38. What is the Strategy design pattern?

Brief summary: The Strategy Design pattern is a pattern that allows us to define a family of algorithms to perform some tasks. The concrete strategy can be chosen at runtime.

The Strategy design pattern is a pattern that allows us to define a family of algorithms that perform some tasks. The concrete strategy can be chosen at runtime.

Let's imagine we implement a platform selling video games. Here is the Game type and some games we currently have in our database:

```
record Game(
    string Title,
    decimal Price,
    decimal Rating,
    DateTime ReleaseDate,
    bool IsAvailable);
```

```
var games = new List<Game>
{
    new Game(
        "Stardew Valley", 19.99m, 98,
        new DateTime(2016, 2, 26), true),
    new Game(
        "Red Dead Redemption II", 60m, 92,
        new DateTime(2018, 10, 26), true),
    new Game(
        "Spiritfarer", 25m, 95,
        new DateTime(2020, 8, 18), false),
    new Game(
        "Heroes III", 10m, 82,
        new DateTime(1999, 3, 3), false),
    new Game(
        "God of War", 60m, 97,
        new DateTime(2018, 4, 20), true),

};
```

At the first version of the platform, the user can only search for games by their title. By definition, we don't want to show games that are not available. The code to implement this behavior could look like this:

```
IEnumerable<Game> FindByTitle(
    IEnumerable<Game> games,
    string searchWord)
{
    return games.Where(g => g.isAvailable &&
    g.Title.Contains(searchWord));
}
```

Great. After some time our platform evolves, and we are asked to add some pre-defined filters to the search options. The first one is "Best games" which returns games with a rating of 95 or more:

```
IEnumerable<Game> FindBestGames(
    IEnumerable<Game> games)
{
    return games.Where(g => g.isAvailable &&
    g.Rating > 95);
}
```

All right. This method is quite similar to the one that we had before, but let's not jump to refactoring yet - we perhaps have better things to do. But soon after, we are asked to add other predefined filters: "Games of this year" showing games released in the current year, and "Best deals" finding games with prices below 25$.

```
IEnumerable<Game> FindGamesOfThisYear(
    IEnumerable<Game> games)
{
    return games.Where(g => g.IsAvailable &&
    g.ReleaseDate.Year == DateTime.Now.Year);
}

IEnumerable<Game> FindBestDeals(
    IEnumerable<Game> games)
{
    return games.Where(g => g.IsAvailable &&
    g.Price < 25);
}
```

Well… this starts to look unmanageable. All those methods are almost identical, and the code is duplicated. Before we start refactoring, let's see how this code could be used:

```
var selectedOption = FilteringType.BestGames;
var searchWord = "Red";
IEnumerable<Game> filteredGames = null;
switch (selectedOption)
{
    case FilteringType.ByTitle:
        filteredGames = FindByTitle(games, searchWord);
        break;
    case FilteringType.BestGames:
        filteredGames = FindBestGames(games);
        break;
    case FilteringType.GamesOfThisYear:
        filteredGames = FindGamesOfThisYear(games);
        break;
    case FilteringType.BestDeals:
        filteredGames = FindBestDeals(games);
        break;
}
```

All right. This doesn't look good. It's high time to introduce the Strategy design pattern. According to this pattern, we should be able to define a family of algorithms that can be injected into some other code at runtime. In our case, the

family of algorithms will contain all predicate methods, that decide whether a game should be included in filtered results or not:

```
Func<Game,bool> SelectStrategy(
    FilteringType selectedOption, string searchWord)
{
    switch (selectedOption)
    {
        case FilteringType.ByTitle:
            return game => game.Title.Contains(searchWord);
        case FilteringType.BestGames:
            return game => game.Rating > 95;
        case FilteringType.GamesOfThisYear:
            return game => game.ReleaseDate.Year ==
                DateTime.Now.Year;
        case FilteringType.BestDeals:
            return game => game.Price < 25;
        default:
            throw new ArgumentException("Invalid option");
    }
}
```

Each strategy is an algorithm enclosed in executable code. In this case, I return it as a Func, but returning it as an object implementing an interface would also be valid and in line with this design pattern. Remember, after all, a Func is like an interface with a single method.

We can now plug this strategy into code doing actual filtering:

```
IEnumerable<Game> FindBy(
    Func<Game, bool> strategy,
    IEnumerable<Game> games)
{
    return games.Where(g => g.IsAvailable && strategy(g));
}
```

And this is how it all can be used together:

```
var strategy = SelectStrategy(selectedOption, searchWord);

var filteredGames = FindBy(strategy, games);
foreach (var game in filteredGames)
{
    Console.WriteLine(game);
}
```

As you can see, this is quite simple. You probably used this pattern before, even if you did not know it. Everywhere where you pass some interchangeable code as a parameter - especially a Func or various objects representing a single interface - you were using the Strategy design pattern.

Using this pattern allowed us to remove code duplications. We have now one clear place where the algorithms are defined. Adding a new way of filtering would now only mean that we must add another **case** to the **switch** in which we define methods of filtering. The generic filtering algorithm (defined in the FindBy method) and specific subfilters are now separated. This makes the code simpler and more easily testable.

So, to summarize: the Strategy design pattern is a pattern that allows us to define a family of algorithms that perform some tasks. The concrete strategy can be chosen at runtime.

Bonus questions:

- "**What are the benefits of using the Strategy design pattern?**"
  It helps to reduce code duplications, makes the code cleaner and more easily testable. It separates the code that needs to be changed often (the particular strategy) from the code that doesn't change that much (the code using the strategy).