The Learning House

427 S 4th Street #300

Louisville KY 40202

Layered Architecture and Key Design Principles

.NET Cohort

Coding Bootcamp

SOFTWARE GUILD

Lesson Goals

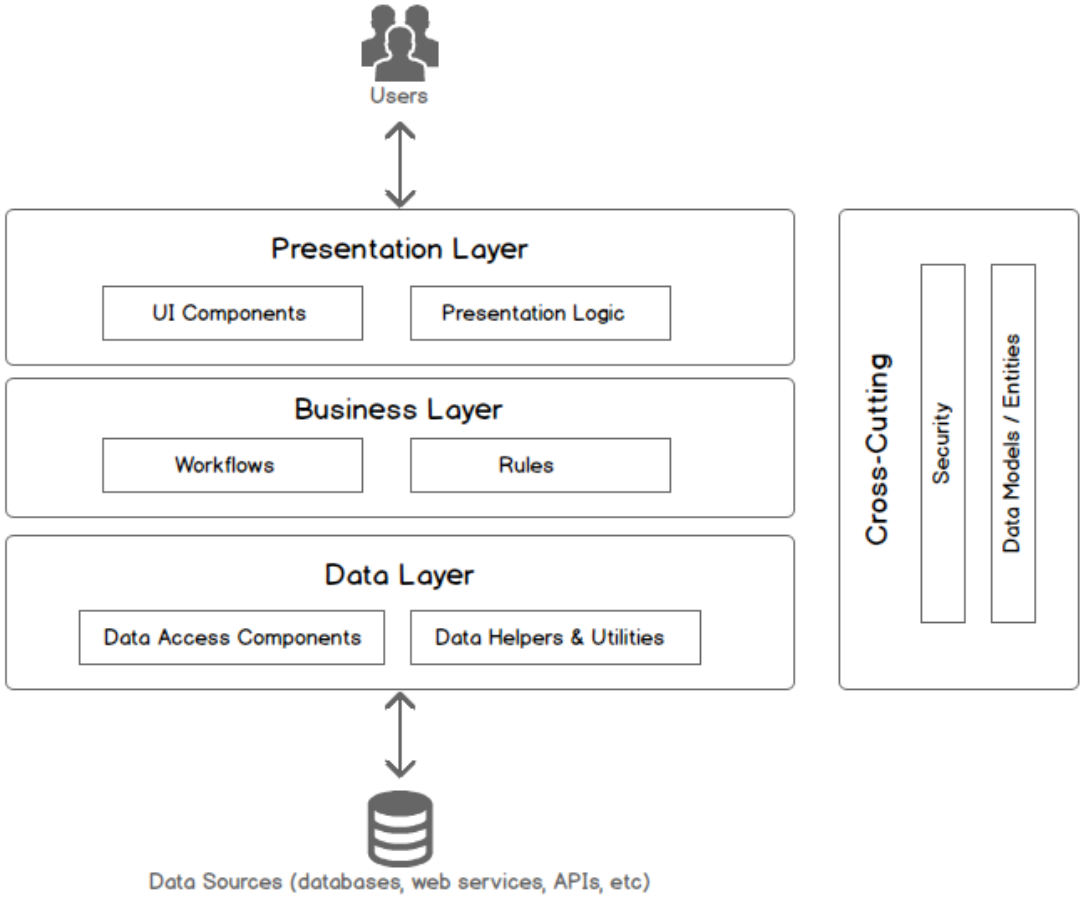
- Understand what layering is and the benefits it brings to large applications and development teams
- Learn key design principles that make code more maintainable and scalable

Professional Developers Layer Their Applications

When structuring large applications, developers run into some challenges:

- When dealing with many components, we want to group related components together. This way, changes we have to make are localized to one part of the solution whenever possible.
- We want to separate our concerns. That is to say our user interface should be separated from business rules, and both should be separate from our database calls.
- Our components should be reusable by multiple applications and hostable across physical tiers.
- Independent teams should be able to work on pieces of the solution in isolation to the others.
- Components should be highly cohesive, loosely coupled, and testable.

A Sample Layered Architecture



Has your IT team ever “fixed” one thing and broken something else?

KEY DESIGN PRINCIPLES

Don’t Repeat Yourself

You should only specify intent in one single place.

For example, if you write code that calculates a tip, that code should only exist in one place and be called from other places.

We try to avoid duplicating functionality at all costs. If duplicated functionality changes later, it takes more work to change and errors are more likely.

Separation of Concerns

If our components are to be truly reusable, we must ensure that they don't overlap with other features.

For example: in our diagram, the data layer should not know or care how the presentation layer is displaying the data. Its only concern in the storage and retrieval of data.

Additionally, if the validation rules are enforced in the business layer, the data layer may not need to be concerned with data validation. It assumes data that gets this far down the layers is good data.

Single Responsibility Principle

Each component in your application should be responsible for only one specific feature or functionality, or aggregation of cohesive functionality.

* In layman's terms: in a store application, I would not expect to find inventory code inside the "create user account" class.

** Tip: If you feel the need to name something with the word "and", you are probably violating single responsibility principle.

Principle of Least Knowledge

This is also known as the Law of Demeter. It means that your components should not know about the internal details of other components.

One component should ask another component to do something for it. It should not seek to control or examine the internal state of other components.

TL;DR Don’t talk to strangers

Breaking the Law of Demeter

The classic example of the Law of Demeter being broken is that of the customer, the wallet, and the paperboy.

A customer has a wallet and the paperboy wants to get paid by the customer.

Notice that the Paperboy class is reaching inside the Customer, taking its Wallet, and then performing logic on the Wallet.

```
public class Wallet
{
    public decimal Money { get; private set; }

    public void AddMoney(decimal amount)
    {
        Money += amount;
    }

    public void SubtractMoney(decimal amount)
    {
        Money -= amount;
    }
}

public class Customer
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public Wallet Wallet { get; set; }
}

public class Paperboy
{
    public void CollectMoneyFrom(Customer myCustomer)
    {
        decimal payment = 2.00M; // I want my $2!

        Wallet theWallet = myCustomer.Wallet;

        if (theWallet.Money > payment)
        {
            theWallet.SubtractMoney(payment);
        }
        else
        {
            // code to send the customer to collections
        }
    }
}
```

Why is this bad?

Would you let a paperboy stop by, take your wallet out of your back pocket, and remove two dollars?

In the real world, there are a number of problems here. We are trusting the paperboy with the entire contents of our wallet. If the wallet had credit cards, the paperboy object would have access to those too. The paperboy could credit and debit all it wants, and set the wallet to a negative number.

We are trusting the Paperboy (and any other object that wants our wallet) to do the right thing!

```
public class Wallet
{
    public decimal Money { get; private set; }

    public void AddMoney(decimal amount)
    {
        Money += amount;
    }

    public void SubtractMoney(decimal amount)
    {
        Money -= amount;
    }
}

public class Customer
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public Wallet Wallet { get; set; }
}

public class Paperboy
{
    public void CollectMoneyFrom(Customer myCustomer)
    {
        decimal payment = 2.00M; // I want my $2!

        Wallet theWallet = myCustomer.Wallet;

        if (theWallet.Money > payment)
        {
            theWallet.SubtractMoney(payment);
        }
        else
        {
            // code to send the customer to collections
        }
    }
}
```

Why is this bad? (2)

When we compile our code, the Paperboy class needs Customer and Wallet, so a change to Wallet can ripple into the Paperboy class. These classes are “tightly coupled.”

What if later we add a Thief class? When the Thief steals a Wallet, the Wallet is set to null. The Paperboy class, however, assumes there will always be a Wallet! The compiler won’t notice this as a problem — at run time, if the Paperboy visits a Customer who has been robbed, the CollectMoneyFrom() method will get a null reference exception and crash!

(This happens in real life code more often than you think.)

```
public class Wallet
{
    public decimal Money { get; private set; }

    public void AddMoney(decimal amount)
    {
        Money += amount;
    }

    public void SubtractMoney(decimal amount)
    {
        Money -= amount;
    }
}

public class Customer
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public Wallet Wallet { get; set; }
}

public class Paperboy
{
    public void CollectMoneyFrom(Customer myCustomer)
    {
        decimal payment = 2.00M; // I want my $2!

        Wallet theWallet = myCustomer.Wallet;

        if (theWallet.Money > payment)
        {
            theWallet.SubtractMoney(payment);
        }
        else
        {
            // code to send the customer to collections
        }
    }
}
```

Fixing the code

Notice that the Customer no longer exposes the Wallet object publically. Now, other classes like the Paperboy are prevented from manipulating the object outside of the public methods we have exposed.

The Paperboy code is now simplified. It simply asks the Customer for a payment and compares the values. In the previous example, it had access to the internals of the Customer and the Wallet. In this new structure, we can completely rewrite the GetPayment() method and the Paperboy code won't need to change at all.

We have isolated the impact of changes!

```
public class Customer
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    private Wallet _myWallet;

    public decimal GetPayment(decimal billAmount)
    {
        if (_myWallet != null)
        {
            if (_myWallet.Money >= billAmount)
            {
                _myWallet.SubtractMoney(billAmount);
                return billAmount;
            }
        }

        return 0;
    }
}

public class Paperboy
{
    public void CollectMoneyFrom(Customer myCustomer)
    {
        decimal payment = 2.00M; // I want my $2!

        decimal paymentAmount = myCustomer.GetPayment(payment);

        if (paymentAmount == payment)
        {
            // thank the customer and give a receipt
        }
        else
        {
            // code to send the customer to collections
        }
    }
}
```

Coding Style Guidelines

Make sure that your organization has an established coding style and naming standards.

The goal is that no one should be able to tell who wrote a module just by looking at it.

This makes the code easier to review, which leads to better maintainability.

Conventions we prefer at the Guild

Layout Conventions

- Use default code editing settings (tabs, spaces, etc.)
- Write one statement per line
- Write one declaration per line
- Add at least one blank line between method definitions and property definitions
- Use parenthesis to make clauses in an expression clear

Example

```
public string StringProperty { get; set; }

public void MyMethod()
{
    int x = 1;
    int y = 2;
    int z = 3;

    if ((x > y) && (y > z))
    {

    }
}
```

SOFTWARE GUILD

Implicitly Typed Local Variables

Guidelines

- Use var when the type of variable is obvious, and for loop control variables
- Do not use var if it is not clear from context
- Do not rely on the variable name to specify the type

Example

```
// DO
var s = "This is obviously a string";
var i = Convert.ToInt32(Console.ReadLine()); // clearly int

for (var j = 0; j < 10; j++)
{

}

foreach (var c in s)
{

}

// Do Not
var inputInt = Console.ReadLine(); // this is not an int
var hrm = SomeMethod(); // not sure what type this is
```

Proper (Pascal) Casing

Proper case the following:

- Class, Interfaces, Structs, Enums, and other Types
- Method names
- Public properties
- Constants
- Project names
- Folder names

Example

```
namespace IntroductionToVisualStudio.UI.CodeConventions
{
    public class ProperCases
    {
        public string StringProperty { get; set; }

        public const string HelloMessage = "Hello";

        public void MyMethod()
        {

        }
    }
}
```

Camel Casing

Camel Case the following:

- Method variables
- Method parameters
- Private fields

Example

```
class CamelCases
{
    // This is fine
    private int orderNumber;
    // I prefer this
    private int _orderNumber;

    public void MyMethod(string barCode, int quantity)
    {
        int itemsInStock = GetItemsInStock(barCode);
    }
```

Keep in mind

Almost every shop is different.

Ultimately, as a junior member of the team, your role is to fit in with the existing style, architecture, and culture.