UNIVERSITY OF WESTMINSTER

Enterprise Application Development

Lecture 10 – Design Patterns (in C# and .NET)

Design Patterns

- Generalized, reusable solutions to common design issues in software engineering
- In the case of object-oriented programming, design patterns are generally aimed at solving the problems of object generation and interaction
- They give generalised solutions in the form of templates that may be applied to real-world problems

Design Pattern Groups

- Creational
  - Provide ways to instantiate single objects or groups of related objects
- Structural
  - Provide a manner to define relationships between classes or objects.
- Behavioural
  - Define manners of communication between classes and objects

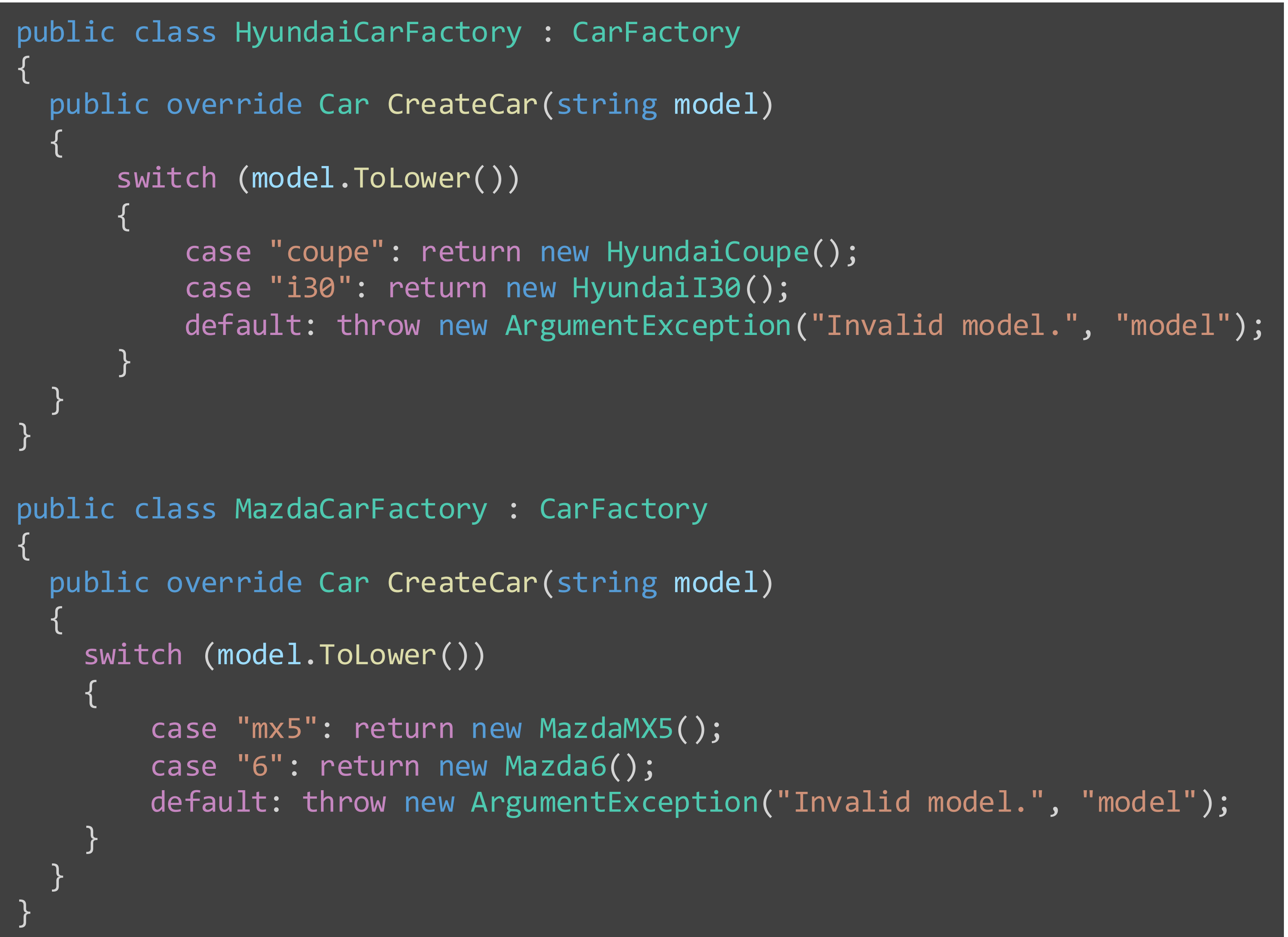
Creational

- Abstract Factory
- Builder
- Factory
- Prototype
- Singleton

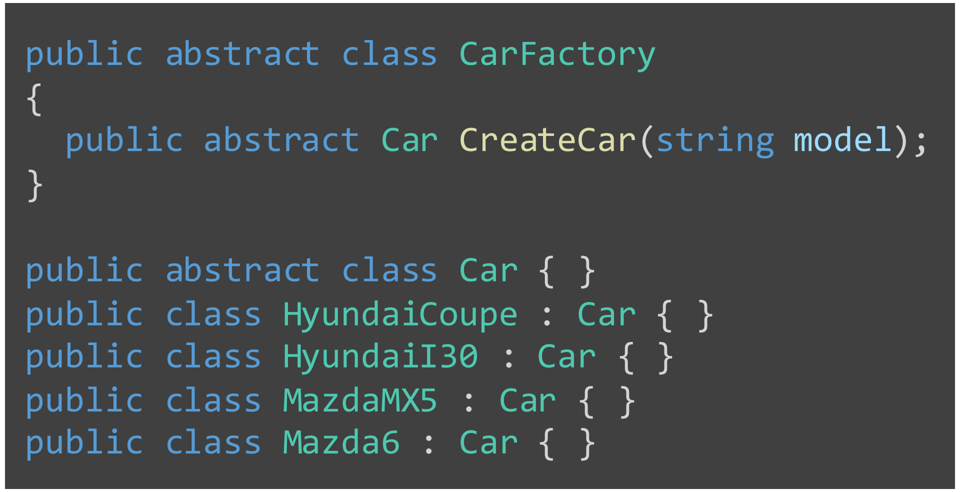
Creational: Factory Pattern

- This is a creational pattern as it is used to control class instantiation
- The factory pattern is used to replace class constructors, abstracting the process of object generation so that the type of the object instantiated can be determined at run-time.
- Example
  - Dynamically generate UI controls based on user input (theme, colours, spacing, etc.)

Factory - Implementation



```
public class HyundaiCarFactory : CarFactory
{
  public override Car CreateCar(string model)
  {
      switch (model.ToLower())
      {
          case "coupe": return new HyundaiCoupe();
          case "i30": return new HyundaiI30();
          default: throw new ArgumentException("Invalid model.", "model");
      }
  }
}

public class MazdaCarFactory : CarFactory
{
  public override Car CreateCar(string model)
  {
    switch (model.ToLower())
    {
        case "mx5": return new MazdaMX5();
        case "6": return new Mazda6();
        default: throw new ArgumentException("Invalid model.", "model");
    }
  }
}
```



```
public abstract class CarFactory
{
  public abstract Car CreateCar(string model);
}

public abstract class Car { }
public class HyundaiCoupe : Car { }
public class HyundaiI30 : Car { }
public class MazdaMX5 : Car { }
public class Mazda6 : Car { }
```

Factory - Implementation

```
CarFactory hyundai = new HyundaiCarFactory();
Car coupe = hyundai.CreateCar("coupe");
Console.WriteLine(coupe.GetType());   // Outputs "HyundaiCoupe"
```

```
CarFactory mazda = new MazdaCarFactory();
Car mx5 = mazda.CreateCar("mx5");
Console.WriteLine(mx5.GetType());   // Outputs "MazdaMX5"
```

Structural

- Adapter
- Bridge
- Composite
- Decorator
- Facade
- Flyweight
- Proxy

Structural: Facade

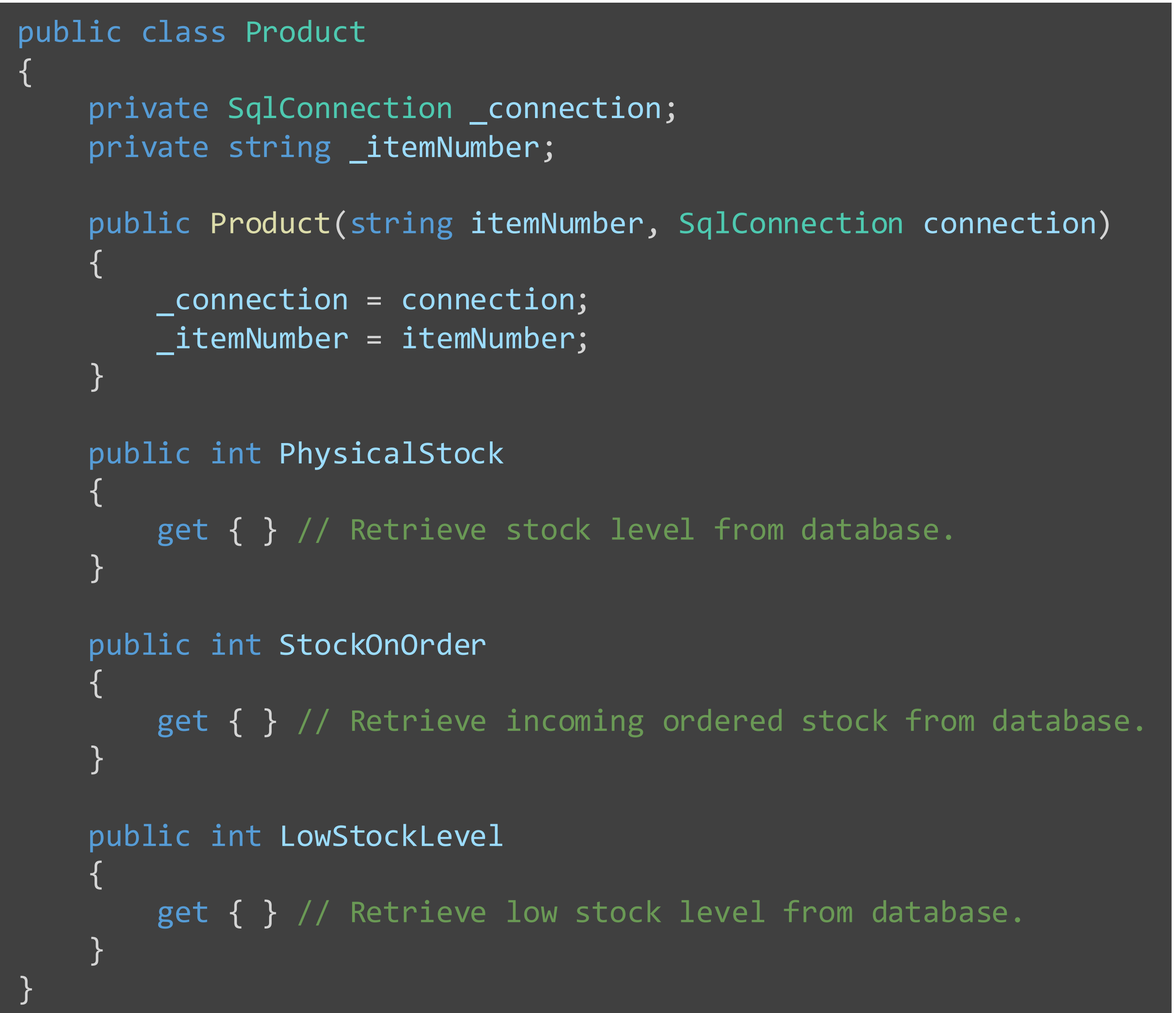
- The facade pattern is a design pattern that is used to simplify access to functionality in complex or poorly designed subsystems

- The facade class provides a simple, single-class interface that hides the implementation details of the underlying code

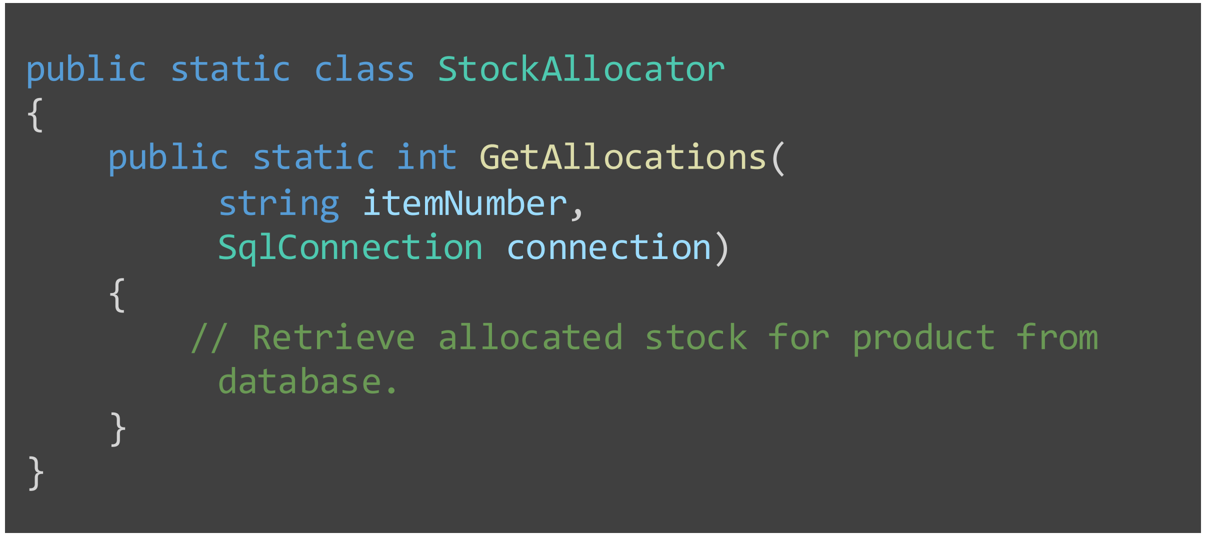
- It is particularly useful when wrapping subsystems that cannot be refactored because the source code is unavailable, or the existing interface is widely used

UNIVERSITY OF WESTMINSTER

Facade - Implementation



```
public class Product
{
    private SqlConnection _connection;
    private string _itemNumber;

    public Product(string itemNumber, SqlConnection connection)
    {
        _connection = connection;
        _itemNumber = itemNumber;
    }

    public int PhysicalStock
    {
        get { } // Retrieve stock level from database.
    }

    public int StockOnOrder
    {
        get { } // Retrieve incoming ordered stock from database.
    }

    public int LowStockLevel
    {
        get { } // Retrieve low stock level from database.
    }
}
```



```
public static class StockAllocator
{
    public static int GetAllocations(
         string itemNumber,
         SqlConnection connection)
    {
        // Retrieve allocated stock for product from
         database.
    }
}
```

Facade - Implementation

```
public class StockFacade
{
    public bool IsLowStock(string itemNumber)
    {
        SqlConnection conn = GetConnection(); // omitted for brevity

        Product product = new Product(itemNumber, conn);

        int physical = product.PhysicalStock;
        int onOrder = product.StockOnOrder;
        int lowStock = product.LowStockLevel;
        int allocations = StockAllocator.GetAllocations(itemNumber, conn);

        int available = physical + onOrder - allocations;
        return (available <= lowStock);
    }
}


static void Main(string[] args)
{
    StockFacade facade = new StockFacade();
    bool low = facade.IsLowStock("ABC123");
}
```

Behavioural

- Chain of Responsibility
- Command
- Interpreter
- Iterator
- Mediator
- Memento
- Observer
- State
- Strategy
- Template Method
- Visitor

Behavioral: Strategy

- The strategy pattern is a design pattern that allows a set of similar algorithms to be defined and encapsulated in their own classes
- The algorithm to be used for a particular purpose may then be selected at run-time according to your requirements
- This allows the behavior of a program to change dynamically according to configuration details or user preferences

Strategy - Implementation

```
public abstract class Storage
{
    public abstract int ReadData(string location);
}

public class Database : Storage
{
    public override string ReadData(string location)
    {
        // Read data from a database
    }
}

public class CSVFile : Storage
{
    public override string ReadData(string location)
    {
        // Read data from a CSV file
    }
}
```

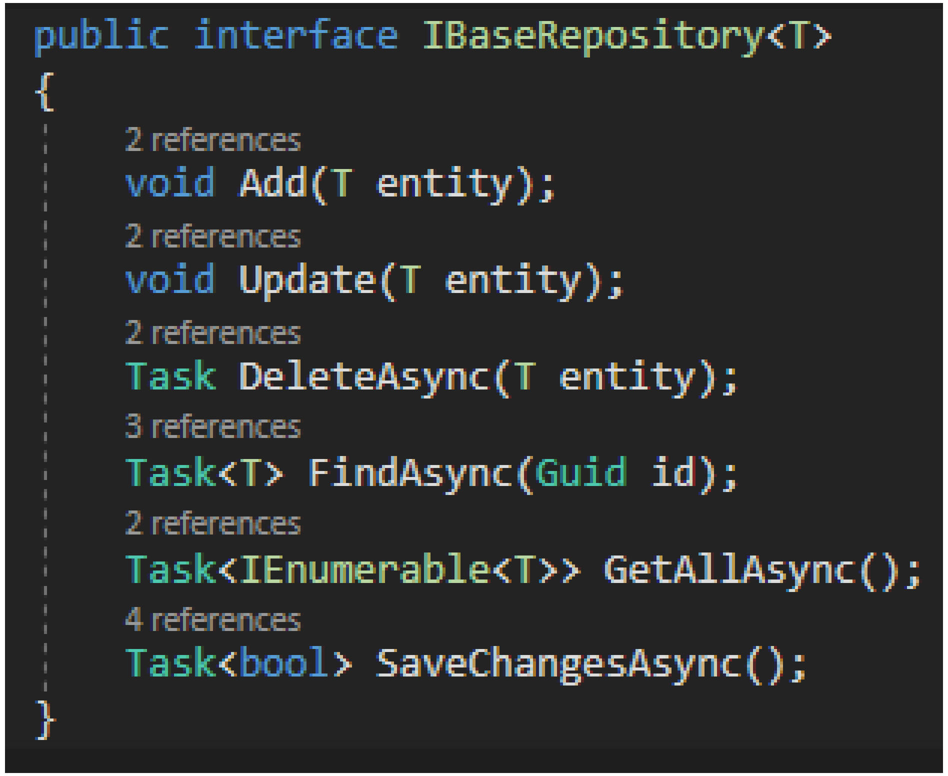
```
public class DataProvider
{
    public Storage StorageClient { get; set; }

    public void ShowStorageData(string location)
    {
        Console.WriteLine(StorageClient.ReadData(location));
    }
}

// Instantiation and method calling
DataProvider dataProvider = new DataProvider();

Console.WriteLine("Database");
dataProvider.StorageClient = new Database();
dataProvider.ShowStorageData("DB Connection string");

Console.Write("Woman");
dataProvider.StorageClient = new CSVFile();
dataProvider.ShowStorageData("file://location");
```

Repository pattern

- A repository performs the tasks of an intermediary between the domain model layers and data mapping
- It’s a popular design pattern mostly because it is fairly simple to implement and very helpful when we want to hide data store and retrieval logic
- It makes it easier to test your application logic

Repository pattern

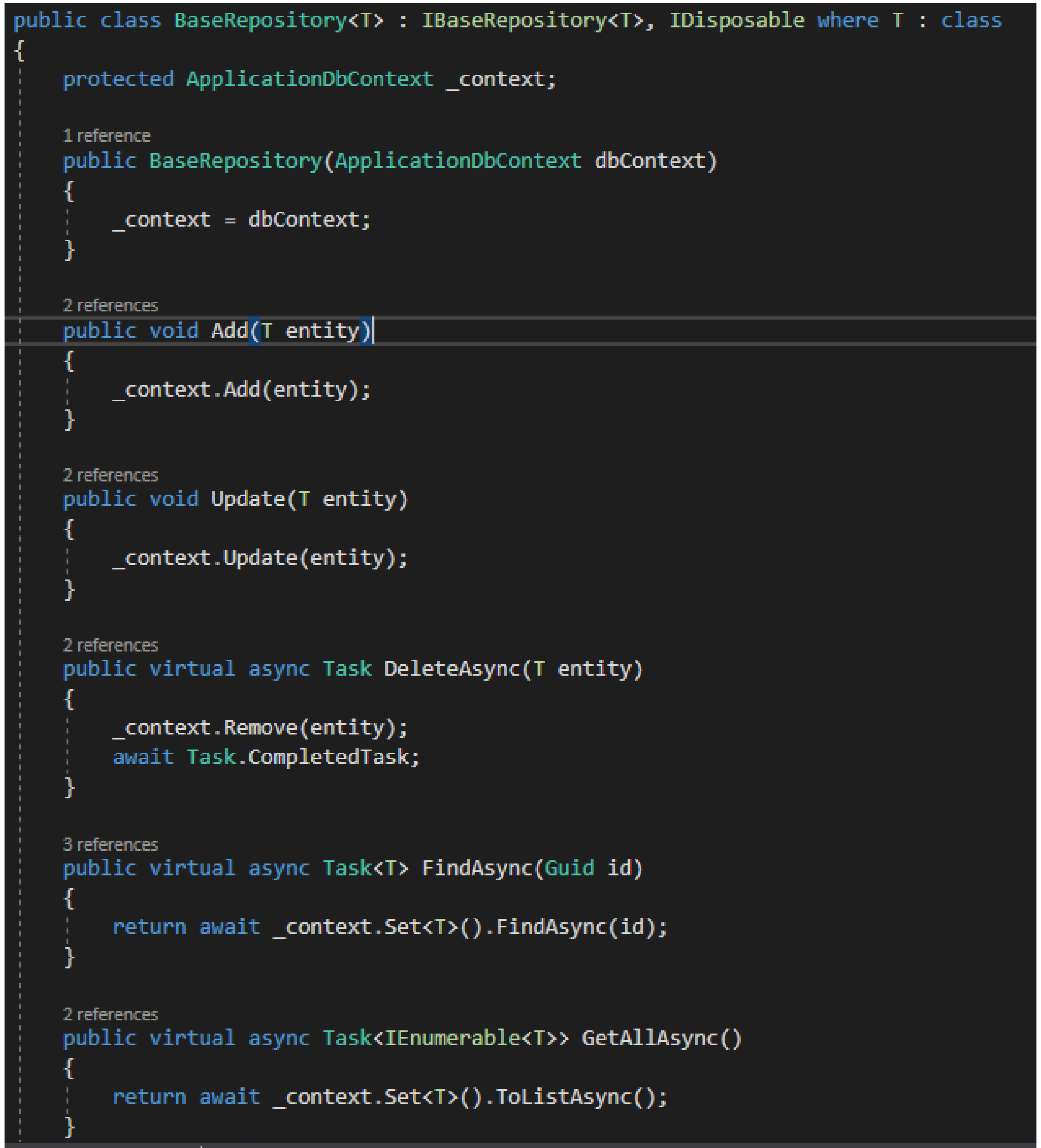
- Step 1: Define a base repository interface



```
public interface IBaseRepository<T>
{
    2 references
    void Add(T entity);
    2 references
    void Update(T entity);
    2 references
    Task DeleteAsync(T entity);
    3 references
    Task<T> FindAsync(Guid id);
    2 references
    Task<IEnumerable<T>> GetAllAsync();
    4 references
    Task<bool> SaveChangesAsync();
}
```

Repository pattern

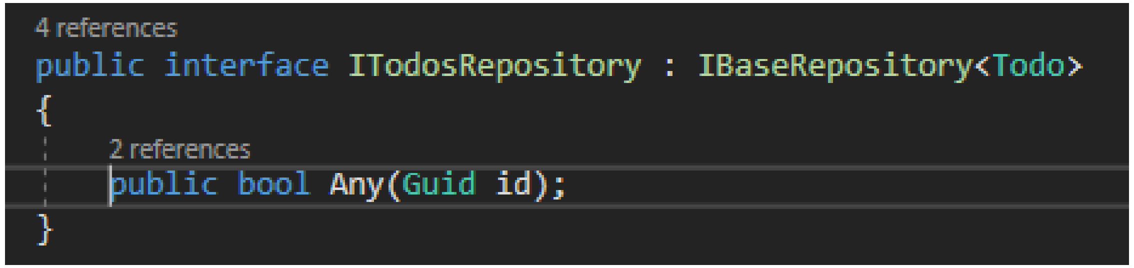
- Step 2: Implement the interface and provide DbContext as a dependency



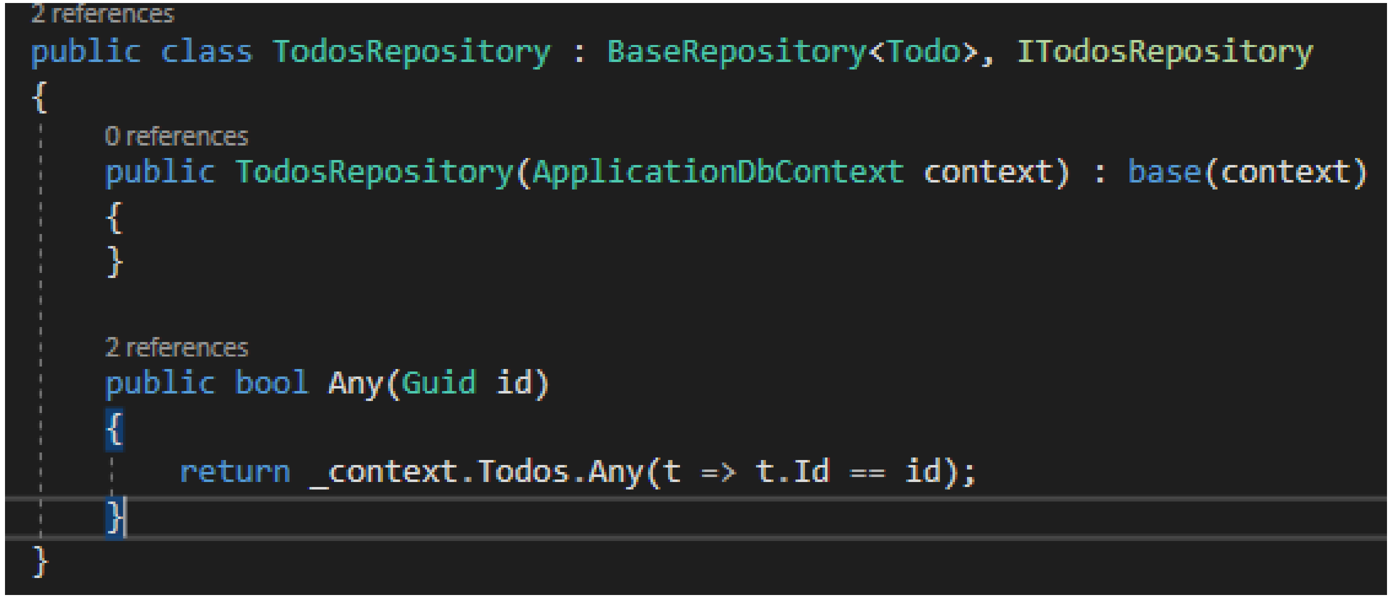
```
public class BaseRepository<T> : IBaseRepository<T>, IDisposable where T : class
{
    protected ApplicationDbContext _context;

    1 reference
    public BaseRepository(ApplicationDbContext dbContext)
    {
        _context = dbContext;
    }

    2 references
    public void Add(T entity)
    {
        _context.Add(entity);
    }

    2 references
    public void Update(T entity)
    {
        _context.Update(entity);
    }

    2 references
    public virtual async Task DeleteAsync(T entity)
    {
        _context.Remove(entity);
        await Task.CompletedTask;
    }

    3 references
    public virtual async Task<T> FindAsync(Guid id)
    {
        return await _context.Set<T>().FindAsync(id);
    }

    2 references
    public virtual async Task<IEnumerable<T>> GetAllAsync()
    {
        return await _context.Set<T>().ToListAsync();
    }
```

Repository pattern

- Step 3 (optional): We can create for specific models (entities) separate interface and repository which will extend the base repository



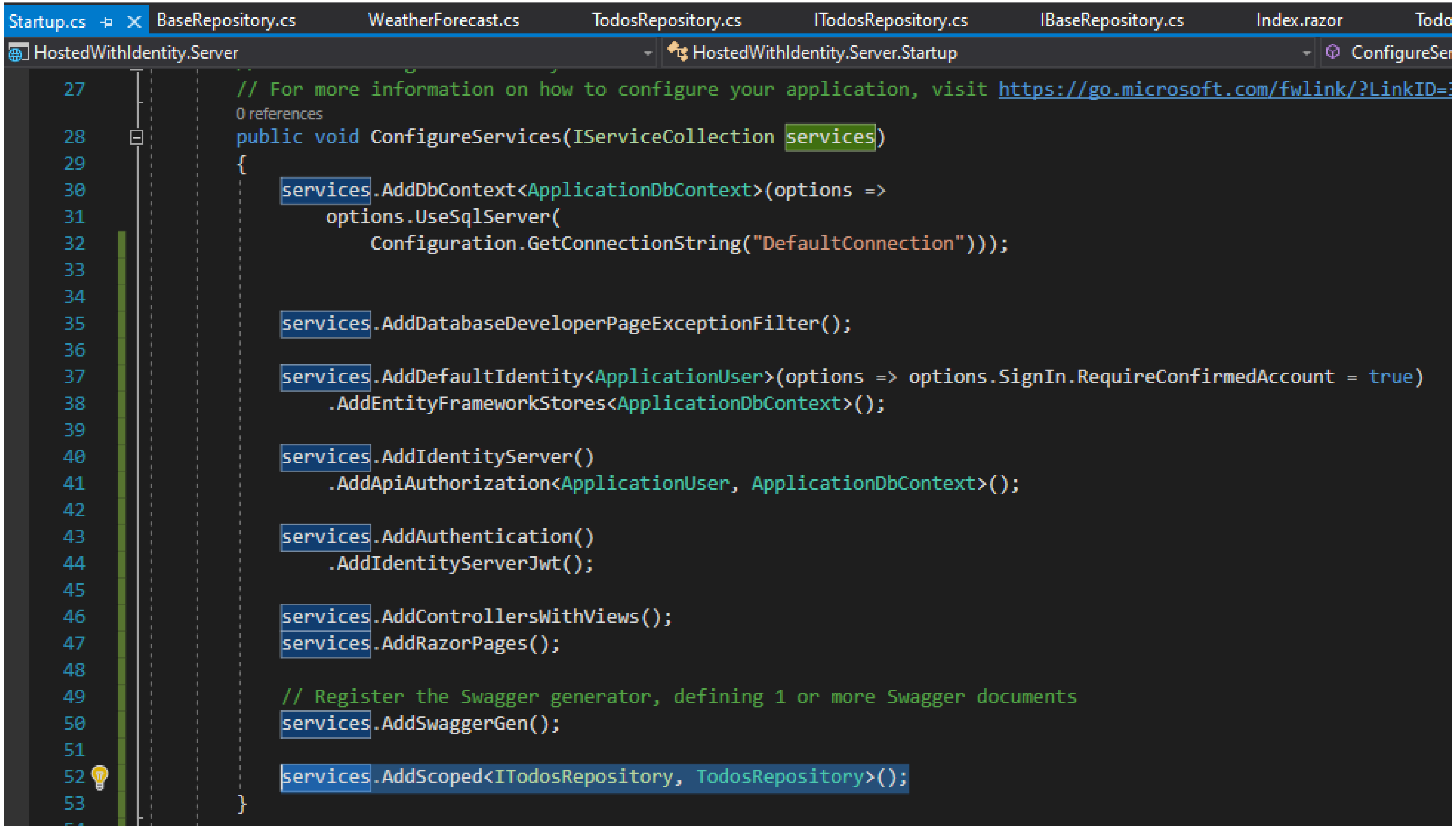
```
4 references
public interface ITodosRepository : IBaseRepository<Todo>
{
    2 references
    public bool Any(Guid id);
}
```



```
2 references
public class TodosRepository : BaseRepository<Todo>, ITodosRepository
{
    0 references
    public TodosRepository(ApplicationDbContext context) : base(context)
    {
    }

    2 references
    public bool Any(Guid id)
    {
        return _context.Todos.Any(t => t.Id == id);
    }
}
```

Repository pattern

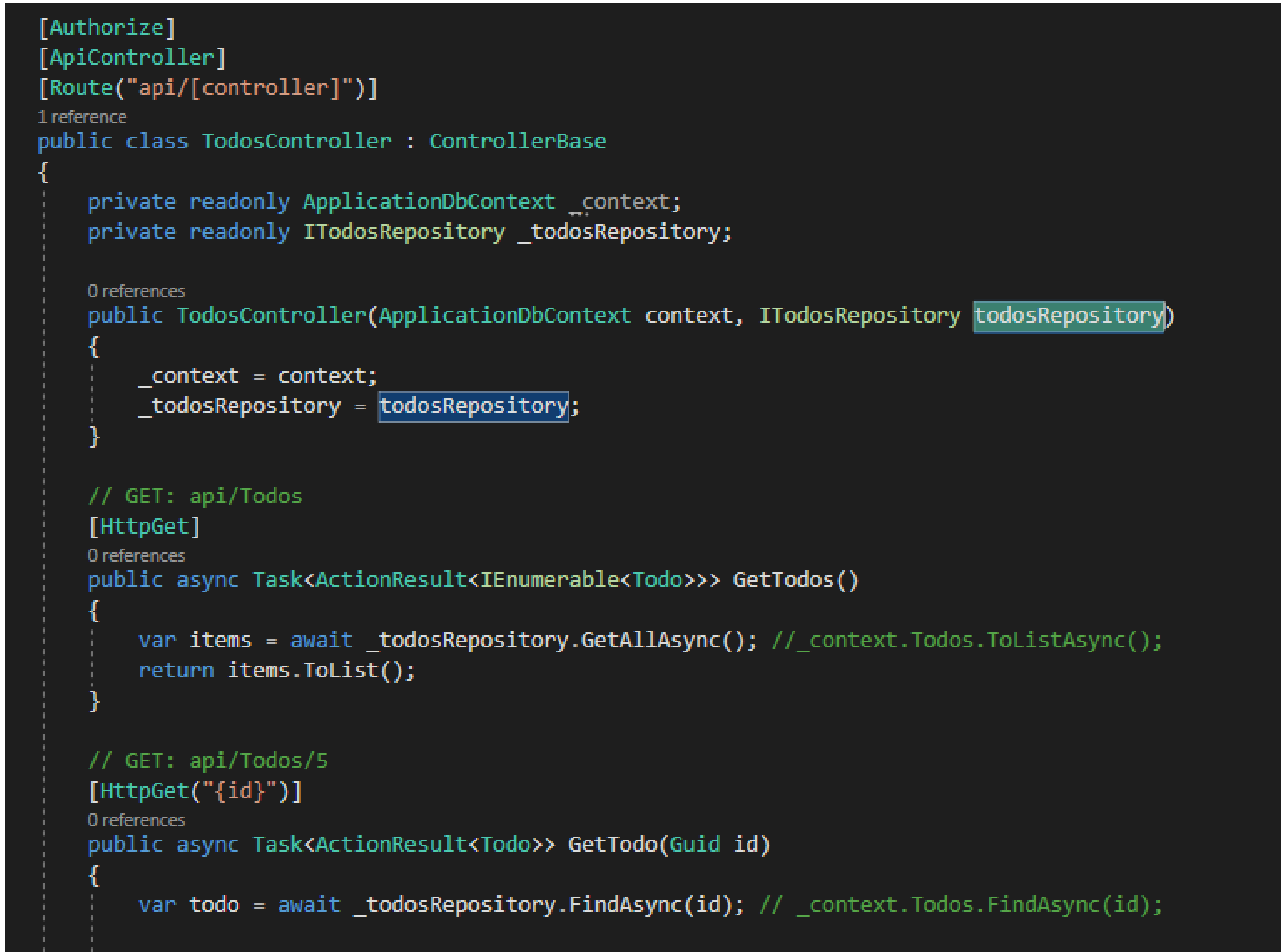
- Step 4: Dependency injection



```
// For more information on how to configure your application, visit https://go.microsoft.com/fwlink/?LinkID=
public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<ApplicationDbContext>(options =>
        options.UseSqlServer(
            Configuration.GetConnectionString("DefaultConnection")));


    services.AddDatabaseDeveloperPageExceptionFilter();

    services.AddDefaultIdentity<ApplicationUser>(options => options.SignIn.RequireConfirmedAccount = true)
        .AddEntityFrameworkStores<ApplicationDbContext>();

    services.AddIdentityServer()
        .AddApiAuthorization<ApplicationUser, ApplicationDbContext>();

    services.AddAuthentication()
        .AddIdentityServerJwt();

    services.AddControllersWithViews();
    services.AddRazorPages();

    // Register the Swagger generator, defining 1 or more Swagger documents
    services.AddSwaggerGen();

    services.AddScoped<ITodosRepository, TodosRepository>();
}
```

Repository pattern

- Step 5: Inject the repository as a dependency



```
[Authorize]
[ApiController]
[Route("api/[controller]")]
1 reference
public class TodosController : ControllerBase
{
    private readonly ApplicationDbContext _context;
    private readonly ITodosRepository _todosRepository;

    0 references
    public TodosController(ApplicationDbContext context, ITodosRepository todosRepository)
    {
        _context = context;
        _todosRepository = todosRepository;
    }

    // GET: api/Todos
    [HttpGet]
    0 references
    public async Task<ActionResult<IEnumerable<Todo>>> GetTodos()
    {
        var items = await _todosRepository.GetAllAsync(); //_context.Todos.ToListAsync();
        return items.ToList();
    }

    // GET: api/Todos/5
    [HttpGet("{id}")]
    0 references
    public async Task<ActionResult<Todo>> GetTodo(Guid id)
    {
        var todo = await _todosRepository.FindAsync(id); // _context.Todos.FindAsync(id);
```

Resources

- Useful links
  - http://blackwasp.co.uk/gofpatterns.aspx
  - https://docs.microsoft.com/en-us/dotnet/architecture/microservices/microservice-ddd-cqrs-patterns/infrastructure-persistence-layer-design
- Books
  - Design Patterns: Elements of Reusable Object-Oriented Software, by The Gang of Four (Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides)
  - Head First Design Patterns, A Brain-Friendly Guide, by Eric Freeman, Elisabeth Robson, Bert Bates, Kathy Sierra