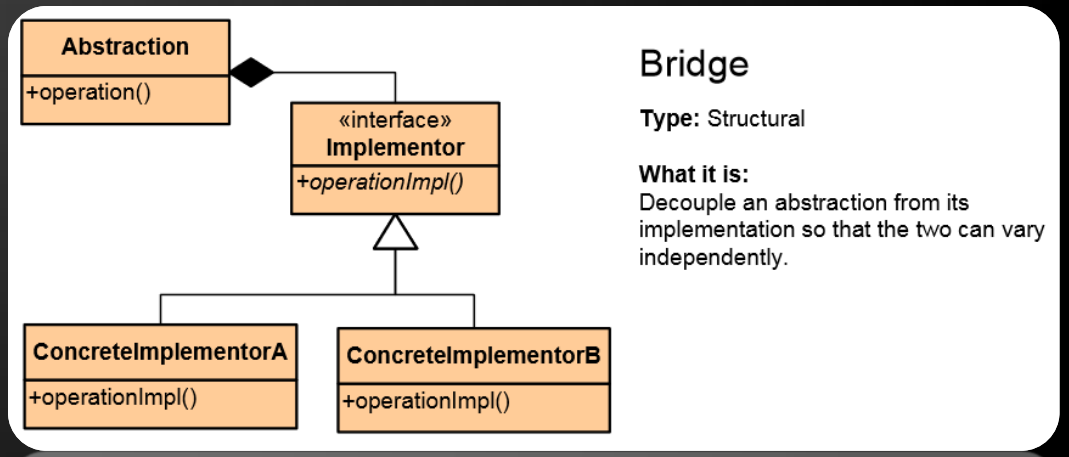
- Allows classes to work together when this is impossible due to incompatible interfaces





Bridge Pattern

- Used to divide the abstraction and its implementation (they are by default coupled)
  - That way both can be rewritten independently
- Solves problems usually solved by inheritance
- From: Abstraction -> Implementation
  To: Abstraction -> Abstraction -> Implementation

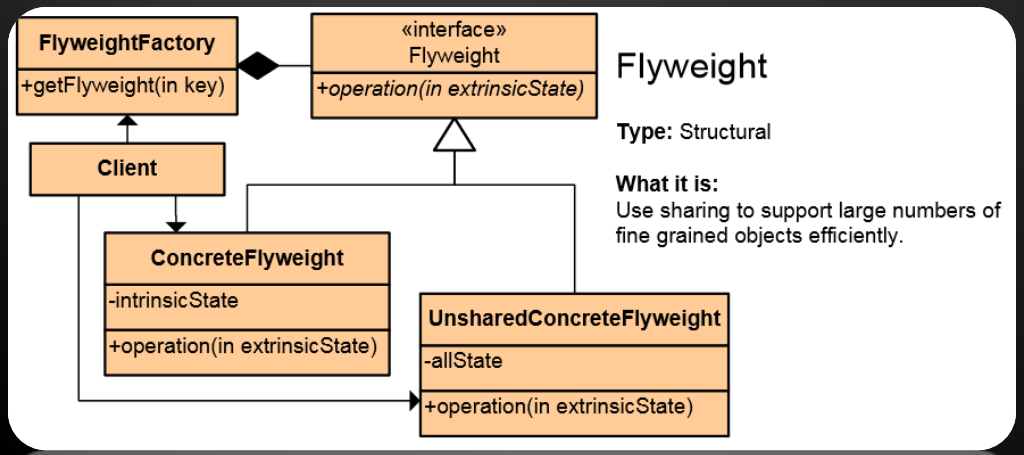


Proxy vs. Decorator vs. Adapter vs. Bridge

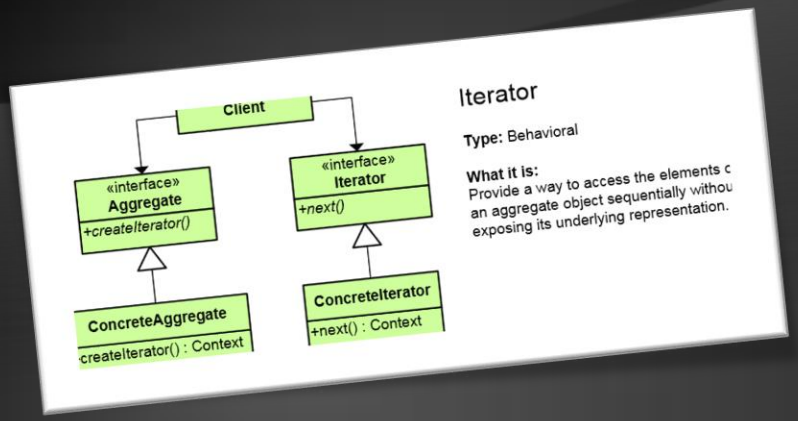
- **Proxy** – to lazy-instantiate an object, or hide the fact that you're calling a remote service, or control access to the object (one-to-one interface)
- **Decorator** – to add functionality to an object runtime (not by extending that object's type)
- **Adapter** – to map an abstract interface to another object which has similar functional role, but a different interface (changes interface for the client)
- **Bridge** – define both the abstract interface and the underlying implementation. I.e. you're not adapting to some legacy or third-party code, you're the designer of all the code but you need to be able to swap out different implementations (all changeable)

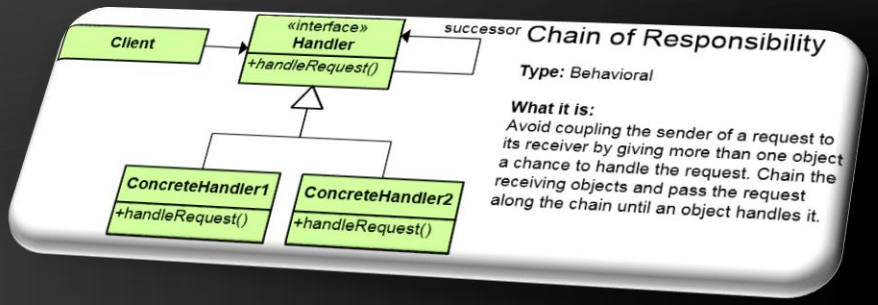
Flyweight Pattern

- Use sharing to support large numbers of fine-grained objects efficiently
  - Reduce storage costs for large number of objects
  - Share objects to be used in multiple contexts simultaneously
  - Retain object oriented granularity and flexibility

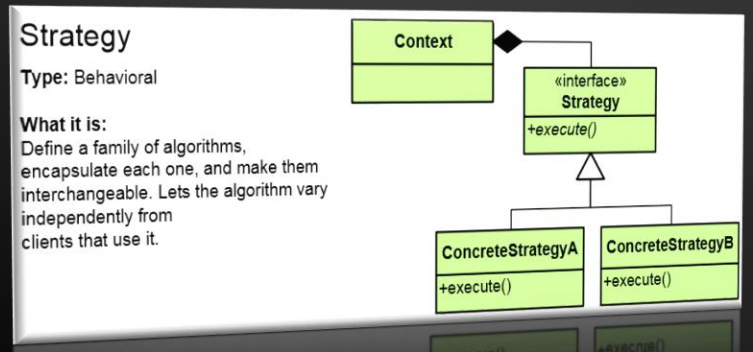


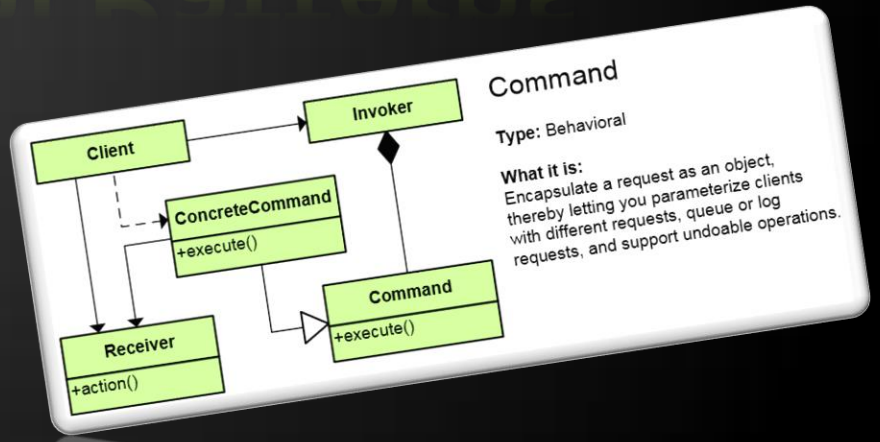
Telerik Academy





Behavioral Patterns





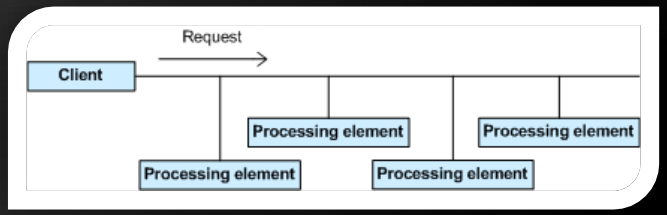
Behavioral Patterns

- Concerned with communication (interaction) between the objects
  - Either with the assignment of responsibilities between objects
  - Or encapsulating behavior in an object and delegating requests to it
- Increase flexibility in carrying out cross-classes communication

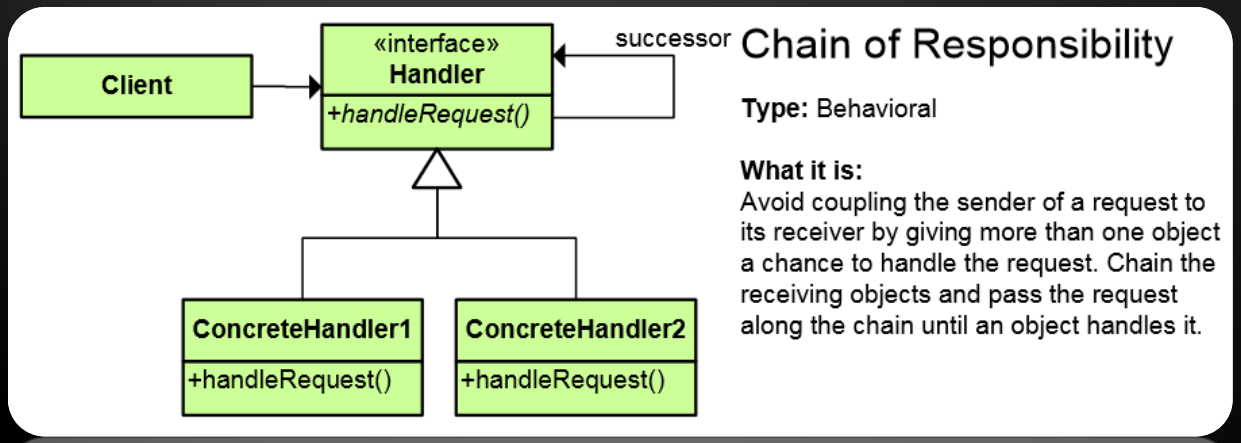
Telerik Academy

Chain of Responsibility Pattern

- Allows you to pass a request to from an object to the next until the request is fulfilled

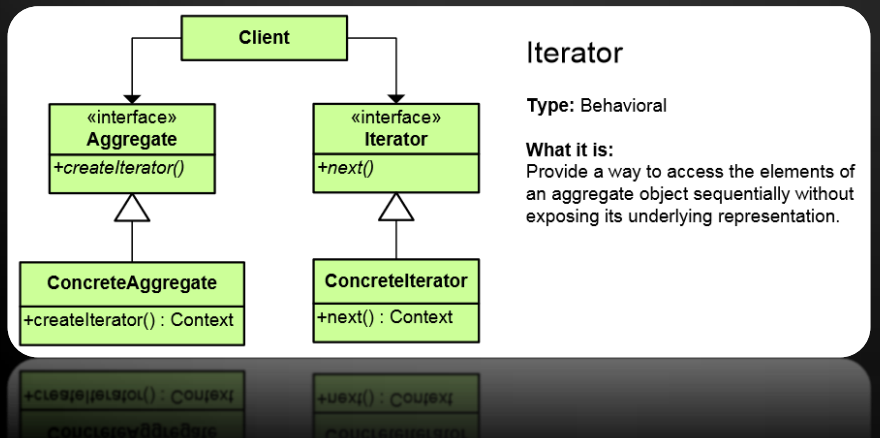


- Analogous to the exception handling
- Simplifies object interconnections
  - Each sender keeps a single reference to the next



Iterator Pattern

- Access to the elements of a complex object without revealing its actual presentation
- Various ways of data structure traversing
- Unified interface for iterating over various data structures
- foreach loops in C# uses the Iterator pattern



Iterator – Example

```
public interface IEnumerator {
    bool MoveNext();
    object Current { get; }
    void Reset();
}
```

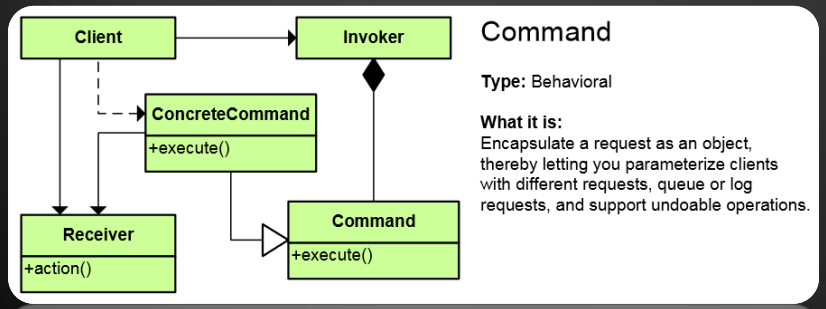
```
public interface IEnumerable {
    IEnumerator GetEnumerator();
}
```

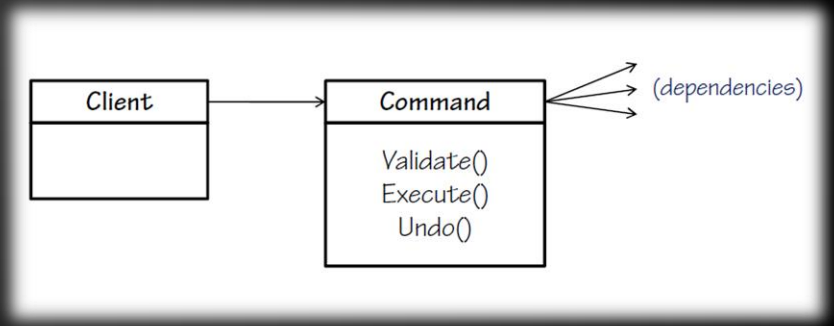
```
private class ConcreteEnumerator : IEnumerator {
    // Implement IEnumerator interface
}
```

```
var enumerator = someObject.GetEnumerator();
enumerator.Reset();
while (enumerator.MoveNext()) {
    // work with enumerator.Current
}
```

The Command Pattern

- An object encapsulates all the information needed to call a method at a later time
  - Letting you parameterize clients with different requests, queue or log requests, and support undoable operations

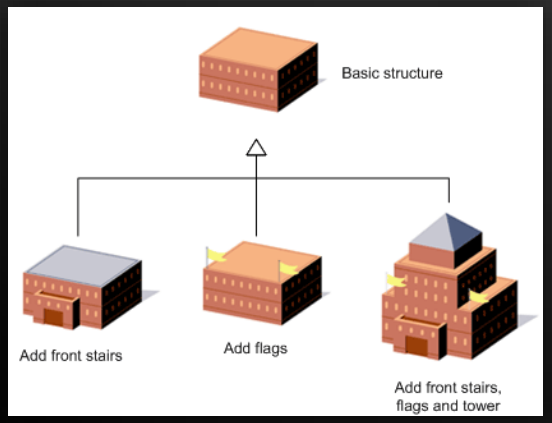




- Command in WPF and Silverlight encapsulate a request to call a method with parameters

Template Method Pattern

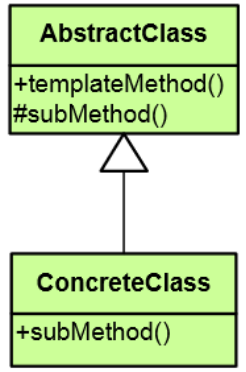
- Defines the base of an algorithm in a method, leaving some implementation to its subclasses
- Template Method allows the subclasses to redefine the implementation of some of the parts of the algorithm
  - Doesn't let the subclasses to change the algorithm structure
- Relies on inheritance; Strategy on composition



Template Method

Type: Behavioral

What it is:
Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.



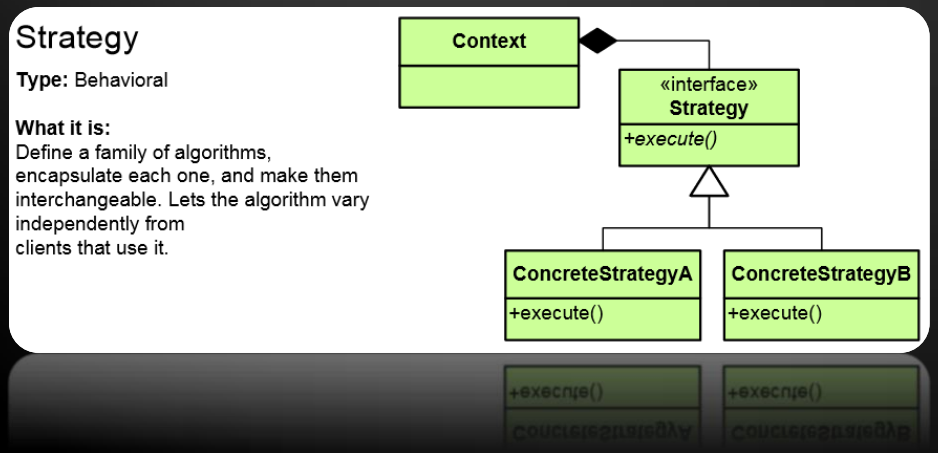
Template Method – Example

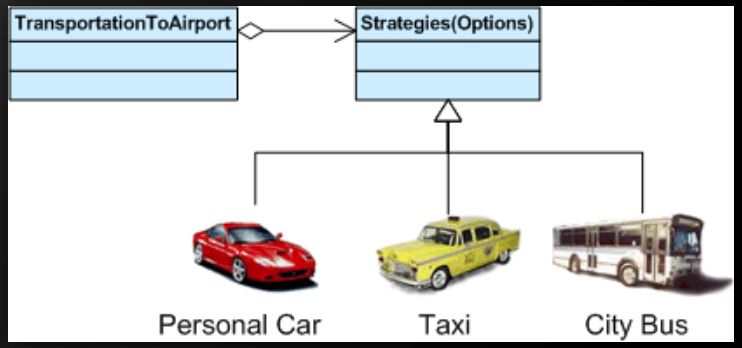
```
public abstract class HotDrink {
    public void PrepareRecipe()
    {
        BoilWater(); Brew(); PourInCup(); AddSpices();
    }
    protected abstract void Brew();
    protected abstract void AddSpices();
    private void BoilWater() { ... }
    private void PourInCup() { ... }
}
public class Coffee : HotDrink {
    protected override void Brew() { ... }
    protected override void AddSpices() { ... }
}
public class Tea : HotDrink {
    protected override void Brew() { ... }
    protected override void AddSpices() { ... }
}
```

Implemented by subclasses

Strategy Pattern

- Encapsulates an algorithm inside a class
  - Making each algorithm replaceable by others
    - All the algorithms can work with the same data transparently
    - The client can transparently work with each algorithm





Strategy Pattern – Example

```
abstract class SortStrategy {
    public abstract void Sort(IList<object> list);
}
```

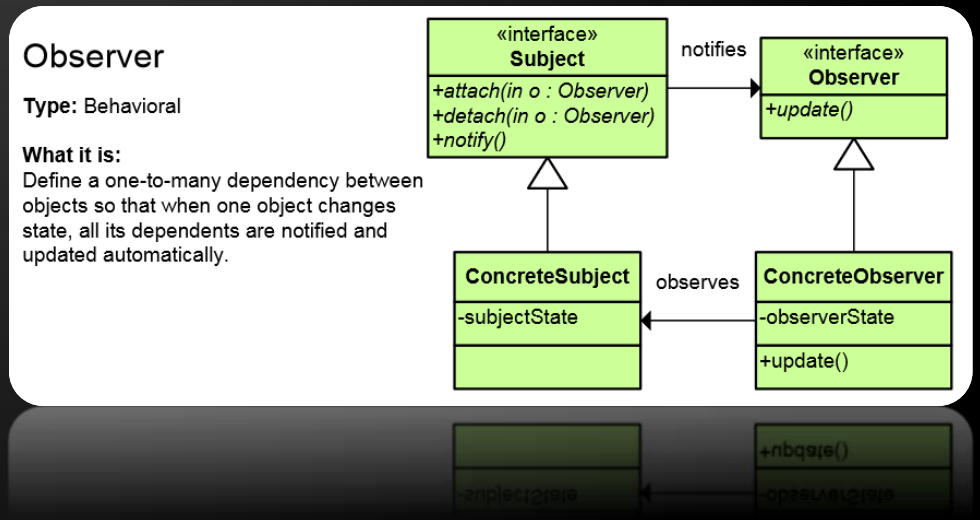
```
class QuickSort : SortStrategy {
    public override void Sort(IList<object> list) { ... }
}
```

```
class MergeSort : SortStrategy {
    public override void Sort(IList<object> list) { ... }
}
```

```
class SortedList {
    private IList<object> list = new List<object>();
    public void Sort(SortStrategy strategy) {
        // sortStrategy can be passed in constructor
        sortStrategy.Sort(list);
    }
}
```

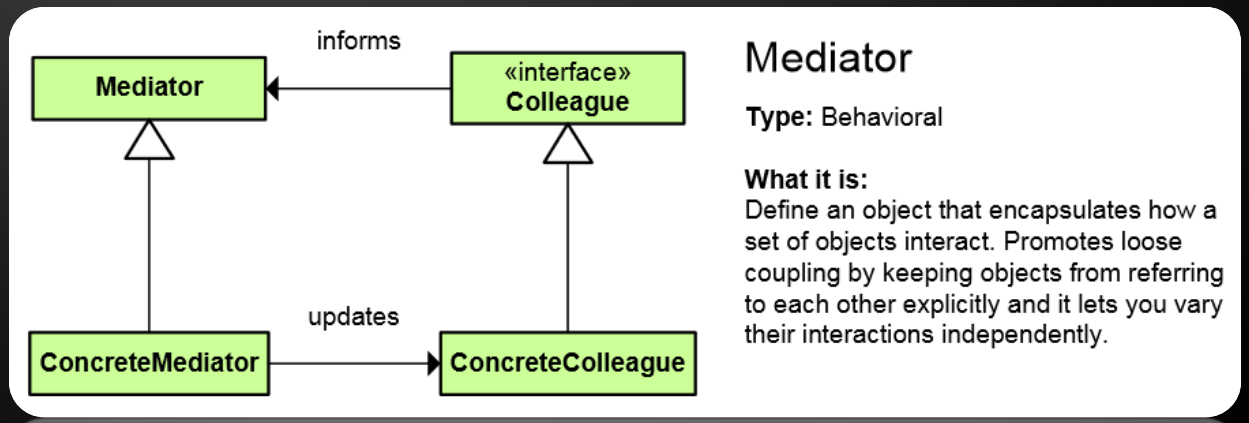
Observer Pattern

- Presents interface, allowing object to communicate without any concrete knowledge about each other
- Also known as Publish-Subscribe pattern
- Object to inform other object about its state, without the knowledge which are these objects
- In .NET Framework events and event handlers use this pattern



Mediator Pattern

- Simplifies communication between classes
- Define an object that encapsulates how a set of objects interact
- Promotes loose coupling by keeping objects from referring to each other explicitly
  - Lets you vary their interaction independently



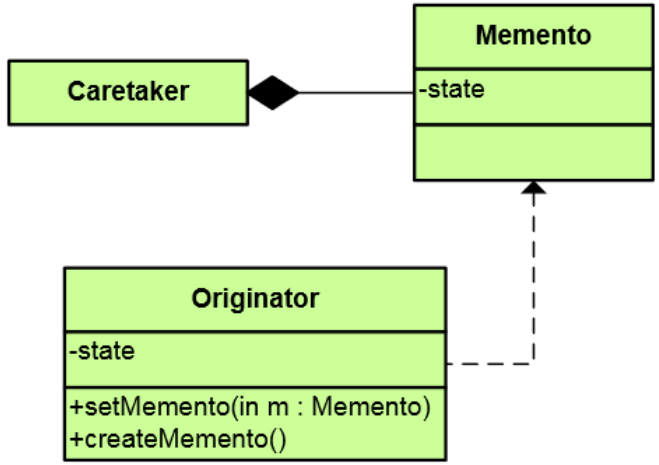
Memento Pattern

- Capture and restore an object's internal state
- Promote undo or rollback to full object status
- A magic cookie that encapsulates a "check point" capability

Memento

Type: Behavioral

What it is:
Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later.



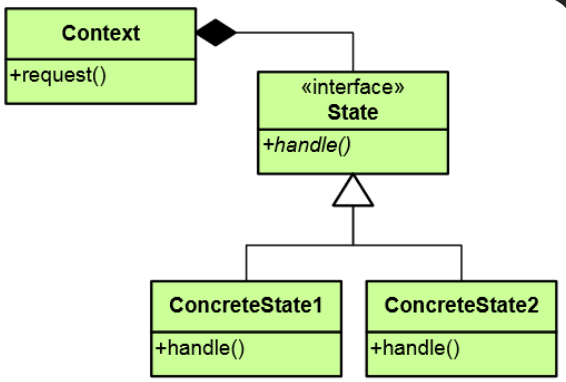
State Pattern

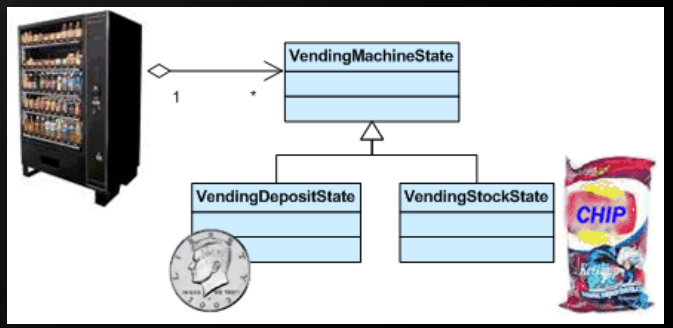
- Alter an object's behavior when its state changes
  - Change behavior of the object with each state
  - Encapsulate the logic of each state into an object
  - Allow dynamic state discovery
  - Make unit testing easier
- An object-oriented state machine

State

Type: Behavioral

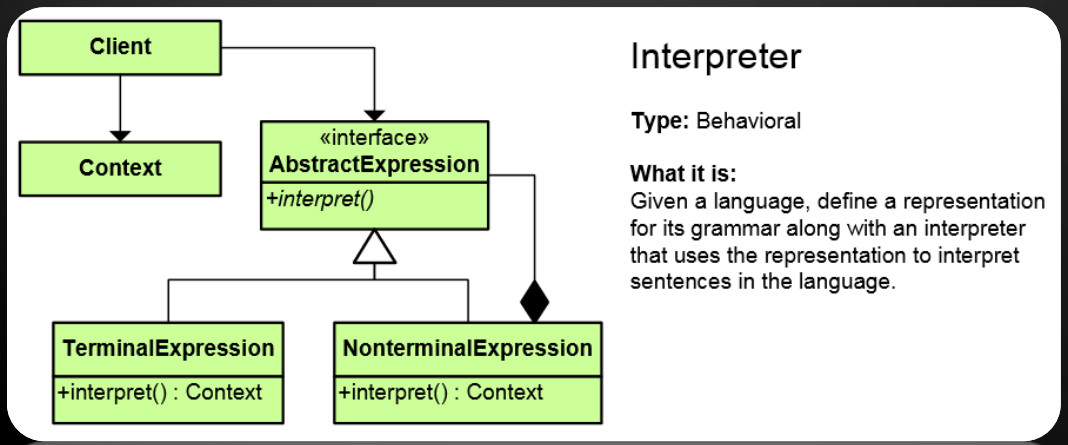
What it is:
Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.

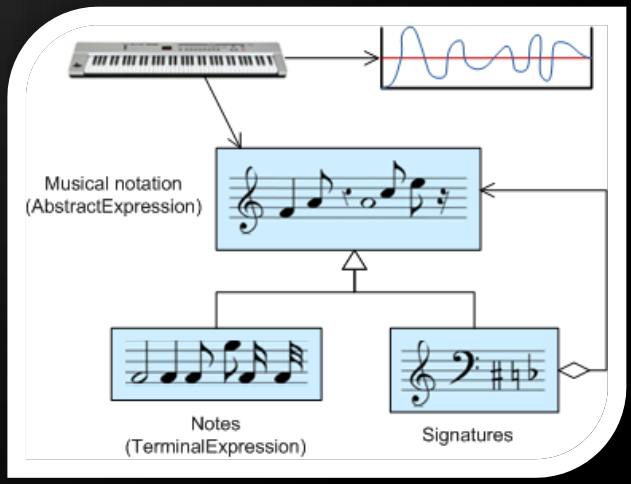




Interpreter Pattern

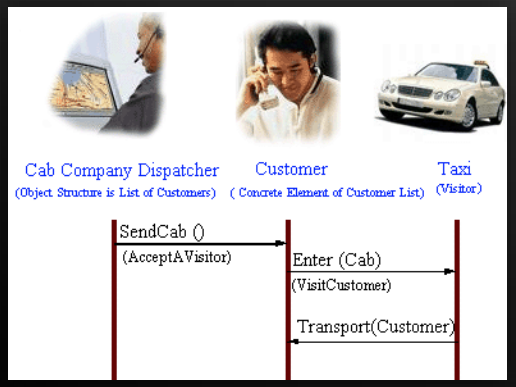
- A way to include language (formal grammar) elements in a program
  - Define a representation for the grammar
  - Define an interpreter that uses the representation to interpret sentences in the language

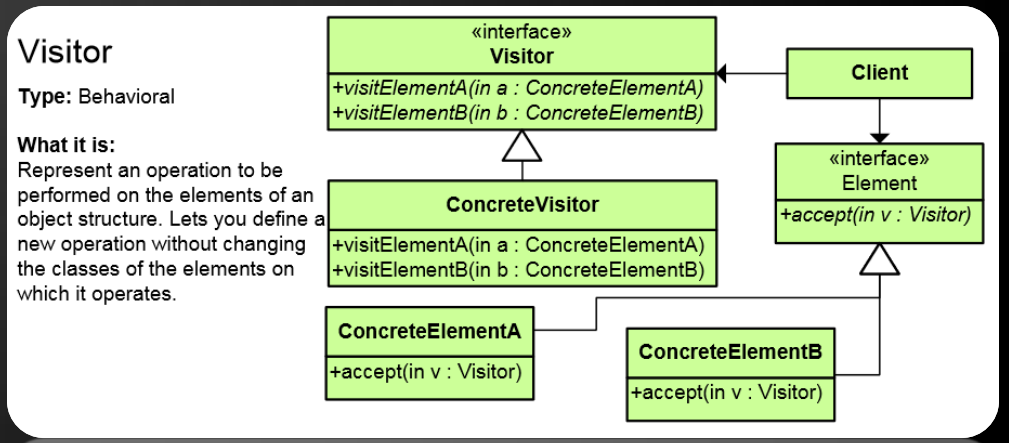




Visitor Pattern

- Defines a new operation to a class without change the elements of the class
  - The classic technique for recovering lost type information
  - Do the right thing based on the type of two objects
  - Double dispatch

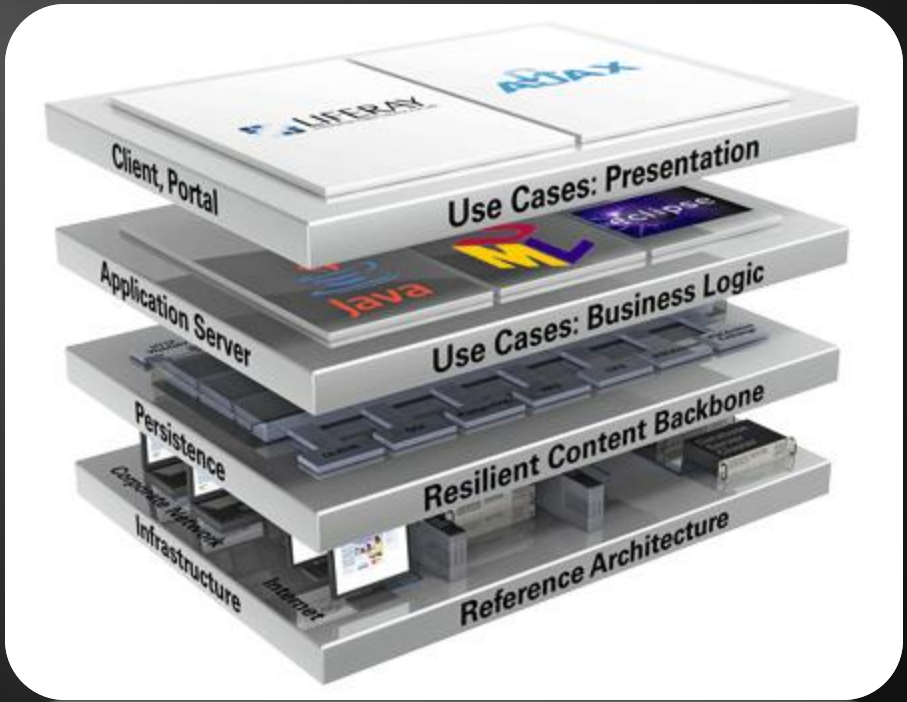




Other Behavioral Patterns

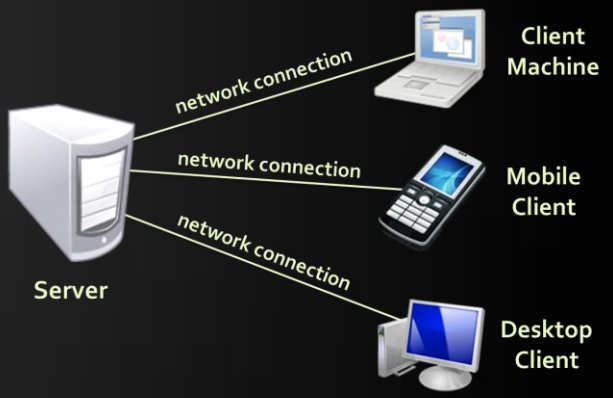
- Null Object
  - Designed to act as a default value of an object
  - In .NET: String.Empty, EventArgs.Empty, etc.
- Hierarchical visitor (Composite + Visitor)
  - Visit every node in a hierarchical data structure
- Protocol stack (Upper Layer / Lower Layer)
- Scheduled-task
- Single-serving visitor (Use and then delete)
- Specification pattern (Combine rules (and/or))

Architectural patterns



Client-Server Architecture

- The client-server model consists of:
  - Server – a single machine / application that provides services to multiple clients
    - Could be IIS based Web server
    - Could be WCF based service
    - Could be a services in the cloud
  - Clients –software applications that provide UI (front-end) to access the services at the server
    - Could be WPF, HTML5, Silverlight, ASP.NET, …

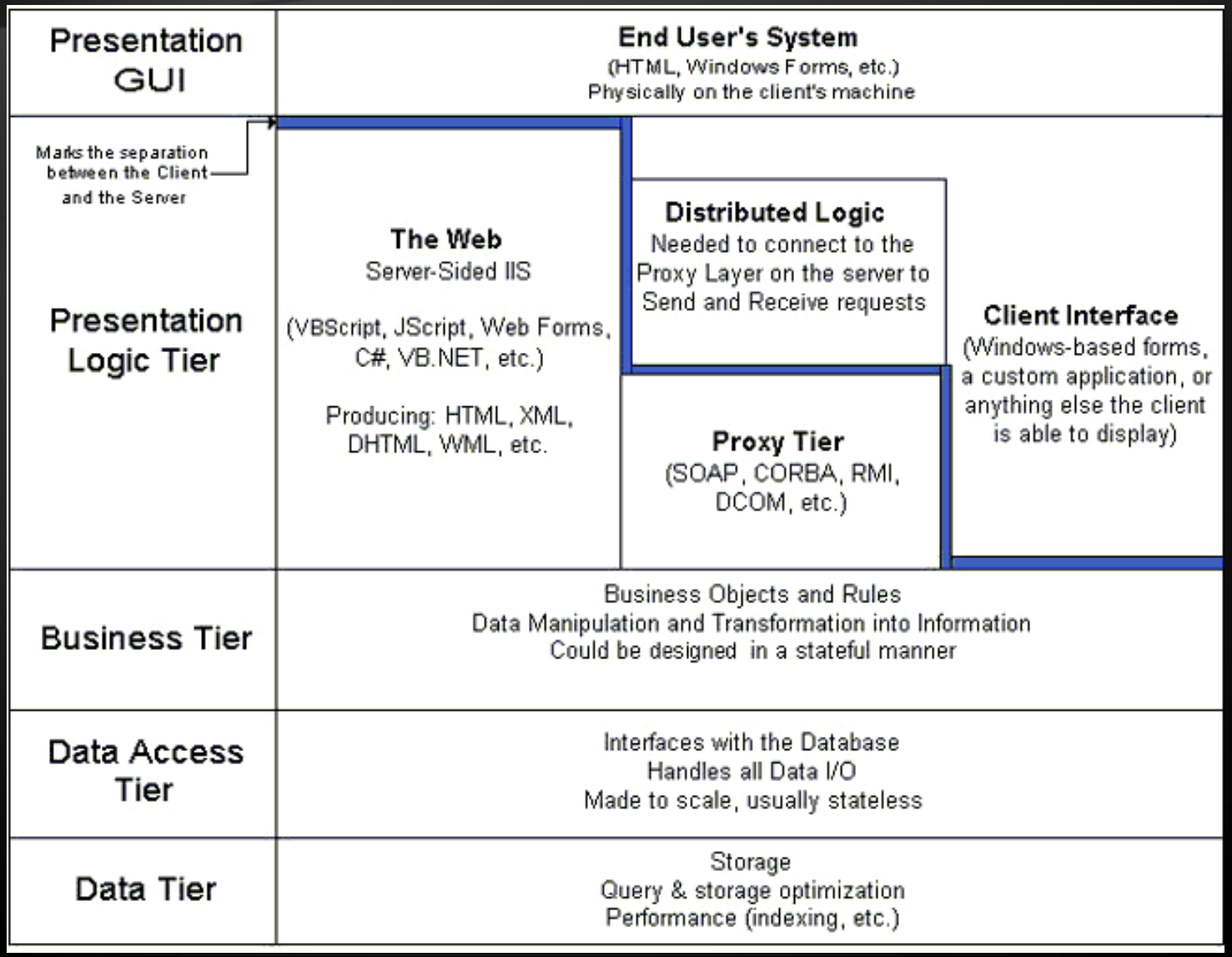


The 3-Tier Architecture

- The 3-tier architecture consists of the following tiers (layers):
  - Front-end (client layer)
    - Client software – provides the UI of the system
  - Middle tier (business layer)
    - Server software – provides the core system logic
    - Implements the business processes / services
  - Back-end (data layer)
    - Manages the data of the system (database / cloud)

Telerik Academy

Multi-Tier Architecture



Telerik Academy

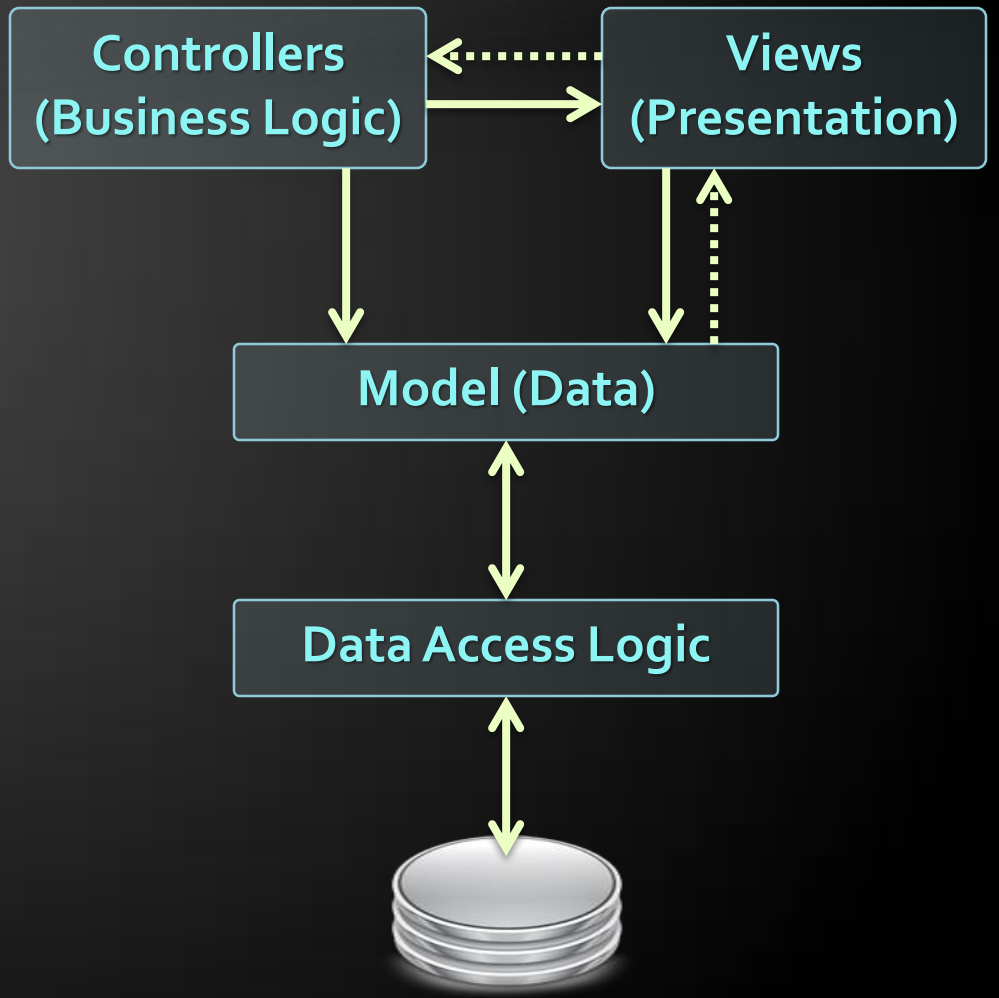
Model-View-Controller (MVC)

- Model-View-Controller (MVC) architecture
  - Separates the business logic from application data and presentation
- Model
  - Keeps the application state (data)
- View
  - Displays the data to the user (shows UI)
- Controller
  - Handles the interaction with the user

Telerik Academy

MVC and Multi-Tier Architecture

- MVC does not replace the multi-tier architecture
  - Both are usually used together
- Typical multi-tier architecture can use MVC
  - To separate logic, data and presentation



Telerik Academy

Model-View-Presenter (MVP)

- Model-View-Presenter (MVP) is UI design pattern similar to MVC
  - Model
    - Keeps application data (state)
  - View
    - Presentation – displays the UI and handles UI events (keyboard, mouse, etc.)
  - Presenter
    - Presentation logic (prepares data taken from the model to be displayed in certain format)

Telerik Academy

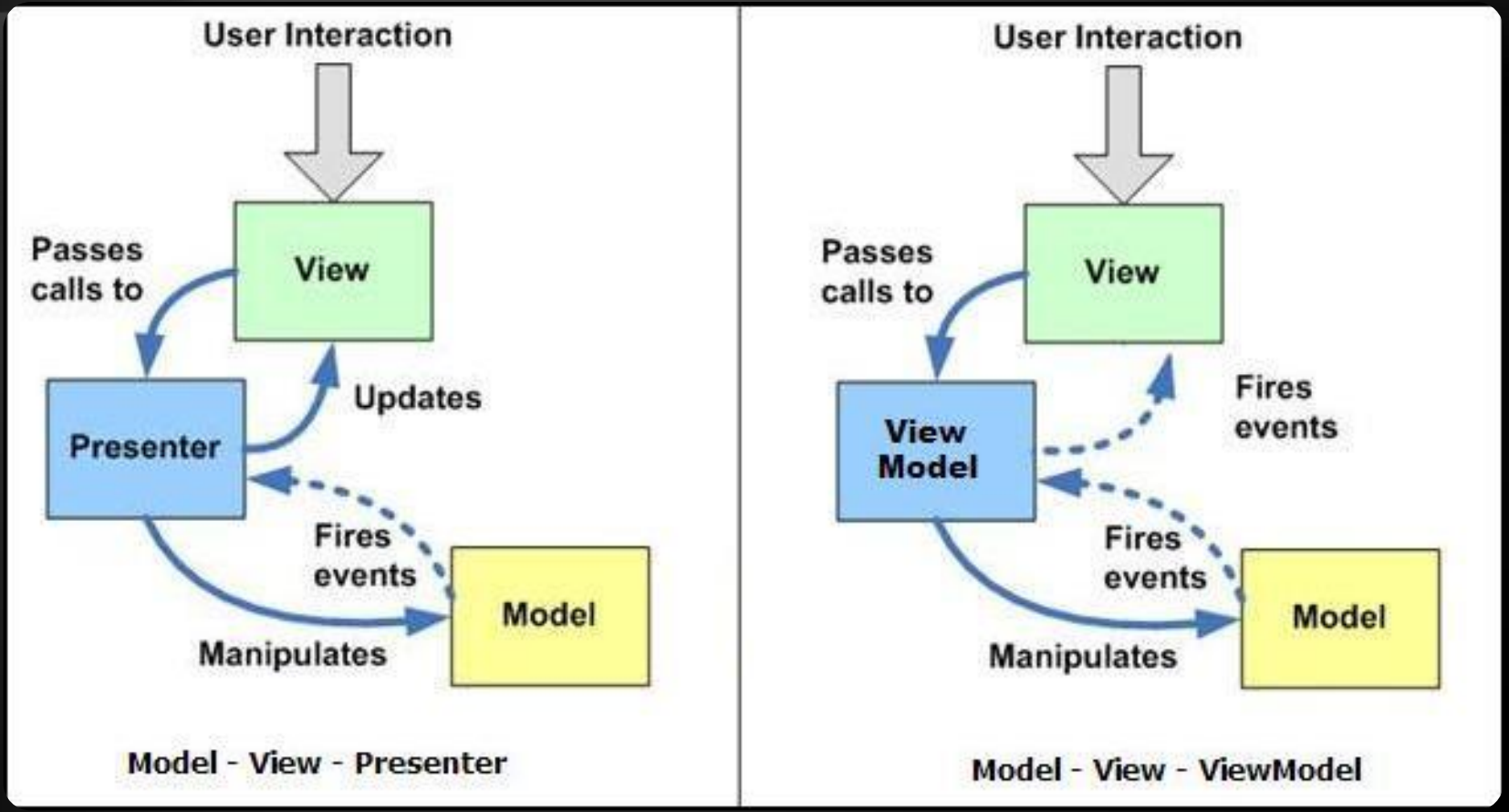
Model-View-ViewModel (MVVM)

- Model-View-ViewModel (MVVM) is architectural pattern for modern UI development
  - Invented by Microsoft for use in WPF and Silverlight
  - Based on MVC, MVP and Martin Fowler's Presentation Model pattern
  - Officially published in the Prism project (Composite Application Guidance for WPF and Silverlight)
  - Separates the "view layer" (state and behavior) from the rest of the application

MVVM Structure

- Model
  - Keeps the application data / state representation
  - E.g. data access layer or ORM framework
- View
  - UI elements of the application
  - Windows, forms, controls, fields, buttons, etc.
- ViewModel
  - Data binder and converter that changes the Model information into View information
  - Exposes commands for binding in the Views

MVP vs. MVVM Patterns



- MVVM is like MVP but leverages the platform's build-in bi-directional data binding mechanisms

SOA

- Service-Oriented Architecture (SOA) is a concept for development of software systems
  - Using reusable building blocks (components) called "services"
- Services in SOA are:
  - Autonomous, stateless business functions
  - Accept requests and return responses
  - Use well-defined, standard interface

Other Design Patterns

- Concurrency patterns
  - Active Object
  - Double Checked Locking pattern
  - Monitor Object
    - An object to can be safely used by many threads
  - Read-Write Lock pattern
  - Thread Pool pattern
    - A number of threads are created to perform a number of tasks
  - And many more

Telerik Academy

More Software Patterns

- Idioms (low level, C++)
  - E.g. when should you define a virtual destructor?
- Design (micro-architectures) [Gamma-GoF]
- Architectural patterns (systems design)
  - Client-Server, 3-Tier / Multi-Tier
  - MVC (Model-View-Controller)
  - MVP (Model-View-Presenter)
  - MVVM (Model View ViewModel)
  - SOA (Service-Oriented Architecture) Patterns

Telerik Academy

Patterns in .NET Framework

- Iterator pattern in `foreach` loops in C#
- Observer pattern – events and event handlers
- Adapter pattern is used in ADO.NET
- Decorator: `CryptoStream` decorates `Stream`
- Command in WPF and Silverlight encapsulate a request to call a method with parameters
- Façade pattern used in many Win32 API based classes to hide Win32 complexity
- Chain of Responsibility is similar to exceptions
- String.Empty is a Null Object

Telerik Academy

Design Patterns

http://academy.telerik.com

Homework

- Select 3 design patterns
  - Write a short description (about half page) for each of them (prefer Swedish language)
    - Describe their motivation, intent, applicability, known uses, implementation, consequences, structure, related patterns, etc.
  - Provide C# examples for their use
  - Provide a UML diagram or image of the pattern
    - You can download it from the Internet