Introduction to Design Patterns

Memi Lavi
www.memilavi.com

Design Patterns:

A collection of general, reusable solutions to common problems* in software design

* Examples:

- How to communicate between classes
- How to initialize interface implementations
- How to access data stores
- And more…

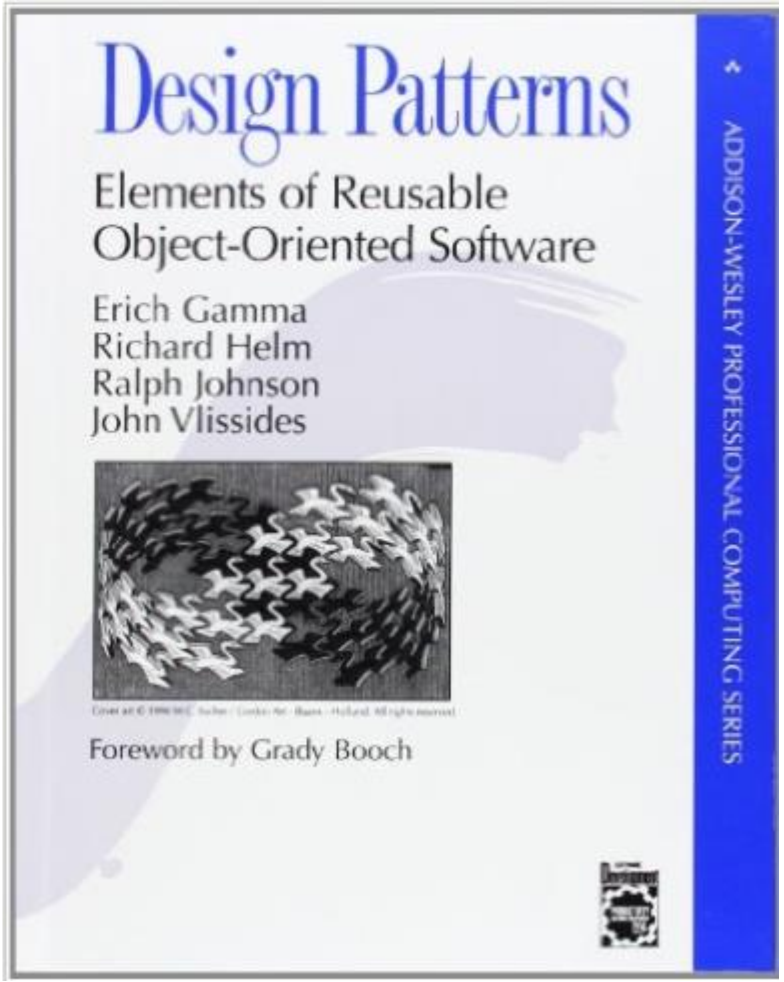
Benefits of Using Design Patterns

- Tested and used by other developers

- Make your code more readable and easy to modify

History of Design Patterns

- Introduced in 1987
- Popularized in this book:



Design Patterns

Elements of Reusable
Object-Oriented Software

Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides

Cover art © 1994 M.C. Escher / Cordon Art - Baarn - Holland. All rights reserved.

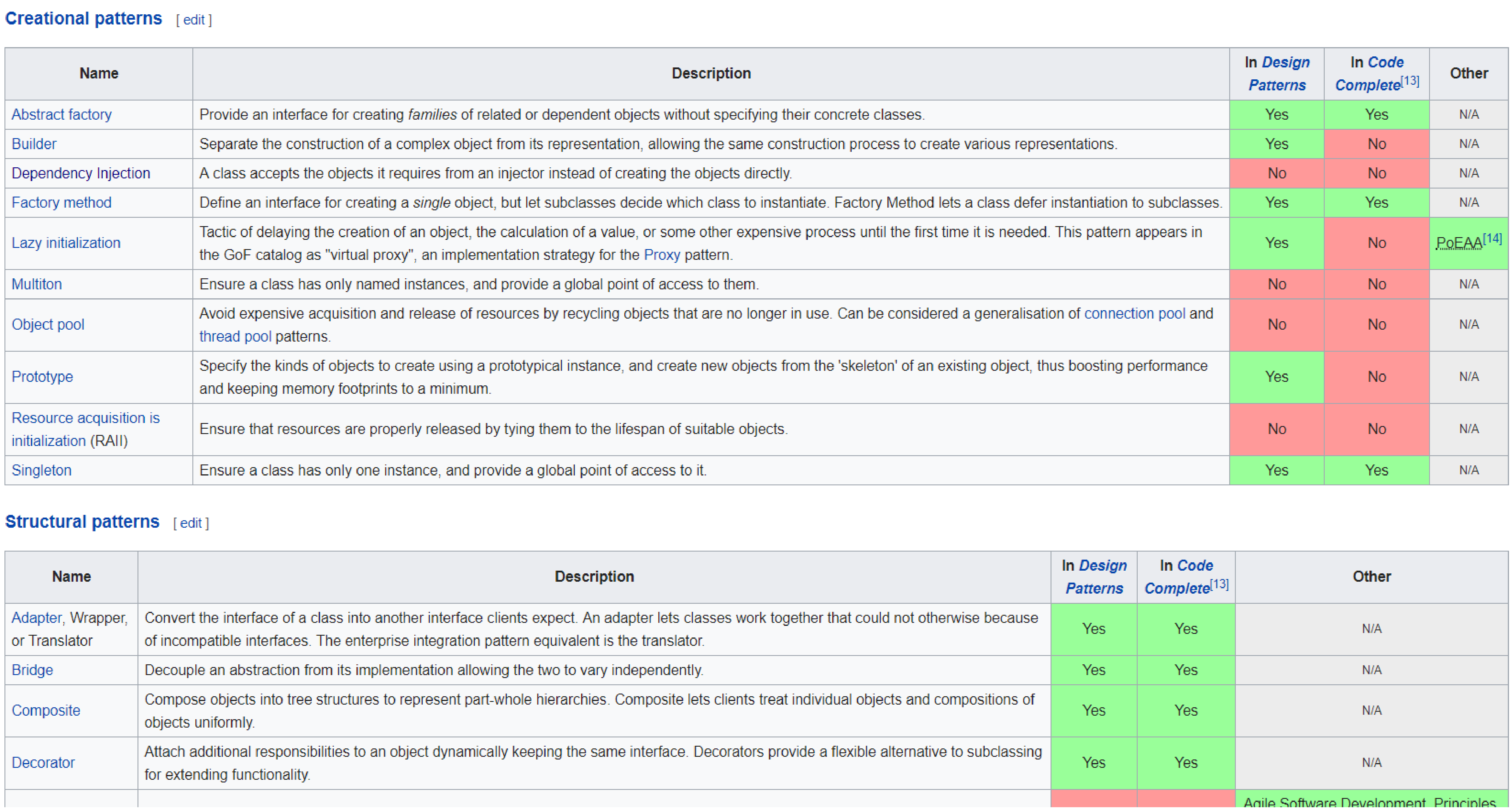
Foreword by Grady Booch

ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

Why Should I Care?

- Patterns are Micro-Architecture

- Always be familiar with the code!



Creational patterns [edit]

Name	Description	In *Design Patterns*	In *Code Complete*[13]	Other
Abstract factory	Provide an interface for creating *families* of related or dependent objects without specifying their concrete classes.	Yes	Yes	N/A
Builder	Separate the construction of a complex object from its representation, allowing the same construction process to create various representations.	Yes	No	N/A
Dependency Injection	A class accepts the objects it requires from an injector instead of creating the objects directly.	No	No	N/A
Factory method	Define an interface for creating a *single* object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.	Yes	Yes	N/A
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. This pattern appears in the GoF catalog as "virtual proxy", an implementation strategy for the Proxy pattern.	Yes	No	PoEAA[14]
Multiton	Ensure a class has only named instances, and provide a global point of access to them.	No	No	N/A
Object pool	Avoid expensive acquisition and release of resources by recycling objects that are no longer in use. Can be considered a generalisation of connection pool and thread pool patterns.	No	No	N/A
Prototype	Specify the kinds of objects to create using a prototypical instance, and create new objects from the 'skeleton' of an existing object, thus boosting performance and keeping memory footprints to a minimum.	Yes	No	N/A
Resource acquisition is initialization (RAII)	Ensure that resources are properly released by tying them to the lifespan of suitable objects.	No	No	N/A
Singleton	Ensure a class has only one instance, and provide a global point of access to it.	Yes	Yes	N/A

Structural patterns [edit]

Name	Description	In *Design Patterns*	In *Code Complete*[13]	Other
Adapter, Wrapper, or Translator	Convert the interface of a class into another interface clients expect. An adapter lets classes work together that could not otherwise because of incompatible interfaces. The enterprise integration pattern equivalent is the translator.	Yes	Yes	N/A
Bridge	Decouple an abstraction from its implementation allowing the two to vary independently.	Yes	Yes	N/A
Composite	Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.	Yes	Yes	N/A
Decorator	Attach additional responsibilities to an object dynamically keeping the same interface. Decorators provide a flexible alternative to subclassing for extending functionality.	Yes	Yes	N/A
				Agile Software Development, Principles

Source: https://en.wikipedia.org/wiki/Software_design_pattern

Patterns We'll Discuss

- Factory
- Repository
- Façade
- Command

Factory Pattern

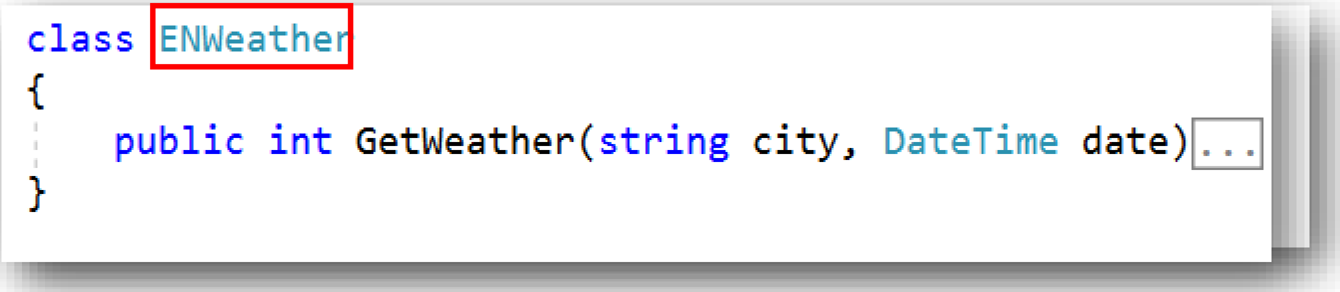
Factory Pattern

Creating objects without specifying the exact class of the object

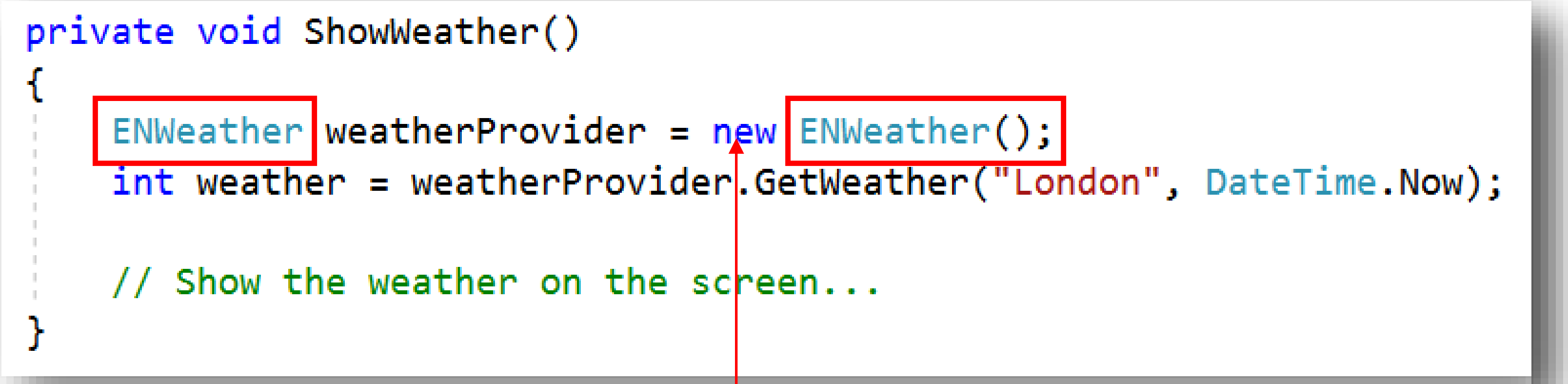
Factory Pattern Motivation

- Avoid strong coupling between classes

Factory Pattern Example – Weather



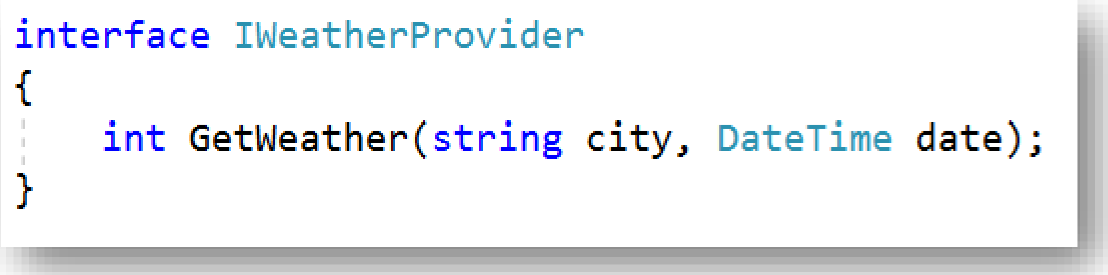
```
class ENWeather
{
    public int GetWeather(string city, DateTime date)...
}
```



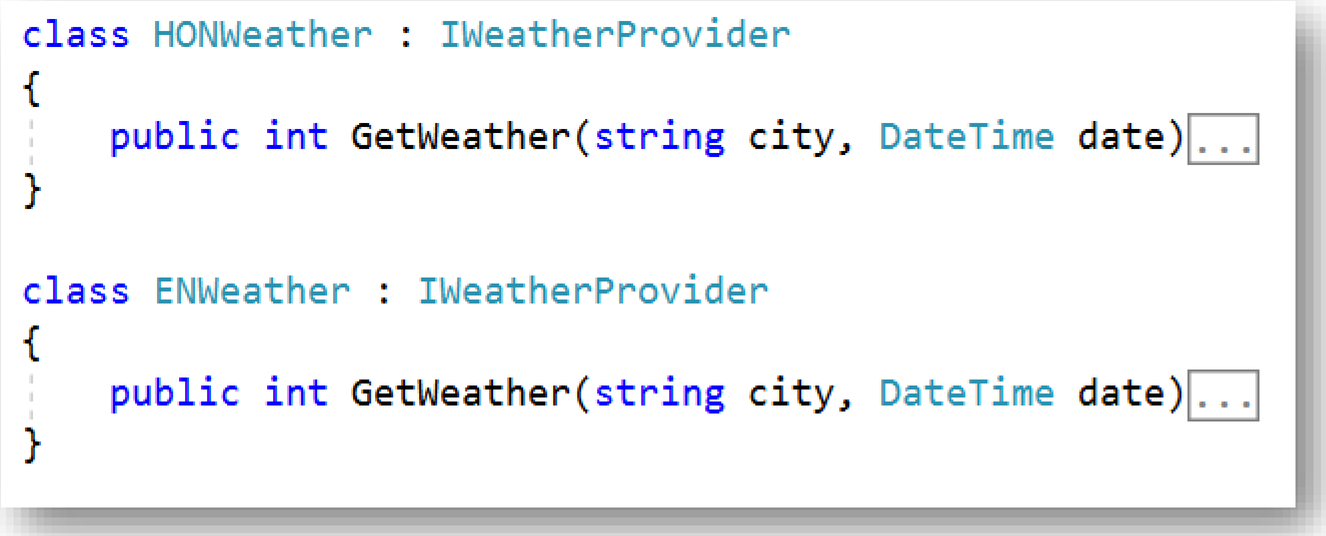
```
private void ShowWeather()
{
    ENWeather weatherProvider = new ENWeather();
    int weather = weatherProvider.GetWeather("London", DateTime.Now);

    // Show the weather on the screen...
}
```

New Is Glue

Factory Pattern Example – Weather

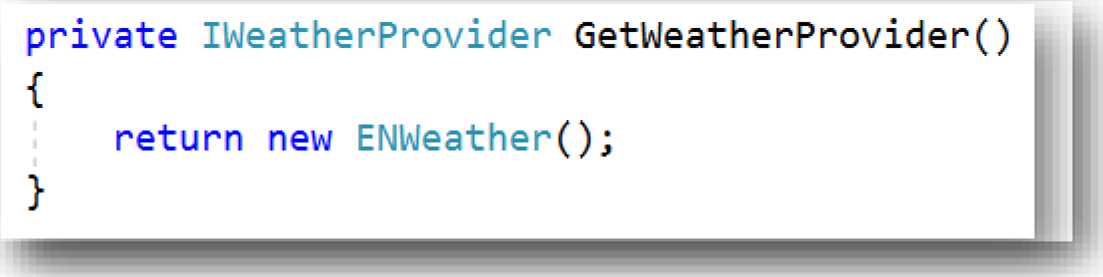


```
interface IWeatherProvider
{
    int GetWeather(string city, DateTime date);
}
```

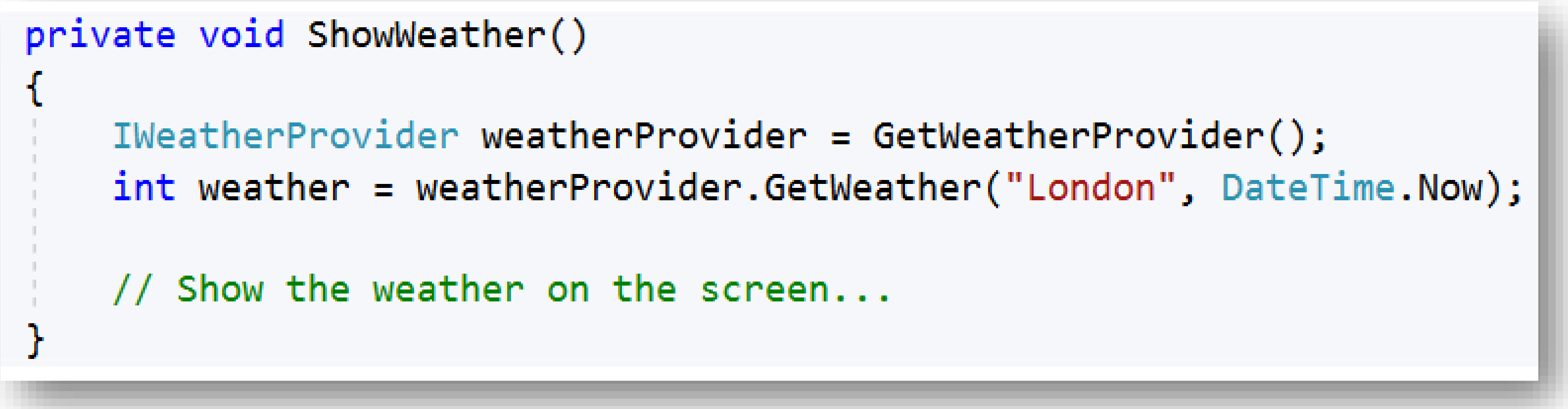


```
class HONWeather : IWeatherProvider
{
    public int GetWeather(string city, DateTime date)...
}

class ENWeather : IWeatherProvider
{
    public int GetWeather(string city, DateTime date)...
}
```

Factory Pattern Example – Weather

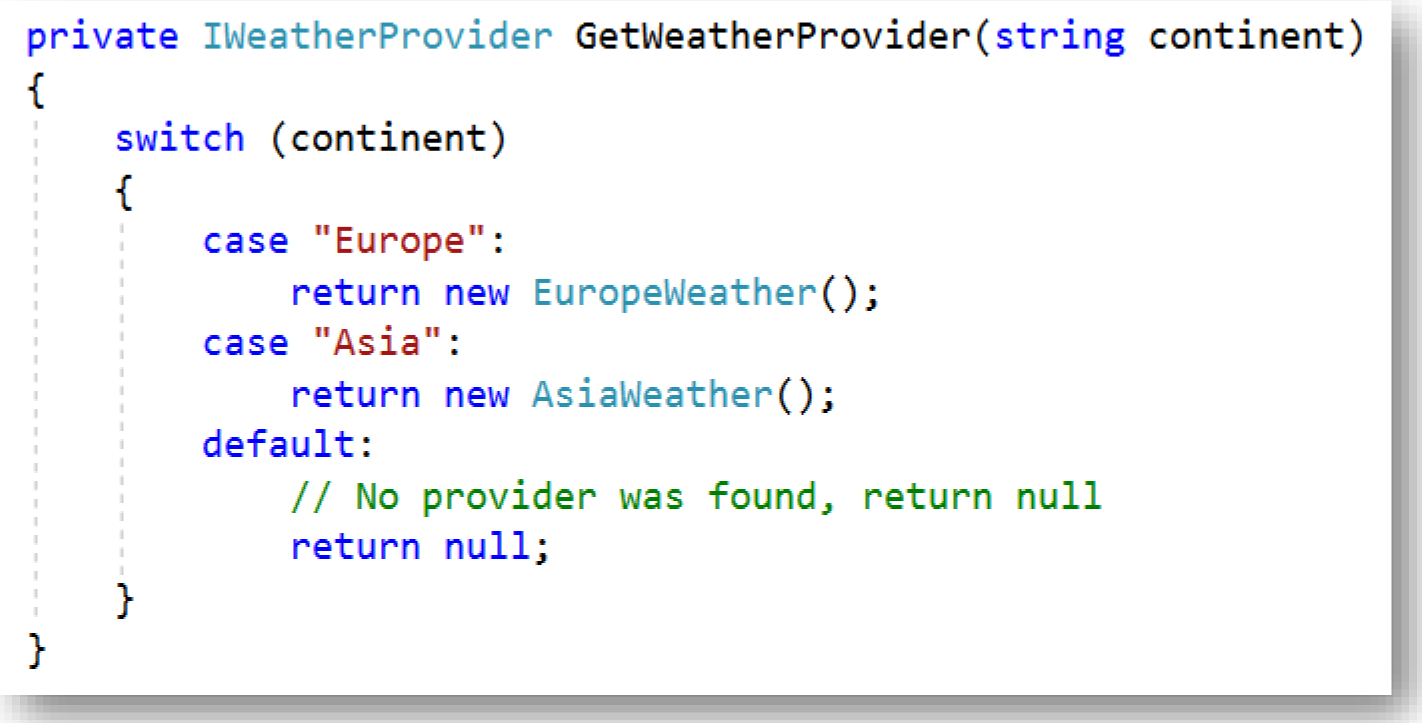


```
private IWeatherProvider GetWeatherProvider()
{
    return new ENWeather();
}
```



```
private void ShowWeather()
{
    IWeatherProvider weatherProvider = GetWeatherProvider();
    int weather = weatherProvider.GetWeather("London", DateTime.Now);

    // Show the weather on the screen...
}
```

More Types of Factory Pattern



```
private IWeatherProvider GetWeatherProvider(string continent)
{
    switch (continent)
    {
        case "Europe":
            return new EuropeWeather();
        case "Asia":
            return new AsiaWeather();
        default:
            // No provider was found, return null
            return null;
    }
}
```

Factory Pattern

- Hugely popular
- Base for other patterns
- Use it to avoid strong coupling

Repository Pattern

Repository Pattern

Modules not handling the actual work with the datastore should be oblivious to the datastore type

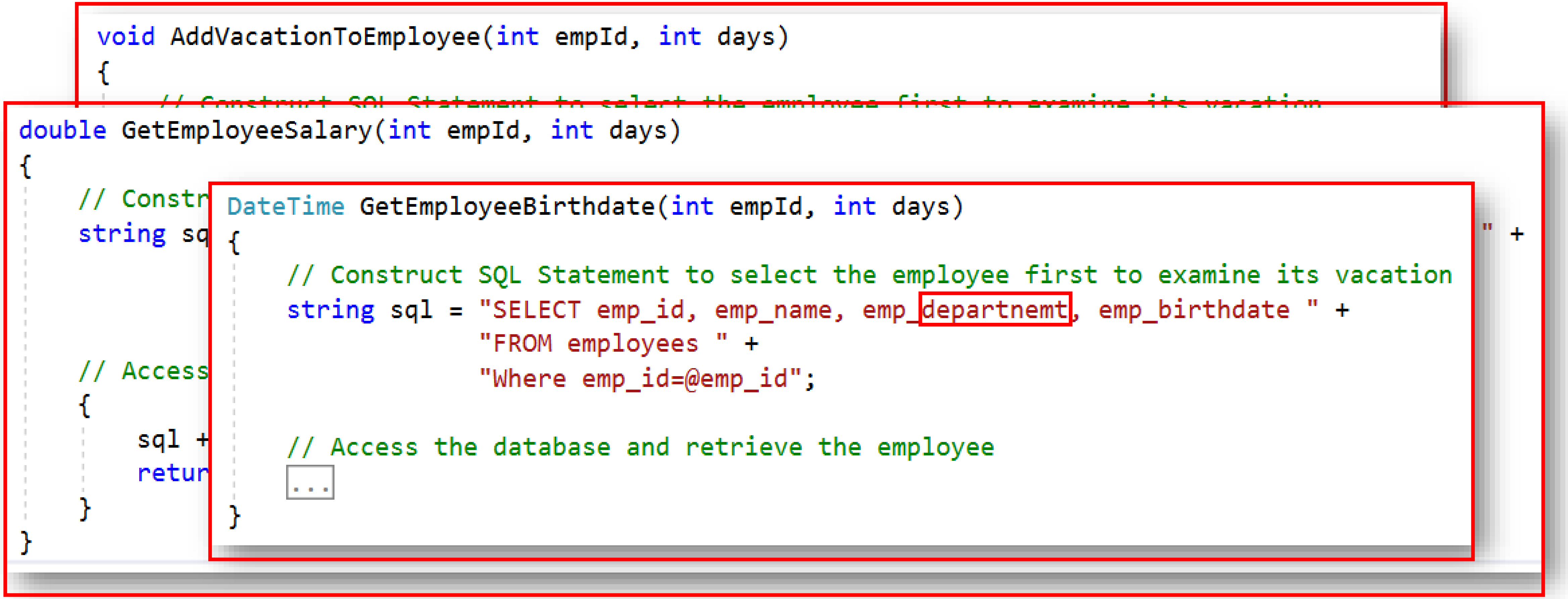
Repository Pattern

- Share similarities with the Data Access Layer
- DAL is for Architects
- Repository Pattern is for Developers

Repository Pattern - Example

- Human Resources Application
  - Create
  - Read by ID & by Department
  - UpdateName
  - Delete

Repository Pattern – Example

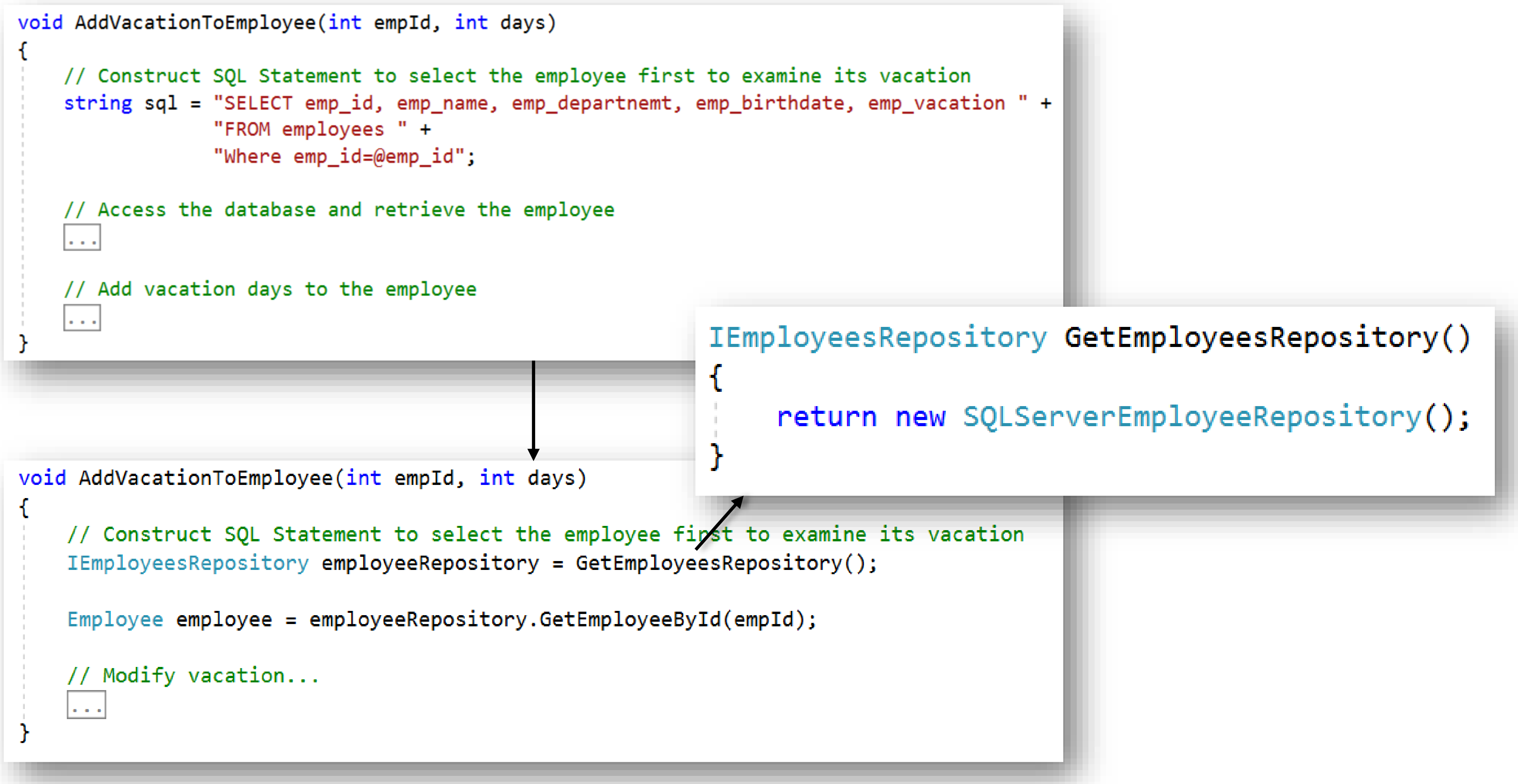


```
void AddVacationToEmployee(int empId, int days)
{
    // Construct SQL Statement to select the employee first to examine its vacation
```

```
double GetEmployeeSalary(int empId, int days)
{
    // Constr
    string sq                                                                      " +



    // Access
    {
        sql +
        retur
    }
}
```

```
DateTime GetEmployeeBirthdate(int empId, int days)
{
    // Construct SQL Statement to select the employee first to examine its vacation
    string sql = "SELECT emp_id, emp_name, emp_departnemt, emp_birthdate " +
                 "FROM employees " +
                 "Where emp_id=@emp_id";

    // Access the database and retrieve the employee
    ...
}
```

Repository Pattern – Usage

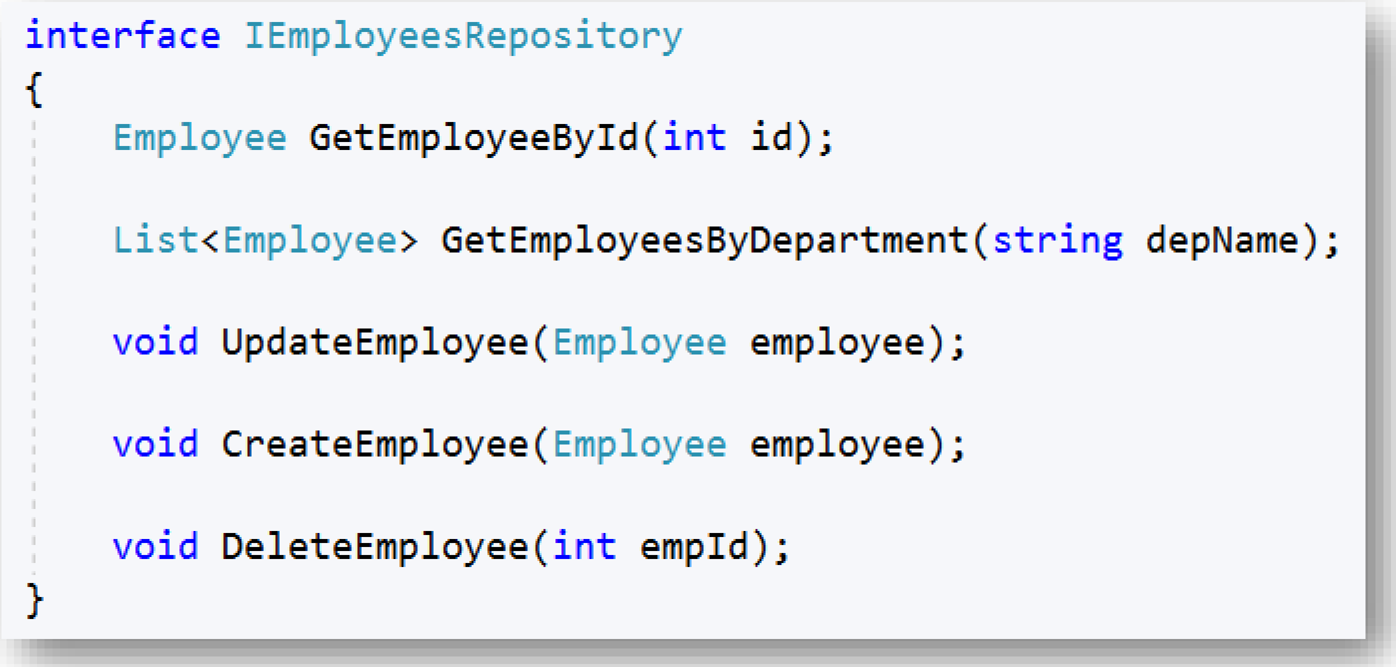


```
void AddVacationToEmployee(int empId, int days)
{
    // Construct SQL Statement to select the employee first to examine its vacation
    string sql = "SELECT emp_id, emp_name, emp_departnemt, emp_birthdate, emp_vacation " +
                 "FROM employees " +
                 "Where emp_id=@emp_id";

    // Access the database and retrieve the employee
    ...

    // Add vacation days to the employee
    ...
}
```

```
IEmployeesRepository GetEmployeesRepository()
{
    return new SQLServerEmployeeRepository();
}
```

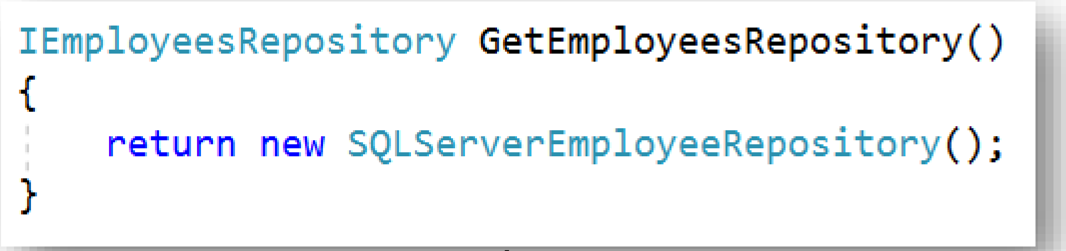
```
void AddVacationToEmployee(int empId, int days)
{
    // Construct SQL Statement to select the employee first to examine its vacation
    IEmployeesRepository employeeRepository = GetEmployeesRepository();

    Employee employee = employeeRepository.GetEmployeeById(empId);

    // Modify vacation...
    ...
}
```

Repository Pattern – Usage

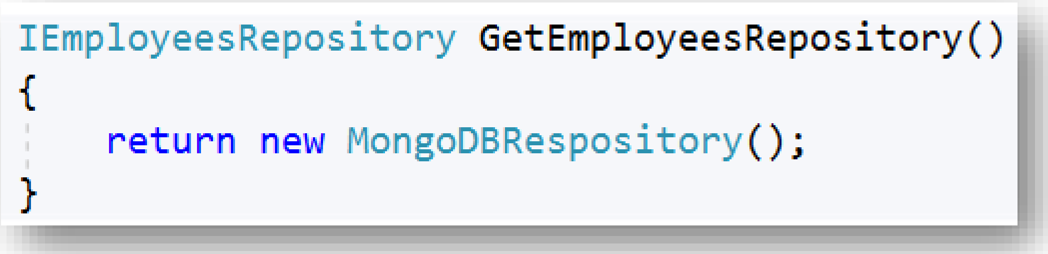


```
interface IEmployeesRepository
{
    Employee GetEmployeeById(int id);

    List<Employee> GetEmployeesByDepartment(string depName);

    void UpdateEmployee(Employee employee);

    void CreateEmployee(Employee employee);

    void DeleteEmployee(int empId);
}
```

Repository Pattern – Data Store Change



```
IEmployeesRepository GetEmployeesRepository()
{
    return new SQLServerEmployeeRepository();
}
```



```
IEmployeesRepository GetEmployeesRepository()
{
    return new MongoDBRespository();
}
```

Repository Pattern

- There are more advanced implementations
  - Generic classes
  - Inheritance
  - Extension Frameworks
- Very useful, Use it!

Façade Pattern

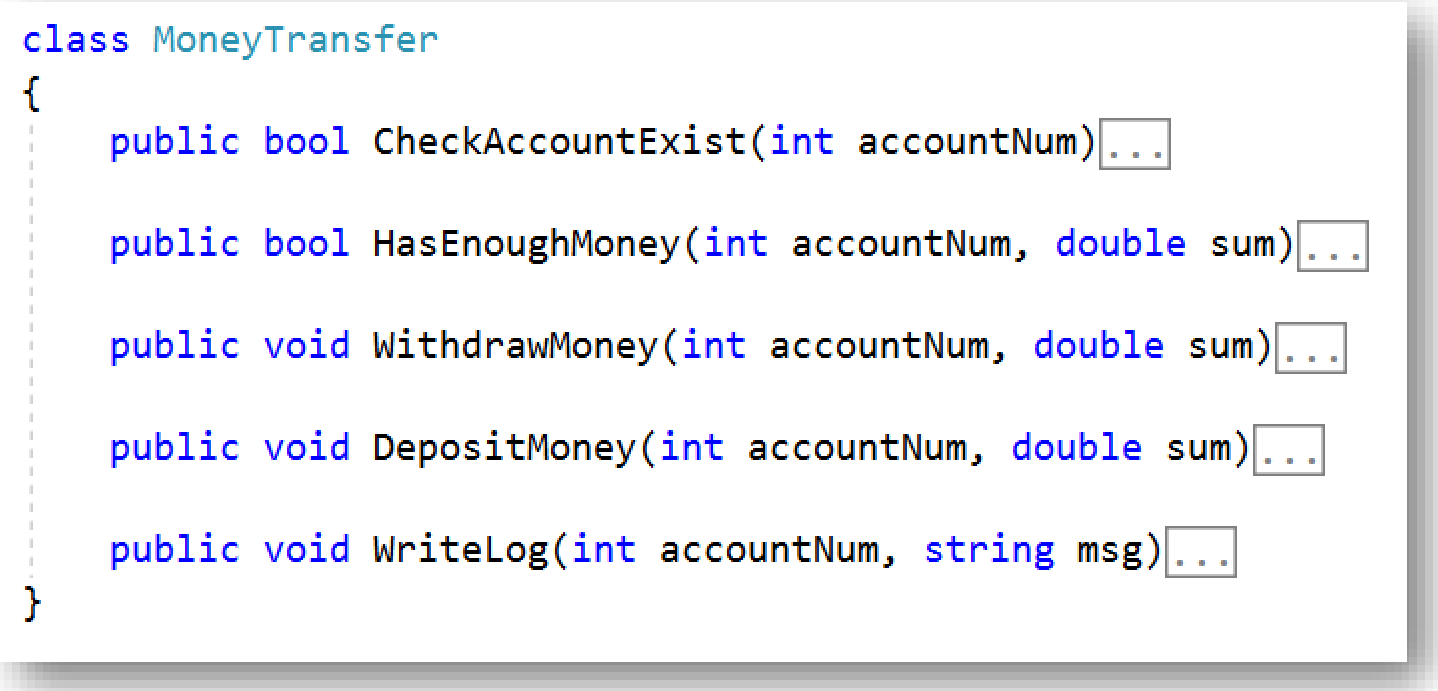
Façade Pattern

Creating a layer of abstraction to mask complex actions

Façade Pattern - Example

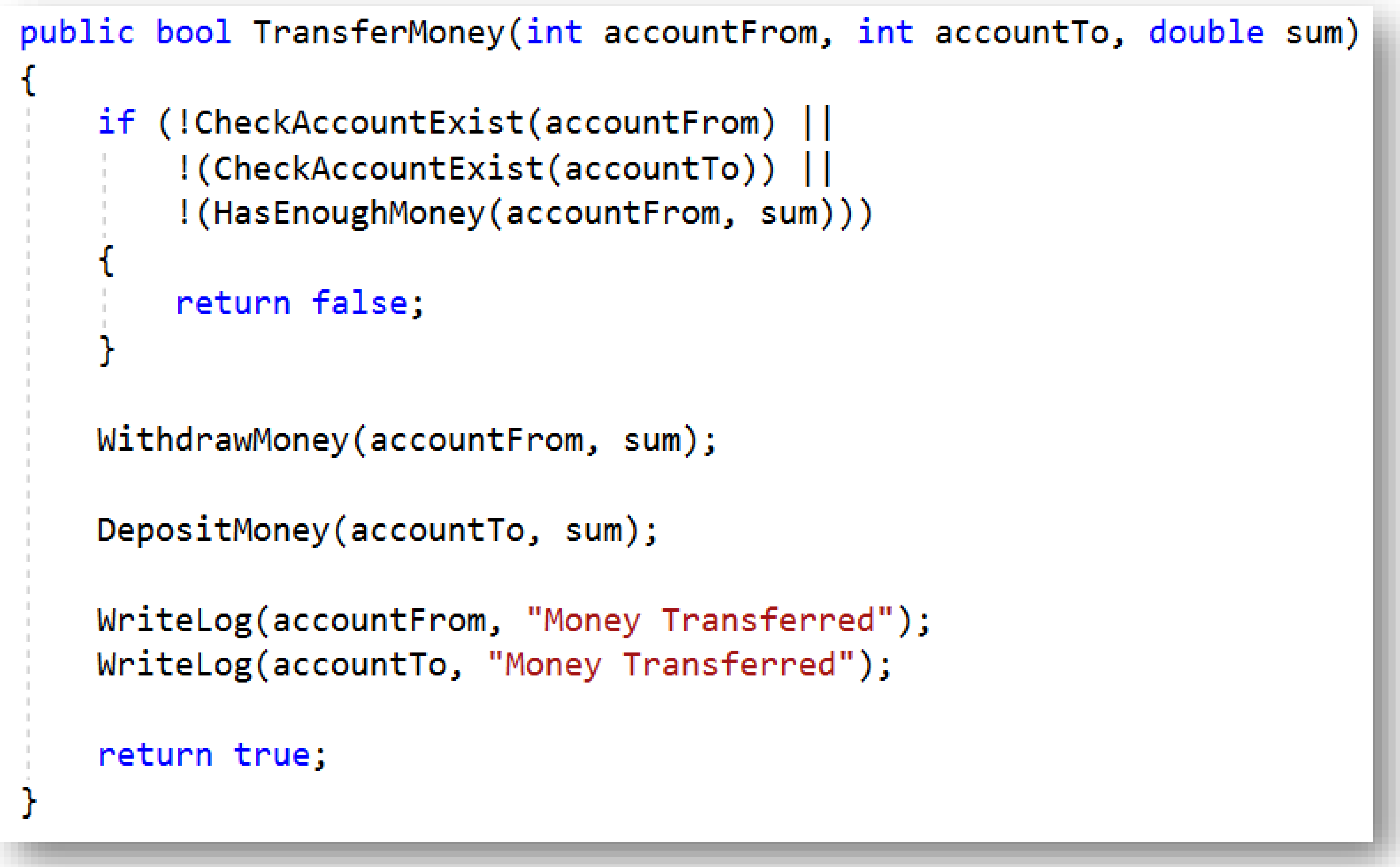
- Banking application
- Transfer money
  - Make sure accounts exist
  - Make sure the first account has enough money
  - Withdraw money from first account
  - Deposit money in second account
  - Add event in account log

Façade Pattern – Example



```
class MoneyTransfer
{
    public bool CheckAccountExist(int accountNum)...

    public bool HasEnoughMoney(int accountNum, double sum)...

    public void WithdrawMoney(int accountNum, double sum)...

    public void DepositMoney(int accountNum, double sum)...

    public void WriteLog(int accountNum, string msg)...
}
```

Façade Pattern – Usage



```
public bool TransferMoney(int accountFrom, int accountTo, double sum)
{
    if (!CheckAccountExist(accountFrom) ||
        !(CheckAccountExist(accountTo)) ||
        !(HasEnoughMoney(accountFrom, sum)))
    {
        return false;
    }

    WithdrawMoney(accountFrom, sum);

    DepositMoney(accountTo, sum);

    WriteLog(accountFrom, "Money Transferred");
    WriteLog(accountTo, "Money Transferred");

    return true;
}
```

Command Pattern

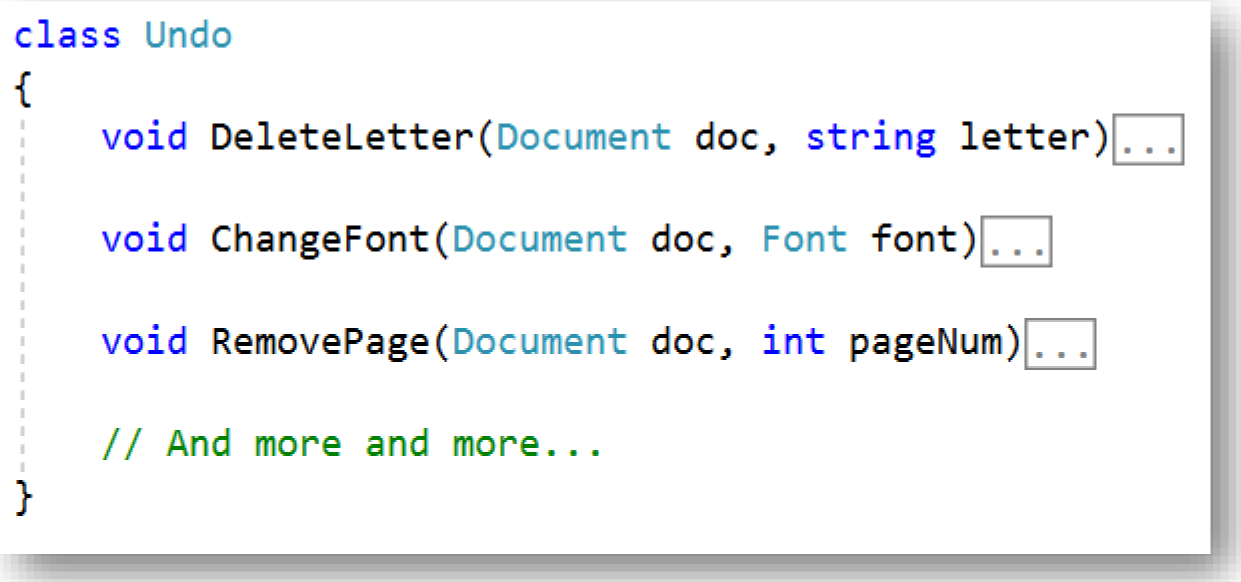
Command Pattern

All the action’s information is encapsulated within an object

Command Pattern - Example

- Undo mechanism

- Naïve implementation:

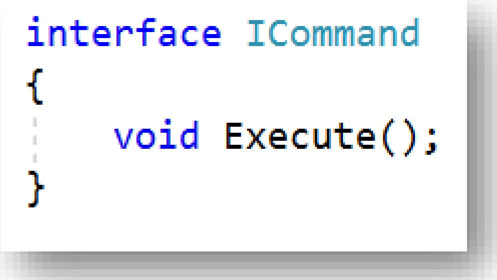


```
class Undo
{
    void DeleteLetter(Document doc, string letter)...

    void ChangeFont(Document doc, Font font)...

    void RemovePage(Document doc, int pageNum)...

    // And more and more...
}
```

This is Bad!

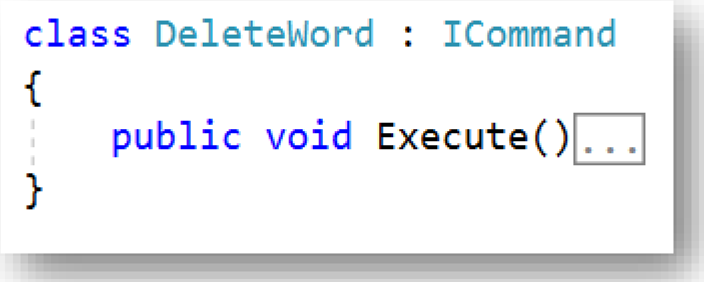
Command Pattern – Usage

1. ICommand Interface:

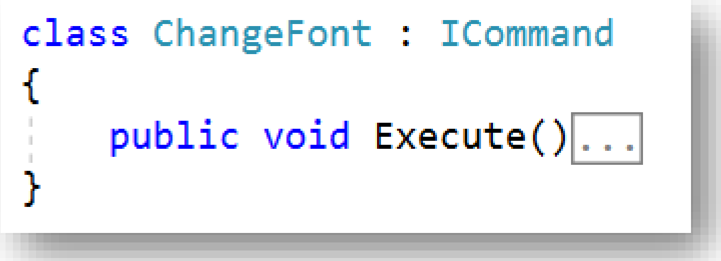


```
interface ICommand
{
    void Execute();
}
```

2. Command Classes:



```
class DeleteWord : ICommand
{
    public void Execute()...
}
```

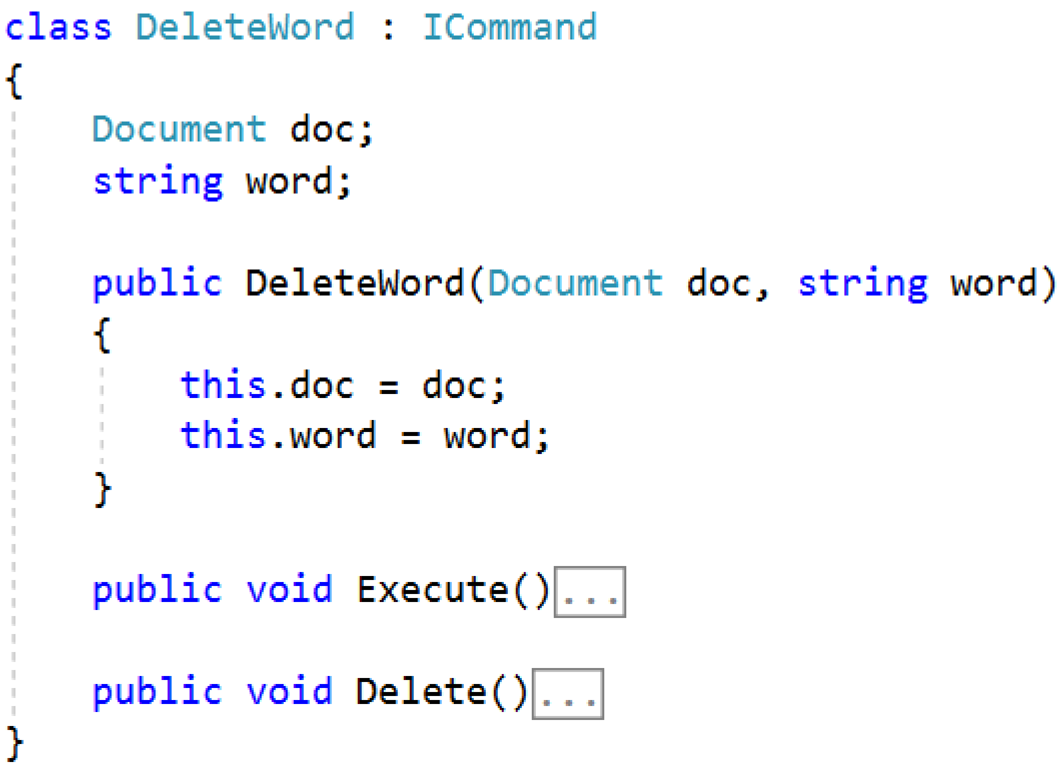


```
class ChangeFont : ICommand
{
    public void Execute()...
}
```

...

Command Pattern – Usage

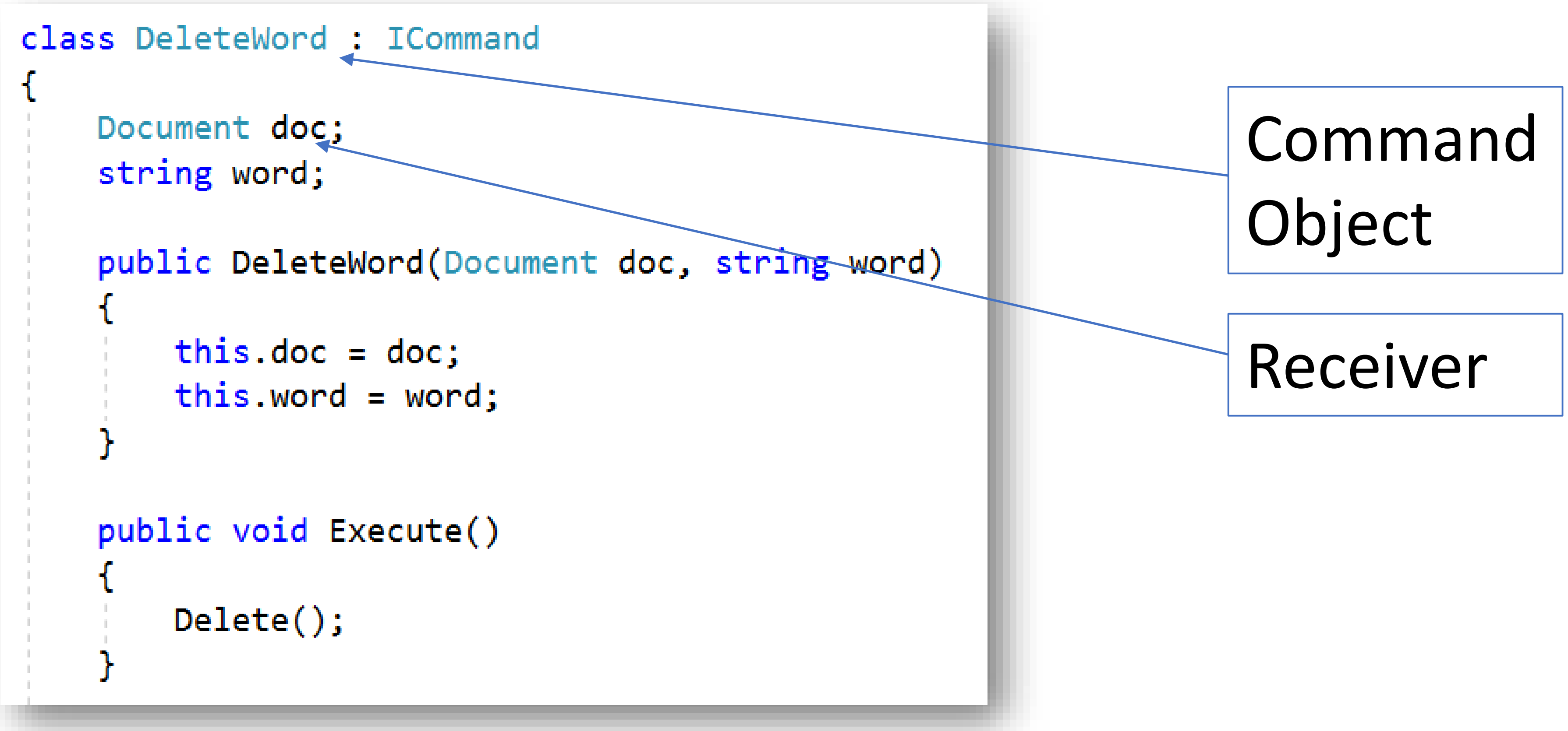
3. Get reference to relevant objects:



```
class DeleteWord : ICommand
{
    Document doc;
    string word;

    public DeleteWord(Document doc, string word)
    {
        this.doc = doc;
        this.word = word;
    }

    public void Execute()...

    public void Delete()...
}
```

Command Pattern – Usage

4. Implement the Interface:



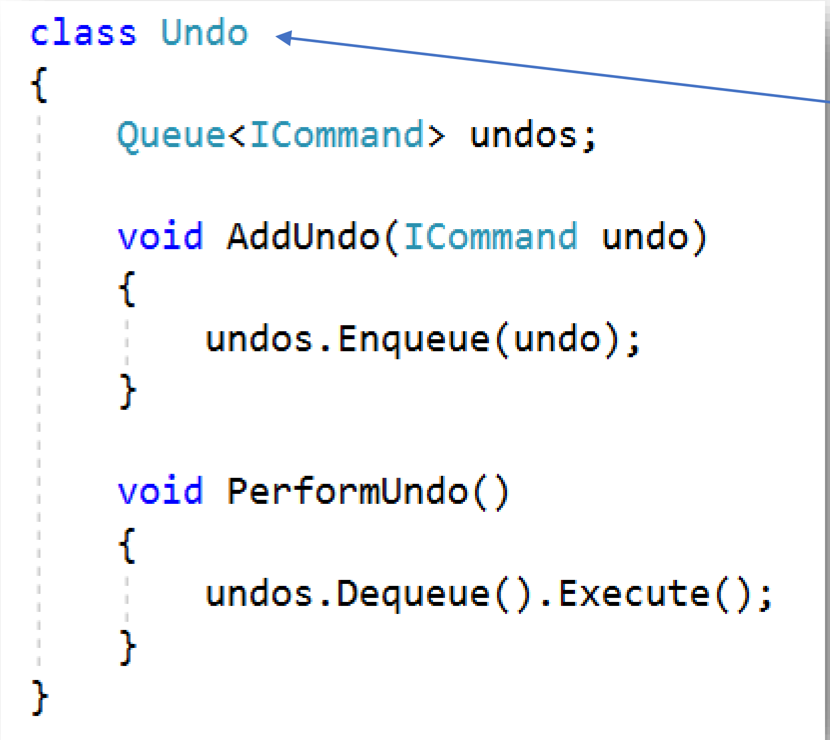
```
class DeleteWord : ICommand
{
    Document doc;
    string word;

    public DeleteWord(Document doc, string word)
    {
        this.doc = doc;
        this.word = word;
    }

    public void Execute()
    {
        Delete();
    }
```

Command Object

Receiver

Command Pattern – Usage

5. Implement the Undo mechanism:



```
class Undo
{
    Queue<ICommand> undos;

    void AddUndo(ICommand undo)
    {
        undos.Enqueue(undo);
    }

    void PerformUndo()
    {
        undos.Dequeue().Execute();
    }
}
```

Invoker

Design Patterns – Summary

- Factory
- Repository
- Façade
- Command

Design Patterns – Summary

- Design consistent, flexible, readable and easy to maintain software
- Use only the patterns you need