Recommended books

1. Eric Freeman, Elisabeth Robson, Bert Bates, Kathy Sierra - *Head First Design Patterns*;
2. Robert C. Martin - *Agile Software Development, Principles, Patterns, and Practices*;
3. Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, Grady Booch - *Design Patterns: Elements of Reusable Object-Oriented Software*;
4. Bertrand Meyer - *Object Oriented Software Construction*.

Course Structure

1. Review of basics of object-oriented programming

2. Elements of UML (Unified Modeling Language)
   - use case diagrams
   - class diagram

3. S.O.L.I.D design principles (5 principles)
   - SRP (Single Responsibility Principle)
   - OCP (Open Closed Principle)
   - LSP (Liskov Substitution Principle)
   - ISP (Interface Segregation Principle)
   - DIP (Dependency Inversion Principle)

4. Design Patterns - design templates on case studies
   - Introduction of scenario
   - Solution proposition
   - Analysis of advantages/disadvantages
   - Extract the pattern used in the case study
   - Overview of the design pattern

5. Templates
   - Structural
     - Adapter Pattern
     - Facade Pattern
     - Composite Pattern

   - Behavioral
     - Command Pattern
     - State Pattern
     - Template Method Pattern
     - Strategy Pattern
     - Observer Pattern

   - Creational
     - Singleton Pattern
     - Abstract Factory Pattern
     - Factory Method Pattern

6. Elements of software architecture

Course objective

- The use of object-oriented methodology in the design of extensible maintainable software systems;
- The use of efficient solution design templates in solving design problems;
- Application of design principles.

Essential OOP definitions

Class

- Prototype indicating the attributes and behavior of a family of objects.
- Structure containing data and methods.

Objects

- An entity that has a name, a certain state and a behavior in that state.
- A variable having class as type, a memory address, space allocated by constructor for each of its variables and a lot of methods.

Methods/Operations

- A member function (operation) of a class. They implement the behaviors of objects.

Constructors

- Special methods of creating objects in their initial state.

Destructors

- A method of releasing resources occupied by an object when its life is over.

Lifetime of an object

- The period between the creation of the object (by the constructor) and its destruction by destructor.

Encapsulation

- Separation of implementation details from the public interface to the outside world and hiding them.
- Communication with the object is done only through the public interface.

Hidden implementation

- All data and hidden methods of the class.

Public interface of objects

- The multitude of public object methods.

```
public class C
{
    //hidden implementation details
    private int _data;
    private void PrivateMethod();

    //public interface
    public C(){}
    public void PublicMethod();
    //getters
    //setters
}
```

Message

- Calling an object's method.

```
class A
{
    public void a(){}
}

class B
{
    A refA;
    public void b()
    {
        refA.a(); //b sends a message "a" to the refA object
    }
}
```

Behavior of objects

- Actions of an object in a certain state;
- Behavior is the way in which the current state of an object is affected when appealed to any of its methods (from the public interface).

Condition of objects

- Particular values of object attributes.

Getter/Setter

- Operations by which we can obtain/modify an object's data in a controlled way. Useful for restricted visibility via encapsulation:
- Get method - gets the current state of the object (always public method).
- Set method – changes the current state of the object on use.

Inheritance - Derived classes/Generalization/Specialization

- We say that a class A is a generalization of a class B if for any object of type B we can say that it is "sort of" a type A object. **BUT BEWARE!** There can be differences from a behavioral (not structural) point of view.
- The square object is a Special case of the rectangle object – the square can be derived from a rectangle. But the behavior remains the same.

Polymorphism

- Ability to transmit a message with the same signature to different objects of a class hierarchy, without knowing the recipients.
- To benefit from (semantic) polymorphism we need the following elements:
  - A hierarchy of classes (a base class and a number of derived classes);
  - A common (virtual) method declared in the base class and redefined in derived classes. The method call is made via a pointer to the base class.
- Important note: polymorphism is extremely important for writing reusable/extensible software components (libraries).

```
//Example - in a possible new derivation of class B (a new extension), the code of class C will
remain FUNCTIONAL without being recompiled. (We say it is closed to changes).

class B
{
 public virtual void method(){...}
}

class D: B
{
 public override void method(){...}
}

class C: B
{
 public override void method(){...}
}

class Client
{
    public void Process(B refB)
    {
        refB.method(); //this is a polymorphic message (method call)
    }
}

Client c = new Client();

c.Process(new B()); //calls from B
c.Process(new C()); //calls from C
c.Process(new D()); //calls from D
```

Abstraction

- The process by which different objects belonging to the same (related) domain are treated uniformly by a general, abstract concept. (For example, 'Square', 'Circle' ➔ “Shape” and 'Guitar', 'Piano' ➔ “Instrument”)
- The advantage of abstraction is that the core concept does not change over time!! Therefore, if we manage to build software components that depend ONLY on abstractions, they will remain unchanged (to new requirements).
- **IMPORTANT RULE:** Client classes must depend on the most general things and concrete/abstract classes must depend on ABSTRACTIONS (because these are fixed in time).

Abstract class

- Abstract classes represent the abstraction of the real world into the programming world.
- Appears at the top of class hierarchies;
- Cannot be instantiated (because it is far too general to know all the details of implementation. Some methods are declared abstract - they do not have implementation)
- Represents a contract between Client classes and the classes providing services (called Server classes) in the hierarchy.
- The contract is represented by the multitude of public methods in the abstract class that can be used by the Client class by reference to the base class.

Interface

- The interface is a particular case of abstract class: it contains only declared methods with no implementation.
- The contract role of an interface is to highlight the common public interface of a hierarchy class.

```
public interface ILogger
{
    void Log(LogModel model);
}

//customer class!!!
class LoggerManager
{
    ILogger Logger { get; set;}

    public void Log(LogModel model)
    {
        Logger.Log(model);
    }
}

internal FileLogger : ILogger
{
    public void Log(LogModel model)
    {

    }
}

//LoggerManager class is the client class that benefits from the ILogger class hierarchy!!!
//It Uses various implementations of the interface without knowing them!
```

Virtual functions

- A method of a base (general) class, which can be called only by the object that prefixes the call.
- The call type is given by the type of the created object, not by its declaration.

```
class B
{
    public virtual void m(){...}
}

class D: B
{
    public override void m(){...}
}

B ref = new D();
ref.m(); ///here the class D variant is chosen (according to the virtual functions principle)

//An object of class D contains two variants for method 'm', saved in a table called VFT (Virtual
Functions Table).
```

Dynamic binding (late binding)

- Closely related to virtual functions. Objects that prefix a virtual function call are created dynamically with the 'new' operator. This prevents the compiler from knowing the 'sequence' of code to be executed at compile time. Thus the code executes at run time.

UML (Unified Modeling Language)

Software system development process

1. Analysis and specification of requirements
2. Software design and implementation (design & code)
3. Testing
4. Maintenance.

General elements of UML (Unified Modeling Language)

UML is a modeling language that allows analysis and modeling of software systems and creating specifications for the requirements. It uses graphical and text elements.

Commonly used UML Diagrams are use-case, class, sequence, collaboration, state, component and activity.

Commonly used CASE (Computer Aided Software Engineering) tools for UML include Rational Rose, Magic Draw, Visio, Visual Paradigm and many more. Such tools can generate source code in various programming languages given a good model. If you have a program's source code, you can reverse engineer its model.

Object-oriented analysis and design

Features of a quality software system are:

1. Extensibility - new functionalities - lower costs
2. Reusability - the system proposes general solutions, which can be reused in other software systems
3. Robustness - the system is well tested, behaves well in limiting situations
4. Maintainability

Use case diagram

- It describes use cases for the user(actor) with the software system(scenarios).
- Actions are represented with an ellipsis and usually are specified with verbs indicating
- An actor can have one or more use cases.
- We say that actor A is a generalization of actor B if B has the same actions as A. It can include additional scenarios.
- To describe a use case, we can use a Template description that includes
  - Author
  - History
  - Preconditions
  - Post conditions
  - Implementation problems
  - Extension points (how the use case can be extended through various implementations)
- Relationships can be established between use cases, labeled as:
  - <Extends> - For exceptional situations that can arise in the execution sequence of a use case. They are implemented as method calls conditioned by IF (or Switch) statements.
  - <Includes> - Indicates that one use case is part of a more complex case. (highlights the hierarchy of more complex cases)

```
void MainCase()
{
    if(condition)
    {
        Extension();
    }
    //main code
}
```

```
//Example
public void Authenticate(string username, string password)
{
    if(!IsValidCredentials(username, password))
    {
        InvalidCredentials(username, password);
    }
    else
    {
        //authorization code
    }
}
```

```
void MainCase()
{
    Subcase1();
    Subcase2();
}

void MainCase2()
{
    Subcase1();
    Subcase3();
}
```

Class diagram

- Indicates the structure of the system.
- Shows what the classes/interfaces of the system are and the relationships between them.
- Used in the design and implementation stage (design & code).
- Represented by a rectangle with three spaces for class Name, Attribute/Value pairs and Methods.
- The syntax for attributes is [access_specifier] name: [type]. The access specifier A.K.A the encapsulation is denoted with:
    - "+" for public;
    - "-"for private;
    - "#" for protected.
- Static data/methods are underlined.

Relations between classes

1. **Association**

We say that between classes A and B there is a relationship of association whether objects in class A are aware of objects in class B and/or vice versa.

Elements of association include:

- Name [of action]
- Roles - at the ends of the association the role of each class is noted
- Multiplicity – denotes how many objects of each class participate in the association (constant, in range, many). They are transformed into corresponding data structures.
- Navigability – shows direction of message flow within the association; A ➔ B shows that messages circulate from A to B and not in the other direction.

We draw association between classes A and B if at least one of the following conditions is fulfilled:

```
// 1. Class A contains a reference to Class B of member variable type.
class A
{
    private B _refB;
}

// 2. When class B appears as a parameter of a class A method
class A
{
    public void Method(B refB)
    {
    }
}

// 3. Class B appears as a returned type of a method from A

public B method()
{
    ///...
    return refB;
}

// 4. Class B appears as an exception in a Class A method

public void method()
{
    try{}
    catch(B refB)
}

public void method()
{
 throw new B();
}

// 5. Class B appears as a local variable in a class A method

class A
{
    public void method()
    {
        B refB;
        //....
    }
}
```

2. **Aggregation and composition**

There are particular cases of association indicating a part-whole relationship.

Aggregation is when an object of class A contains one or more objects of class B.

In the case of aggregation, an object can belong to more than one whole and is possible to exist outside of the relation.

Composition is a particular case of aggregation. The whole depends on its component parts and vice versa, a component only exists as a whole. If the whole thing disappears (it is destroyed, it ends its life), the component parts disappear.

They cannot exist in other objects.

3. **Generalization and Specialization**

A class A is a generalization of a B if we can say that the objects of class B are "sort of" type A objects. "Sort of" refers to behavior, but not attributes. They act similar to the derived object.

Generalization allows code reuse BUT should not be used in certain situations because generalization **introduces a high degree of dependency** between classes.

In some cases whenever we need the functionality of a class (Server class), it is preferable to use aggregation/combination at the expense of inheritance.

Classroom design techniques

Given the system requirements, how do we find the classes/interfaces and the relationships between classes? The process is iterative and every step refines the model to meet the requirements.

1. **Noun identification technique**

Requirements described in natural language are analyzed and nouns in the text that have to do with the system are highlighted.

From the list of candidate classes the very general ones are removed.

This technique provides a first list of classes, without indicating anything about the relationships between classes

2. **Technical, CRC cards**

A CRC (Class, Responsibility, Collaboration) card indicates:

- The main responsibilities of the class which help form attributes and methods;
- Relationships with other classes (associations, compositions, aggregations, generalizations) from collaborations.

The CRC card is drawn for each class.

S.O.L.I.D. design principles

What does the acronym stand for?

- S RP (Single Responsibility Principle)
- O CP (Open Closed Principle)
- L SP (Liskov Substitution Principle)
- I SP (Interface Segregation Principle)
- D IP (Dependency Inversion Principle)

Design by Contract technique

(See the book Object Oriented Software Construction, B. Meyer)

Server class 'signs' a contract with the Client class via abstract classes or interfaces. All public methods in the interface/abstract class can be used by the Client.

For each method two properties (logical formulas) are established:

- Precondition - a true invariant before calling the service (method)
- Post condition - a true invariant after calling the service (method).

```
class Server
{
    //Precondition "P"
    public void Method() {...}
    //Post condition "Q"
}
```

According to the Design by Contract technique, the client is obliged to call the method in a context where the precondition is true. The server is obliged to provide Post condition.

The advantage is that it simplifies Server class code because the Server class no longer needs to check (via "if" statements) that the precondition is true.

SRP (Single Responsibility Principle)

A class must contain one basic responsibility.
As a consequence, the corresponding class will have 'only one reason' to be modified, changed.
The code is easy to maintain and test.
The bigger advantage would be that the class will only have one reason for change - extension points.

SRP – The single responsibility principle and OCP – The open-closed principle

SRP dictates that a software entity (class/method) must be open to extensions and closed to changes.

A class is designed to implement one or more functionalities in the beginning. The class must be designed in such a way that we can change, optimize or add new features without modifying the current source code (without having to recompile the class), which refers to the closure part.

The class, therefore, is well defined, compilable and does not need to be modified if new functionality is desired.

```
//Example: Implement a class for error logging, console information.
public enum LogLevel
{ Info, Error, Debug, Warn }
```

```
class Logger
{
    public static void Log(string message, LogLevel level)
    {
        Console.WriteLine("{0} - {1} - {2}", DateTime.Now, level.ToString(), message);
    }
}
//What if it requires logging into a file? Or a database?
//The previous class obviously violates the OCP in this respect.
//Solution: abstract this functionality through an abstract class or interface

public interface ILog
{
    void Log(string message, LogLevel level);
}

class ConsoleLogging : ILog
{
     public void Log(string message, LogLevel level)
    {
        Console.WriteLine("{0} - {1} - {2}", DateTime.Now, level.ToString(), message);
    }
}

class FileLogging : ILog
{
    public FilePath {get; private set;}

    public FileLogging(string filePath)
    {
        FilePath = filePath;
    }
    public void Log(string message, LogLevel level)
    {
        Console.WriteLine("{0} - {1} - {2}", DateTime.Now, level.ToString(), message);
    }
}
//open to modifications, open to extensions, because the possibility of logging has been isolated
in a separate class hierarchy (via ILog interface)
class Logger
{
    ILog _log; /// extension point

    public ILog Log {get{return _log;};set {_log=value;}}

    //extensible method (open to changes)!!!
    public void Log(string message, LogLevel level)
    {
        _log.Log(message, level); //polymorphism
    }
}
```

In general, when we want a class to follow the OCP from a particular point of view (a certain functionality), the solution would be an abstract class or interface (ILog) which acts as a contract between the class and possible variants of functionality respectively.

The reference to the interface (abstract class) will be called an extension point.

Another classic example is when we have Client and Server classes. The Client class benefits from a service from the Server class.

<u>In general, a class is called a "Client" class if it benefits from the services (methods) another class, called the "Server" class.</u>

```
//Suppose that in the initial version the classes are associated at a low-level level.
class Client
{
    Server _server; //server class is concrete class

    public void DoJob()
    {
        _server.Job();
    }
}

//If we want to be able to change the service without affecting the Client class, the solution
is OCP. Thus, we abstract the service.
ISever interface
{
    void Job();
}

class MailServer : IServer
{
    public void Job() {  }
}

class FTPServer : IServer
{
    public void Job() {  }
}

class Client
{
    IServer _server; //high-level dependency

    public void DoJob()
    {
        _server.Job();
    }
}
```

The above technique is also called "dependency break". The Client class no longer depends <u>directly</u> on the Server class. It depends on the contract between the two made with the help of the interface.

But how do you go about getting OCP right?

- Identify the main functionality (see CRC cards) of the class.
- Check if the respective functionality can be replaced/optimized over time.
- If yes, we apply abstraction and the initial class will depend on something abstract because what is abstract is generally fixed.

Liskov Substitution Principle (LSP)

OCP recommends the use of abstraction, which leads to class hierarchies (inheritance).

```
class Client
{
    AbstractTask _task;

    public void Task { get {return _task;} set {_task = value;} }

    //extendable!
    public void DoTask()
    {
        _task.Job();
    }
}

abstract class AbstractTask
{
    public abstract void Job();
}
```

Given that nothing is known about derived classes in the Client class and yet we assume the Job is implemented by them, we must ensure that the method can be called in a valid context indicated by the AbstractTask abstract class. Therefore, classes derived from AbstractTask must provide a valid execution context for the Job method.

```
class ConcreteTask1: AbstractTask
{
    public void Job()
    { //implementation that must respect the contract imposed by the base class!!! }
}
```

LSP shows precisely how this hierarchy is correctly constructed.

If an entity (method/class) depends on a reference to a base/interface class, it must remain functional for all derived classes **without knowing anything about the derived classes**.

In other words, if we replace that reference with ANY class that implements the abstract class/interface, the entity remains functional. **The important thing is the substitution has to be valid so the behavior is expected.**

LSP violation example

```java
class Rectangle {
    protected int width, height;

    public Rectangle() {  }

    public Rectangle(int width, int height) {
        this.width = width;
        this.height = height;
    }

    public int getWidth() { return width; }
    public int getHeight() { return height; }
    public void setWidth(int width) { this.width = width; }
    public void setHeight(int height) { this.height = height; }
    public int getArea() { return width * height;}
}

class Square extends Rectangle {
    public Square() {  }

    public Square(int size) { width = height = size; }

    public void setWidth(int width) {
        super.setWidth(width);
        super.setHeight(width);
    }

    public void setHeight(int height) {
        super.setHeight(height);
        super.setWidth(height);
    }
}

//Explanation: The invariant in the rectangle that we have is that when we change either the
height or width, the other one remains the same. Since the square is a rectangle (in the
mathematical sense) its behaviour is different.
//The solution would be to add an abstraction (shape abstract class)



https://medium.com/@alex24dutertre/square-rectangle-and-the-liskov-substitution-principle-e
e1eb8433106
```

Design by Contract technique for LSP

An advantage we did not mention in the beginning for this technique is that the class can satisfy the LSP under the conditions that any derived class that redefines a base class method must provide a 'weaker' precondition and a 'stronger' post-condition than the base class.

```
class ServerA
{
    //P
    public virtual void Method() { }
    //Q
}

class ServerB : ServerA
{
    //P'
    public override void Method() { }
    //Q'
}

//P => P' (true)
//Q' => Q (true)
//Under these 'legacy' conditions, LSP is correct. Why?

class Client
{
    ServerA _server;

    public void Job()
    {
        /// here P is true!!! Thus P => P' follows and P' true
        _server.Method(); // Here, why does the code remain functional for ServerB??
        //P' true, according to Design by Contract, so Q' it true! Which leads to Q' => Q;
        // here Q is true
    }
}
```

Class invariants can be established. What is a correct class? Invariants for default builders.

I is the invariant of a class.

Method m has precondition P and post condition Q, then I & P must be true before execution and I & Q after execution;

Interface Segregation Principle (ISP)

The principle states that software entities must depend on abstractions (abstract classes) and interfaces. Situations may arise where a class depends on an interface with very many methods ("fat interface")

```
public interface IFat
{
    void Method1();
    void Method2();
    void Method3();
    void Method4();
}

class Client12
{
    IFat _fat;

    public void Method()
    {
        _fat.Method1();
        _fat.Method2();
    }
}

class Fat1 : IFat
{
    //all implemented methods
}
```

The IFat interface will be implemented by certain classes. These will implement most if not all methods. Indirectly, the Client12 class depends on all methods from the IFat interface, even if it only uses Method1 and Method2.

If only Method3 or Method4 is changed, the code of Client12 class is also affected. Furthermore, if an interface contains many methods, this certainly leads to violation of the SRP principle.

A class must not depend on methods it does not use (through an interface). The right solution would be to split the interfaces.

```
public interface IFat12
{
    void Method1();
    void Method2();
}

public interface IFat34
{
    void Method3();
    void Method4();
}

public interface IFat: IFat12, IFat34 {  }
```

```
class Client12
{
    IFat12 _fat;
}
```

Dependency Inversion Principle (DIP)

The Dependency Inversion Principle is a general design guideline which recommends that classes should only have direct relationships with high-level abstractions.

The Dependency Inversion Principle (DIP) emphasizes decoupling and abstraction. The principle consists of two core concepts: high-level modules should not depend on low-level modules, and both should depend on abstractions. This inverted dependency relationship promotes flexibility, testability, and maintainability.

Another way to look at it would be interface-oriented programming, not implementation-oriented.

```
class ConcreteClass
{
    public void Job() { }
}

class Client
{
    ConcreteClass s;

    public void Task()
    {
        ConcreteClass c = new ConcreteClass(); //incorrect
        c.Job();
    }
}
```

```
//The proper way to implement DIP
public interface IJob
{
    void Job();
}

class ConcreteClass1 : IJob
{
    public void Job() {  }
}

class ConcreteClass2 : IJob
{
    public void Job() {  }
}
```

```
class Client
{
    IJob _job;

    public Client(IJob job)
    {
        _job = job;
    }

    public void Task()
    {
        _job.Job();
    }
}
```

Dependency Injection Technique (injecting dependencies)

We say that a class is dependent on another class if this class appears as a reference in the initial class (there is an association relationship between classes).

The DIP principle says that association should be done at the highest level (general classes, interfaces).

Dependency Injection is a technique by which various concrete class objects are used in a Client class without their knowledge. It allows you to avoid hardcoding dependencies in your classes, making your code more flexible and adaptable to changes.

Injection techniques

```
// 1. Constructor Injection
class Client
{
    IJob _service;

    public Client(IJob service)
    {
        _service = service;
    }

    public void Task()
    {
        _service.Job();
    }
}

var client = new Client(new ConcreteClient());
//client class must be instantiated for each new injection!!!
client = new Client(new ConcreteClient1());
```

```
// 2. Methods Injection (getter/setter)
class Client
{
    IJob _service;

    public IJob Service
    {
      get { return _service; }
      set { _service = value; }
    }

    public Client(IJob service)
    {
        _service = service;
    }

    public void Task()
    {
        _service.Job();
    }
}

var client = new Client();
client.Service = new ConcreteClient(); //we inject the dependency in the method.

// 3. Variable Injection to the method
class Client
{
    public void Task(IJob service)
    {
        service.Job();
    }
}
```

Case study

It is desired to design a multi-panel system (wizard type).
Use involves going through a set of panels.
There is an initial and a final panel. Several panels can be final panels.
The running of a sequence of panels will be called a 'session'. The path is the session.
Any session starts with an initial panel, ends with the final panel.
The panels are laid out in a graph-like structure.
Each node in the graph represents a panel.
At any given time, within the application, only one panel is the initial panel (marked as such).
Whenever a panel needs to be executed, this implies the following actions:

- display panel; (Display)
- reading data provided by the user; (Read)
- validation of read data (IsValid)
    - If valid, will be processed. (Process)
    - If invalid, an error message is displayed and the panel is re-displayed. (Message)

A session runs through the panels and executes them in succession.
At each step, after execution, the user also provides an option of continuation (choice).

When designing the solution, the following conditions will be considered:

- Extensibility;
- Flexibility;
- Most general solution;
- If new panels are desired in the system (new transitions between panels), it can be done without great cost (current code to be as little affected as possible)
- Graph structure to be efficient. (type of data selected, few resources consumed)

Design templates (Design Patterns)

Template

General design solution, usually reusable, that solves a particular design scenario and can be applied to a wide range of applications.

Typically, the pattern solves design problems of 3 categories:

- Behavioral - behaviors of objects in different states;
- Structural - relationships between object classes;
- Creational - how you create/construct the objects that participate in the pattern.

A design pattern is generally described by the following scheme by the GoF: Gang of Four's [, specifically E. Gamma] catalogue of design patterns:

1. **Name** - A suggestive name (State, Template Method, Singleton, Composite...);
2. **Intent** - What is the main objective of the template? (What design problem does it solve?);
3. **Motivation** - A real problem to be solved is presented in which the template was used;
4. **Structure** - What are the classes/interfaces participating in the template and the relationships between them? ;
5. **Participants** - Classes participating in the pattern;
6. **Applicability** - in which situations can the pattern be used? ;
7. **Advantages and Disadvantages**;
8. **Collaborations** - How objects communicate with each other via messages to implement the template;
9. **Implementation** - Indicate implementation options;

State Behavioral Pattern

1. **Name** - State Design Pattern;
2. **Intent** - Change the state of an object and its behavior without concrete state dependencies;
3. **Motivation** – For multi view panel application (a panel represents a state, a session represents the object that changes state; the change of state coincides with the display of the next panel; the change of behavior coincides with different executions of the current panel);
4. **Structure**
   - Context class (Session class) is needed for objects that go through states/behaviors differently.
   - The Request() method is the behavior that changes when the state changes (Execute from Session). (state.Handle ⇔ currentState.Execute: behavior in the "state".);
5. **Participants** - Abstract class State is the abstraction of the states and behaviors of the Context object (Session). Some particular, concrete states/behaviors are ConcreteStateA, ConcreteStateB, etc. ;
6. **Applicability** - When an object goes through many states and behaves differently in different states. For extensibility, maintenance. When a method implements extremely complex switch or if/else instructions, which generally depend on general enumeration.

```
//Each case becomes a state;
switch(enum Value)
{
    case 1: process1;
    case 2: process2;
    ...
    case N: processN;
}
```

```
abstract class State
{
    public abstract void processing();
}

class State1: State
{
    public void process()
    {
        //implement processing1
    }
}

if(state1)
    process1();
else if(state2)
    process2();
```

7. **Advantages -** Extensibility (OCP for Context class);
   Disadvantages – complexity [for small enough and simple classes];
8. **Collaborations**
   - Client calls the Request method of the Context class;
   - The Context object appeals through the reference "states", using Handle method/s. By polymorphism, depending on the real object that prefixes the call (current state), the version from a "ConcreteStateN" class will be executed;
   - The change of state is simulated and different behavior occurs.
9. **Implementation** [variants]:
   a) All possible states are created at application initialization and stored in the Context; analogue and transitions;
   b) The context class only holds reference to the current state, concrete states are created dynamically, as the condition changes. Transitions are no longer stored in memory, they are implemented "in code", in classes derived from the States.

```
class Context
{
    public State CurrentState { get; set; }

    public void Request()
    {
        CurrentState.Handle();
        CurrentState.NextState(this);
    }
}

abstract class State()
{
    abstract void NextState(Context c);
}
```

```
class StateA: State
{
    public void NextState(Context c)
    {
        if (condition)
        {
            c.CurrentState = new StateB();
        }
    }
}

//Advantage - the transitions are easy to see from the code; memory advantage.
//Disadvantage - derived "State" means classes are no longer independent.
```

Template Method Behavioral Pattern

1. **Intent** - The skeleton of an algorithm is known, but its steps are not known;
2. **Motivation -**

```
public abstract class State
    {
        public int Id { get; set; }
        public static int Choice { get; set; }
        public abstract void Display();
        public abstract void Read();
        public abstract bool IsFinal();
        public abstract string Message();
        public abstract bool IsCorrect();
        public abstract void Process();

        //method Template (algorithm skeleton)
        public void Execute()
        {
            var ok = false;

            do
            {
                Display();
                Read();
                ok = IsCorrect();
                if (!ok)
                {
                    Message();
                }
            }
            while (!ok);

            Process();
        }
    }
```

3. **Structure** - An abstract class and at least one concrete implementation;
4. **Participants**
    - AbstractClass - contains the schema of the algorithm in a concrete method. The steps of the algorithm (PrimitiveOperation1, PrimitiveOperation2) appear as abstract methods.
    - ConcreteClass - implements the steps of the algorithm.
5. **Advantage**
    - Reuse of source code; (reuse of common behavior: the "execute" method is common to all panels)
    - Adherence to the Hollywood principle: "don't call us, we'll call you!": the correct principle of source code re-use.

```
class A
{
    public void m()
    {
        //the method code of the base class
        hook(); //"we'll call you!"
    }

    public virtual void hook() { }
}

class B: A
{
    public override void hook() {  }
}
```

Particular example of Template Method Pattern Application

```
//search in collections of T objects data from the database!!!
//reading method will be a template method!!!
//for select statements from the database
public abstract class DbReader<T, P>
{
    public string ConnectionString { get; set; }

    //template method
    public List<T> Get(P parameters)
    {
        var result = new List<T>();

        try
        {
            using (var connection = new SqlConnection(ConnectionString))
            {
                connection.Open();

                var param = GetParameters(parameters); //hook method

                using (var command = new SqlCommand(SqlText, connection))
                {
                    if(param.Count > 0)
                    {
                        command.Parameters.AddRange(param.ToArray());
                    }
                    var reader = command.ExecuteReader(); //result
                    result = MapAll(reader); //mapping a reader to an object of type T
                }
            }
            return result;
        }
        catch (Exception e)
        {
            return result;
        }
    }

    //template method
    private List<T> MapAll(IDataReader reader)
    {
        var result = new List<T>();

        while (reader.Read())
        {
            result.Add(Map(reader)); //maple current line
        }
        return result;
    }
    protected abstract T Map(IDataReader reader); //hook method

    public abstract string SqlText { get; }

    protected virtual List<IDataParameter> GetParameters(P parameters)
    {
        return new List<IDataParameter>();
    }
}

internal class UserDbReader: DbReader<User, UserSearchOption>
{
    public override string SqlText => "SELECT Username, Password FROM User WHERE IsAdmin = @p1";

    protected override User Map(IDataReader reader)
    {
        return new User()
        {
            UserName = (string)reader["Username"],
            Password = (string)reader["Password"]
        };
    }

    protected override List<IDataParameter> GetParameters(UserSearchOption parameters)
    {
        return new List<IDataParameter>()
        {
            new SqlParameter("@p1", parameters.IsAdmin)
        };
    }
}

internal class FlightDbReader: DbReader<Flight> {  }

class User {  }
```

Undo/Redo (multi-level) applications (Template Command)

For an application with a graphical user interface, we can implement n-level undo/redo mechanism. The application goes through a sequence of states caused by a sequence of commands performed:

- S0=>S1=>S2...Sn=>Sn+1
- C0=>C1=>C2...Cn=>Cn+1

Upon executing "Undo" - undo the effect of the Cn+1 command; the application will switch to the Sn state.
Note - not every order is cancellable! We decide which orders can be cancelled/refunded.

Project 1 - implementation of a multi-panel application (wizard)

Implement a multi-panel application for a flight agency (Tarom).
It is desired to design a multi-panel system (wizard type).

Use involves going through a set of panels.
There is an initial and a final panel.
The running of a sequence of panels will be called a 'session'.

The panels are laid out in a graph-like structure.
Each node in the graph represents a panel.
The path is the session.
Any session starts with an initial panel, ends with the final panel.
At any given time, within the application, only one panel is the initial panel (marked as such).
Several panels can be final panels.

Users are considered:

- Admin
- Client

Services to be implemented:

- Authentication (login) - P1 (initial)
- Registration (register - create account -admin, client) - P2
- Add flight (admin) – P3
- Modify flight(admin) – P4
- Delete flight (admin) – P5
- Reserve flight (admin, client) – P6
- Purchase/pay flight(admin, client) – P7
- Search flight(admin, client) – P8
- Flight list – P9
- Reserve seat on a flight – P10
- Help(info) – P11
- Exit - P12 (final)

Implementation solutions

1. **Functional decomposition (procedural programming)**

    Assume we don't know object programming.
    Each facility we turn into a function.
    Do I have to run a session? Implement an 'ExecuteSession' function.
    Do I have to run a panel? Implement an 'ExecutePanel' function with the following functions:

    - Display (panel display)
    - Read data
    - IsCorrect (date validation)
    - Message (error message)
    - Process (processing of correctly provided data)

    Disadvantages of this approach - no extensibility. Each function has a panelId parameter in implementation (and is controlled by e.g. a switch statement).

```
void ExecutePanel(int panelId, out int choice, out data)
{
    do{
        Display(panelId);
        data = Read(panelId, out choice);
        ok = IsCorrect(data, panelId);
        if (!ok)
        {
            Message(panelId);
        }
    }
    while(!ok);

    Process(data, panelId);
}

int IntialPanel()
{
    return initialPanelId;
}

bool IsFinal(int panelId)
{
    switch(panelId)
    {

    }
}

//in the automata, return the following panel depending on the current panel and the choice of
continuation
//transitions between panels (transition function)
//function sets the next panel from the current one
int Transition(int panelId, int choice)
{  return //..... }

//running a session:
void ExecuteSession()
{
    //get initial panel
    int panelId = InitialPanel();
    //bool final = IsFinal(panelId);
    int choice;
    do{
        ExecutePanel(panelId, out choice);
        //obtain the following panel id
        panelId = Transition(panelId, choice);
    }
    while(!IsFinal(panelId));
}
```

2. Object-oriented design (Design Patterns: State, Template Method)

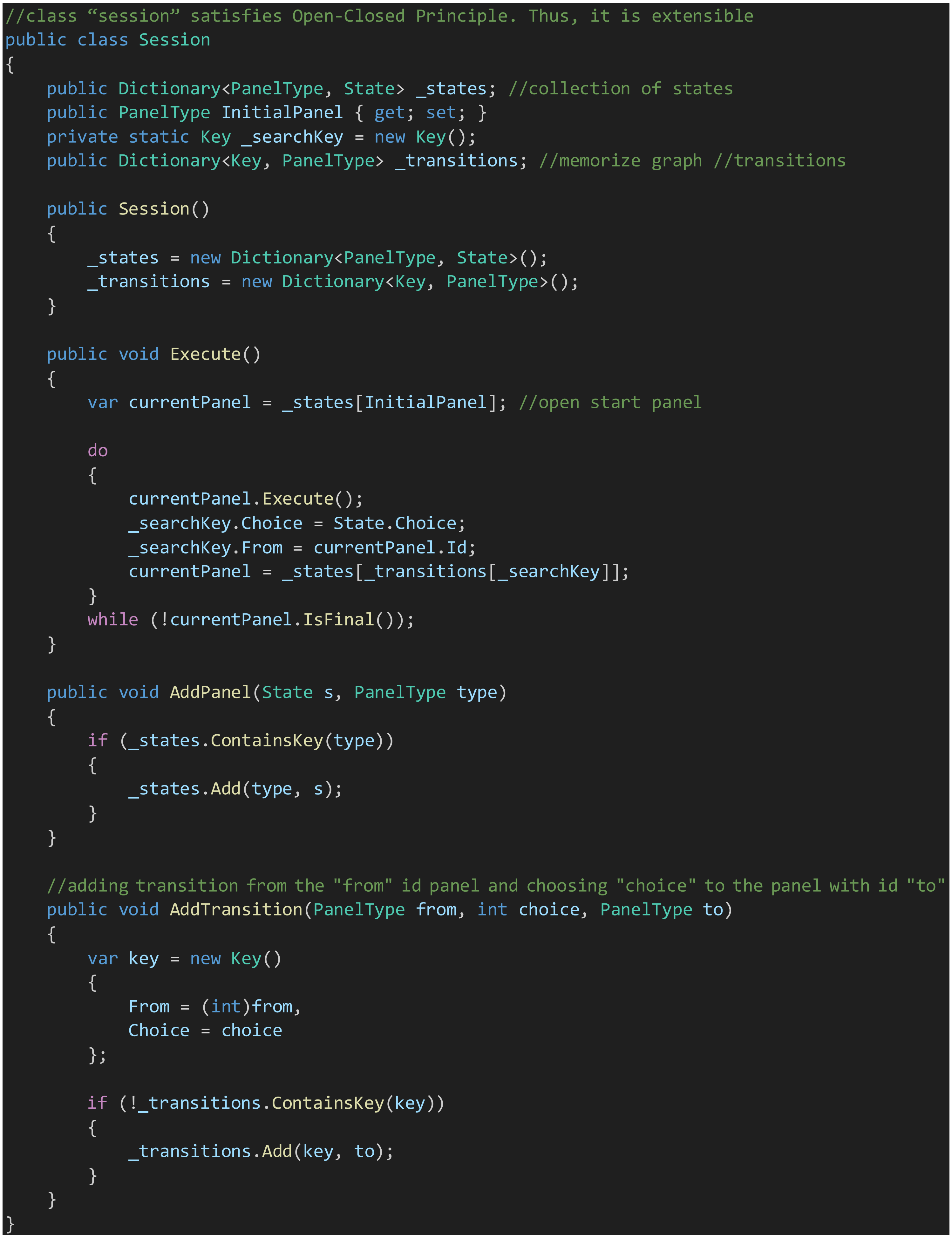
Object-oriented solution will be implementing the State design template.

Displaying and processing a current panel forms the current state of the application. Therefore, we abstract the notion of panel by a class "State". For each panel, a different State class is implemented.

Similarly, we abstract the notion of "session".

```
//abstracting the notion of a panel via an abstract class State
public abstract class State
{
    public int Id { get; set; }
    public static int Choice { get; set; }

    //Template method (algorithm skeleton)
    public abstract void Display();
    public abstract void Read();
    public abstract bool IsFinal();
    public abstract string Message();
    public abstract bool IsCorrect();
    public abstract void Process();

    //we know how to run current panel
    public void Execute()
    {
        var ok = false;

        do
        {
            Display();
            Read();
            ok = IsCorrect();
            if (!ok)
            {
                Message();
            }
        }
        while (!ok);

        Process();
    }
}
```

```
class LoginPanel : State
{
    UserModel _user;

    public LoginPanel() { Id = 1; }

    public override void Display()
    {
        Console.WriteLine("Display Login");//display panel
    }

    public override bool IsCorrect()
    {
        //search _user database
        return false;
    }

    public override bool IsFinal()
    {
        return false;
    }

    public override string Message()
    {
        return "wrong username or password";
    }

    public override void Process() {  }

    public override void Read()
    {
        _user = new UserModel();
        //user identification (from database or console)
    }
}
```



```
//class “session” satisfies Open-Closed Principle. Thus, it is extensible
public class Session
{
    public Dictionary<PanelType, State> _states; //collection of states
    public PanelType InitialPanel { get; set; }
    private static Key _searchKey = new Key();
    public Dictionary<Key, PanelType> _transitions; //memorize graph //transitions

    public Session()
    {
        _states = new Dictionary<PanelType, State>();
        _transitions = new Dictionary<Key, PanelType>();
    }

    public void Execute()
    {
        var currentPanel = _states[InitialPanel]; //open start panel

        do
        {
            currentPanel.Execute();
            _searchKey.Choice = State.Choice;
            _searchKey.From = currentPanel.Id;
            currentPanel = _states[_transitions[_searchKey]];
        }
        while (!currentPanel.IsFinal());
    }

    public void AddPanel(State s, PanelType type)
    {
        if (_states.ContainsKey(type))
        {
            _states.Add(type, s);
        }
    }

    //adding transition from the "from" id panel and choosing "choice" to the panel with id "to"
    public void AddTransition(PanelType from, int choice, PanelType to)
    {
        var key = new Key()
        {
            From = (int)from,
            Choice = choice
        };

        if (!_transitions.ContainsKey(key))
        {
            _transitions.Add(key, to);
        }
    }
}
```

```
public class Key
{
    public int From { get; set; } //starting point
    public int Choice { get; set; } //choice for continuation

    //implementation
    public override int GetHashCode()
    {
        return 2 * From.GetHashCode() + 3 * Choice.GetHashCode();
    }

    public override bool Equals(object obj)
    {
        return base.Equals(obj);
    }
}

public enum PanelType
{
    Login = 1,
    //...
}
```

Database with a minimum number of tables:

- User (userid, username, password, tip - client, admin)
- Flight (flightid, from, to, date, price, ...)
- FlightSeat(seatid, flightId, isFree)
- FlightReservation(userId, flightId, date, expirationDate)

Project structure in solution

1. Flight.Model (class library - *.dll)
2. Main classes (models) (State, Session, LoginPanel, UserModel, FlightModel, FlightReservation, ...), enums
3. Flight.DataAccess - access to the database (repository)

```
public interface IRepository
{
    get();
    create();
    update();
    delete();
}

class UserRepository : IRepository {  }
class FlightRepository : IRepository {  }
```

4. Flight.UI
   - Console
   - windows forms (recommended)

Project 2 - Laboratory

Build a .NET application for drawing / resizing / moving graphic objects.
The graphic primitives, that should be realized are line, circle, rectangle, squares and groups of graphic primitives.

Directions

1. All graphic objects will be treated uniformly by means of a general Shape class.
2. Implement a GraphicTool object that represents a container of objects of type Shape.
3. The drawing will be done in a PictureBox control.
4. A graphic context is obtained - Graphics g = control.CreateGraphics(). The Paint event will be redefined for redraw if the main window is: minimized, maximized, resized...
5. MyPaint.Editor (UI)
6. Ability to save currently drawn objects to disk.

```
abstract class Shape
{
    public Point2D Origin { get; set; }
    public string NameKey { get; set; }
    public abstract void Draw();
    public abstract void Resize(/*scale factor*/);
    public abstract void MoveTo(/*new point2D*/);
    public virtual void Add( Shape shape) { }
    public virtual void Remove(Shape shape) { }
};

public class Point2D
{
    public int X { get; set; }
    public int Y { get; set; }
}

class Line: Shape
{
    public override void Draw() {  }
    public override void MoveTo() {  }
    public override void Resize() {  }
}
```

```
 class Picture: Shape
 {
     private List<Shape> _shapes;

     public Picture() { _shapes = new List<Shape>(); }

     public override void Draw()
     {
         foreach (var item in _shapes)
         {
             item.Draw();
         }
     }

     public override void MoveTo() {  }
     public override void Resize() {  }
     public override void Add(Shape shape) { _shapes.Add(shape); }
     public override void Remove(Shape shape) { _shapes.Remove(shape); }
 }

class GraphicTool
{
    public List<Shape> Shapes { get; private set; }

    public GraphicTool() { Shapes = new List<Shape>(); }

    public void Add(Shape shape) { Shapes.Add(shape); }

    public void Remove(Shape shape) { Shapes.Remove(shape); }

    public void DrawAll()
    {
        foreach(var shape in Shapes)
        {
            shape.Draw();
        }
    }
}
```