Software Components' Architecture

Memi Lavi
www.memilavi.com

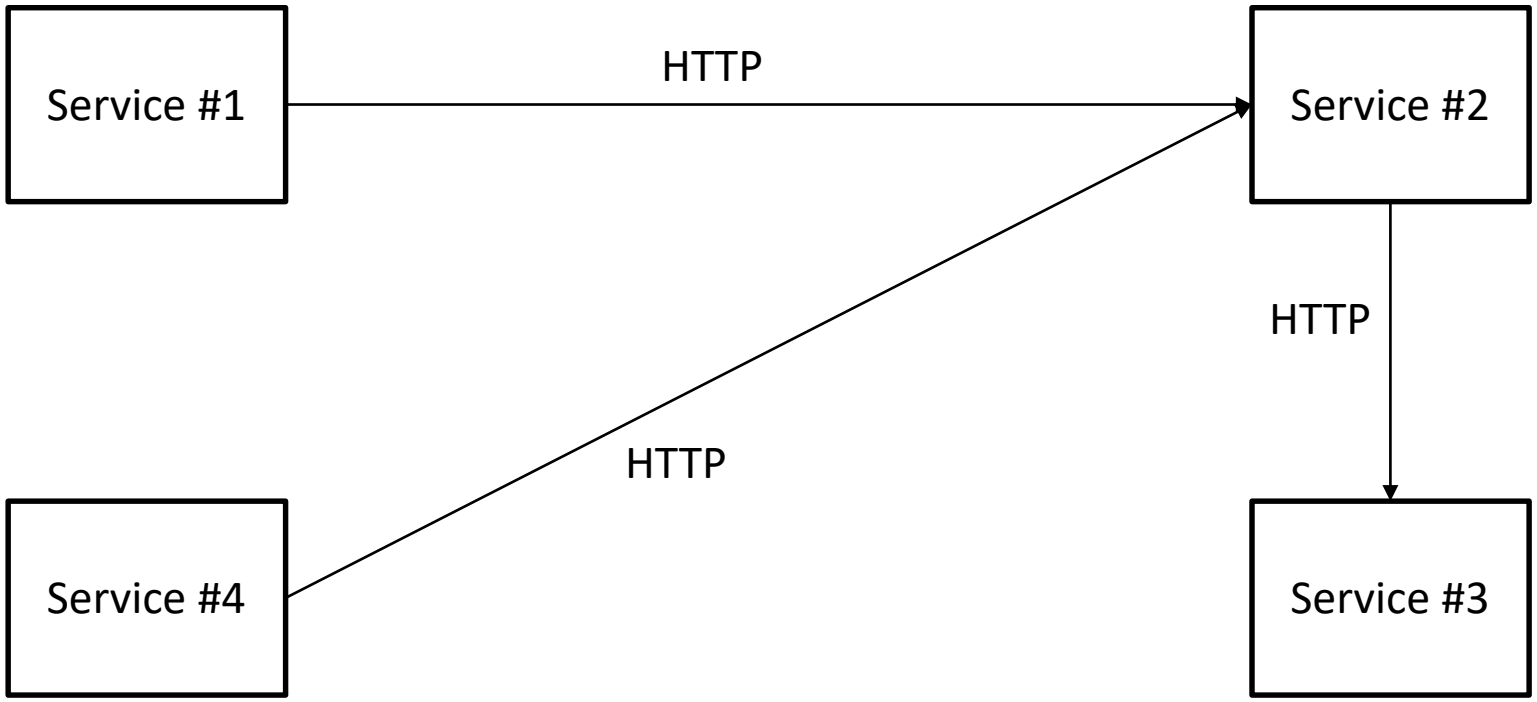
Software Component:

A piece of code that runs in a single process

Distributed Systems:

- Composed of independent Software Components
- Deployed on separate processes
- …or containers
- …or servers

Typical Distributed System



Note: This diagram depicts a very simple, non-best-practiced system, for clarity purpose only.

Two Levels of Software Architecture

- Component's Architecture
  - Inner Components
  - Interaction between them
  - Make the code fast and easy to maintain

Two Levels of Software Architecture

- System Architecture
  - Bigger Picture
  - Scalable, Reliable, Fast, Easy to maintain

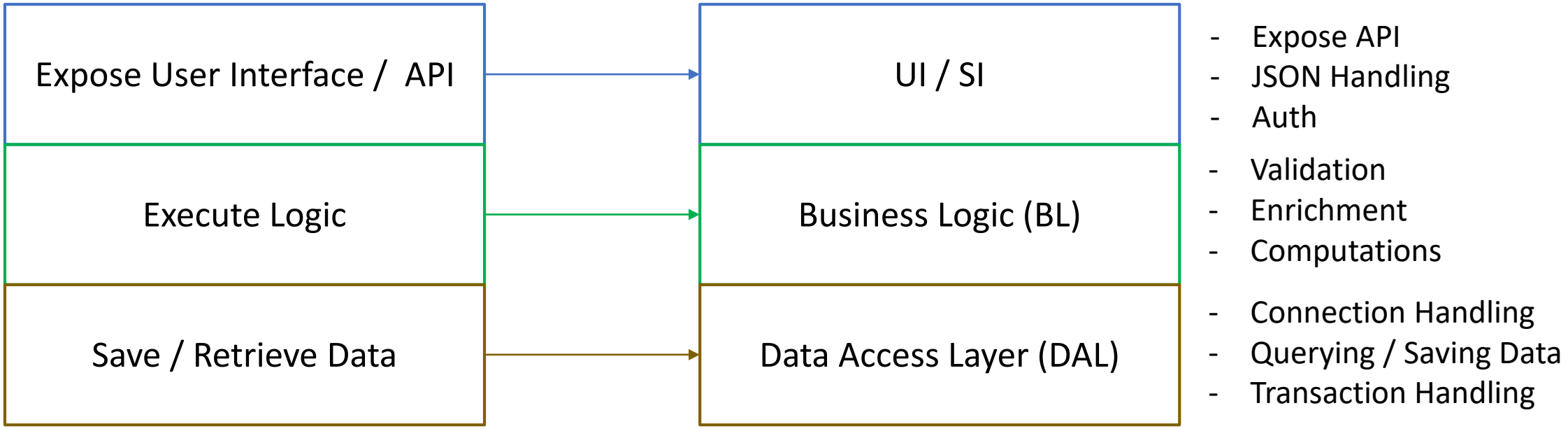
Low Level Discussion Ahead!


Do Not Distance Yourself from the Code

Layers

Layers

- Represent horizontal functionality

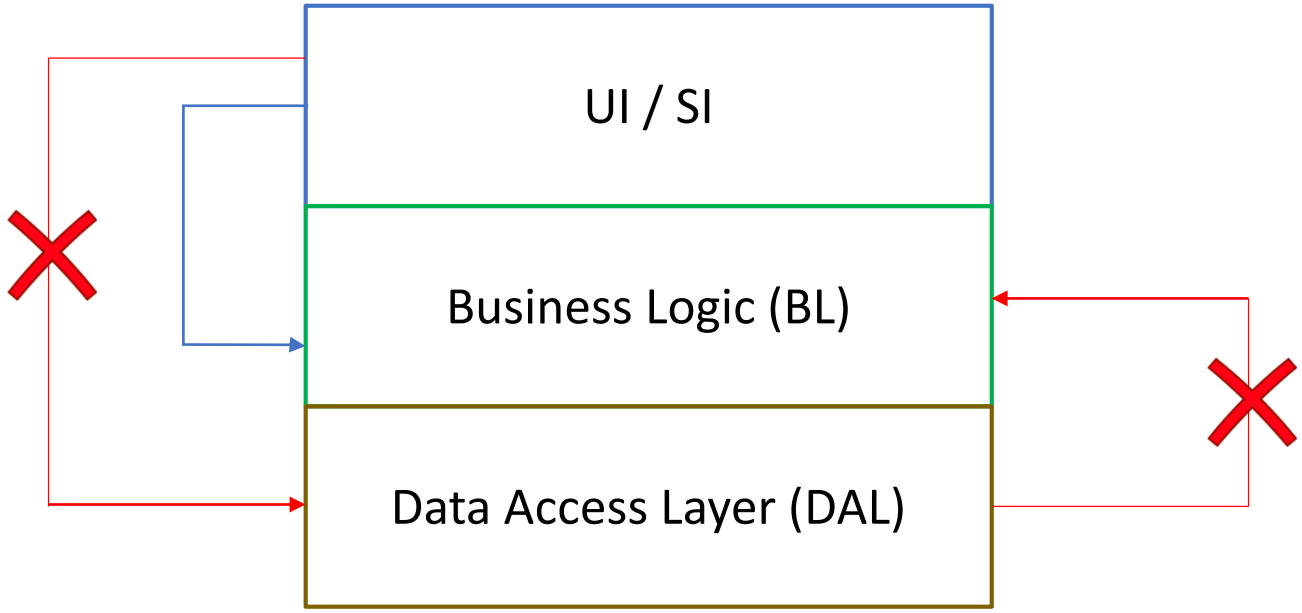


Purpose of Layers

- Forces well formed and focused code
- Modular

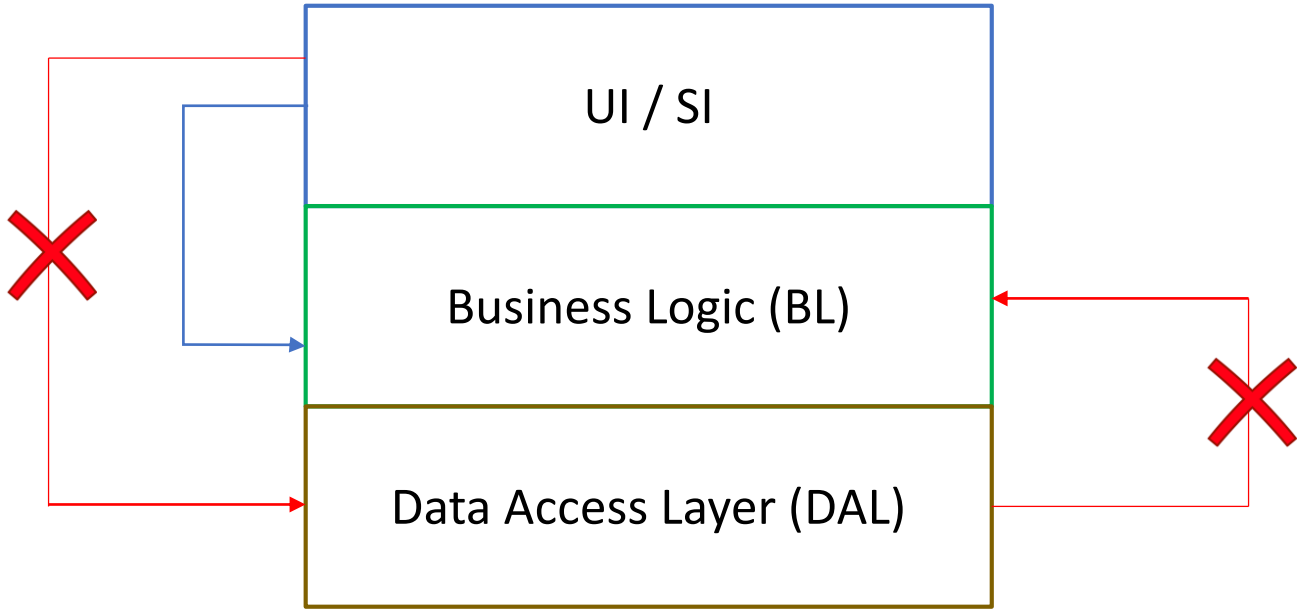
Concepts of Layers

- Code Flow



Concepts of Layers

- Code Flow



Concepts of Layers

- Loose Coupling

```
DAL dal=new DAL();
var entity=dal.GetData(101);
```

Concepts of Layers

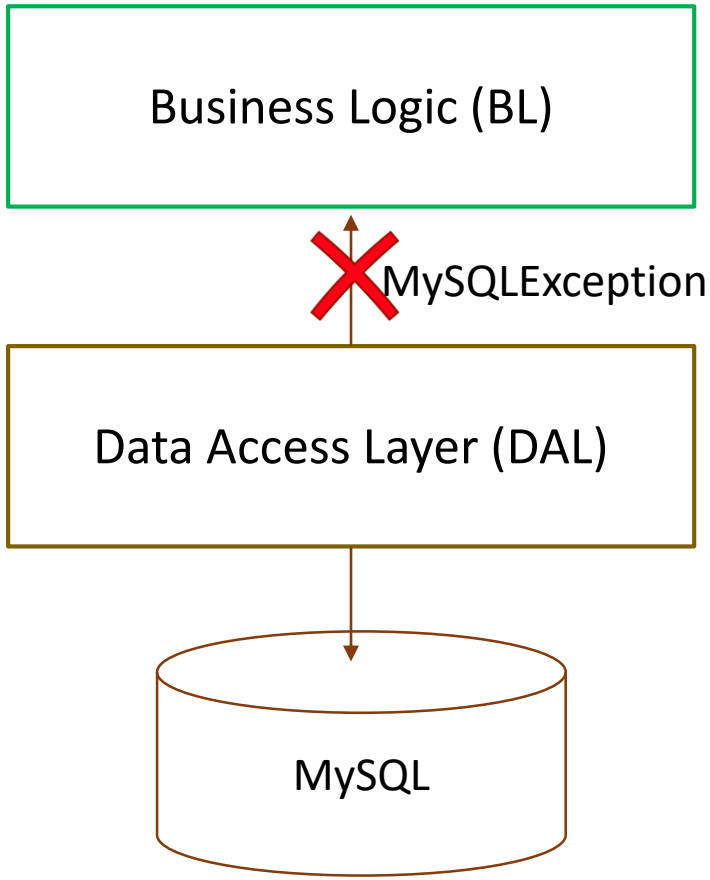
- Loose Coupling

```
IDAL dal=GetDAL();
var entity=dal.GetData(101);
```

Dependency Injection

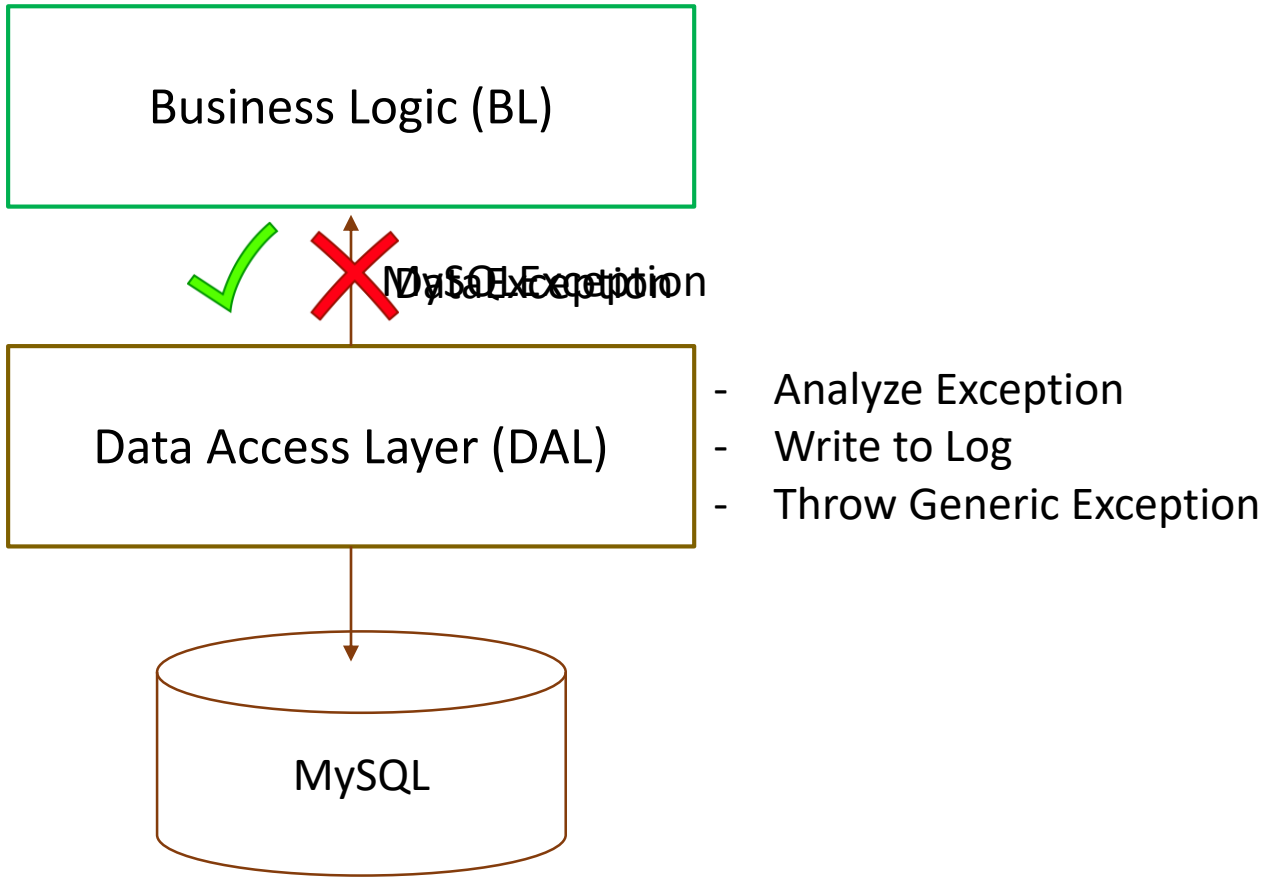
Concepts of Layers

- Exception Handling



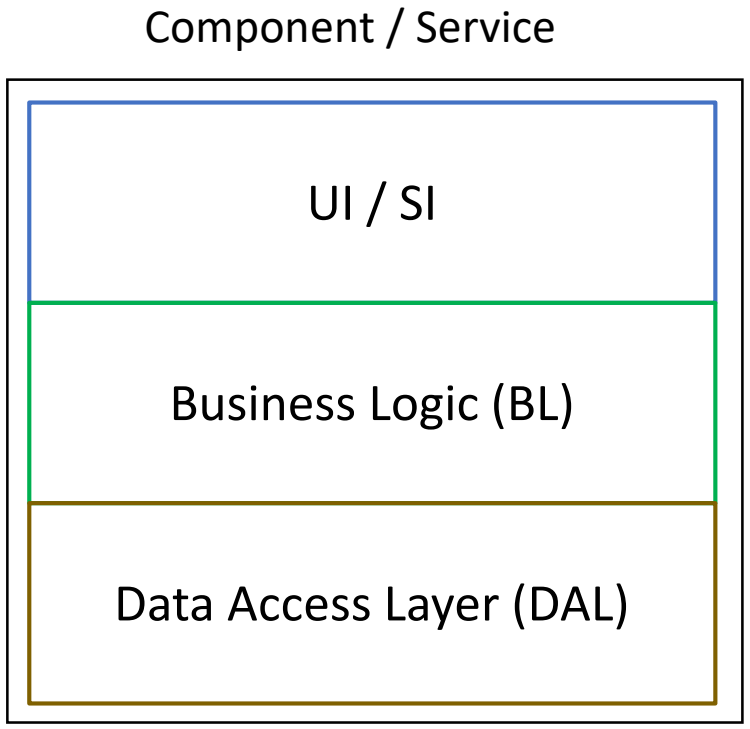
Concepts of Layers

- Exception Handling

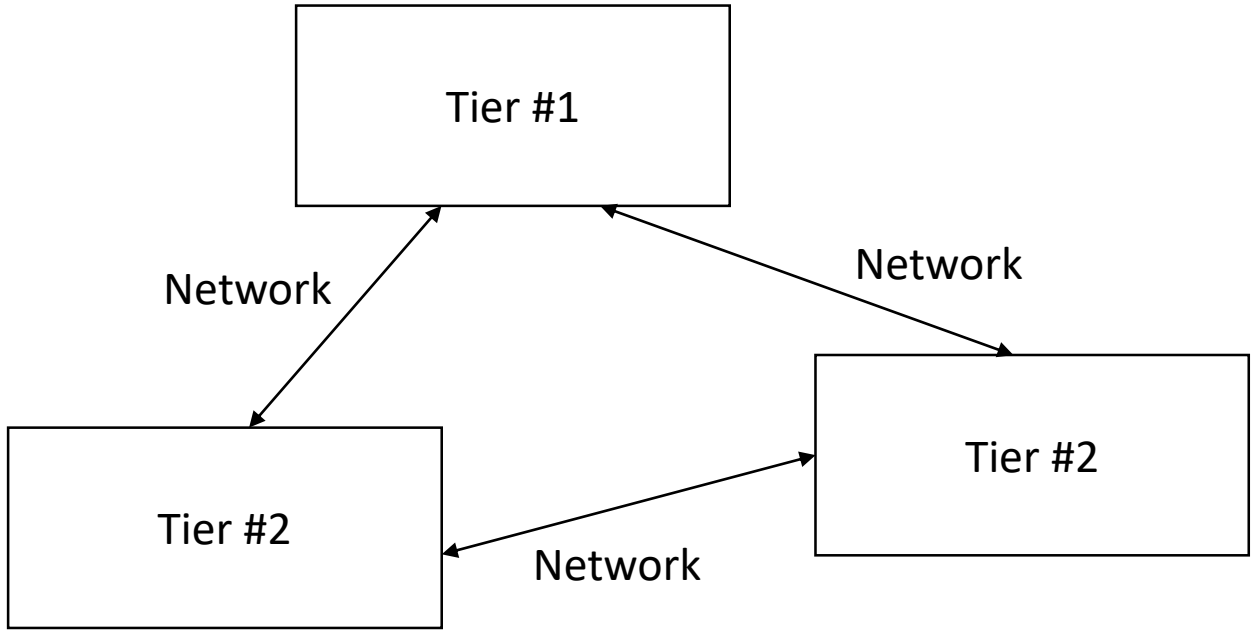


Layers vs Tiers

Layers:



Tiers:



Interfaces

Interface

- A contract
- Declares the signatures of an implementation

The Calculator

```
0 references
interface ICalculator
{
    0 references
    double Add(double num1, double num2);

    0 references
    double Subtract(double num1, double num2);

    0 references
    double Multiply(double num1, double num2);

    0 references
    double Divide(double num1, double num2);
}
```

The Calculator Without Interfaces

```
0 references
public Main()
{
    Calculator calc = new Calculator();
    double result = calc.Add(5, 2);
}
```

The Calculator Without Interfaces

```
0 references
public Main()
{
    AdvancedCalculator calc = new AdvancedCalculator();
    double result = calc.Add(5, 2);
}
```

New Is Glue

The Calculator With Interfaces

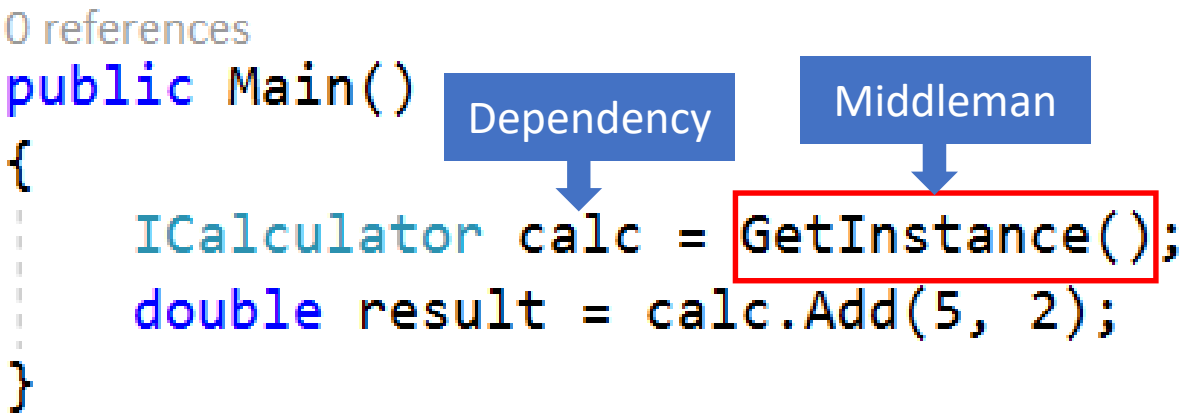
```
0 references
public Main()
{
    ICalculator calc = GetInstance();
    double result = calc.Add(5, 2);
}
```

Dependency Injection

Dependency Injection

A technique whereby one object … supplies the dependencies of another object

Dependency Injection Example



GetInstance Implementation #1

```
private ICalculator GetInstance()
{
    return new Calculator();
}
```

GetInstance Implementation #2

```
private ICalculator GetInstance(string type)
{
    switch (type)
    {
        case "regular":
            return new Calculator();
        case "advanced":
            return new AdvancedCalculator();
        default:
            return new Calculator();
    }
}
```

GetInstance Implementation #2

```
private ICalculator GetInstance()
{
    var type = ConfigurationManager.AppSettings["calcType"];

    switch (type)
    {
        case "regular":
            return new Calculator();
        case "advanced":
            return new AdvancedCalculator();
        default:
            return new Calculator();
    }
}
```

Constructor Injection

```
public class HomeController : Controller
{
    ILogger logging;

    public HomeController(ILogger logging)
    {
        this.logging = logging;
    }

    Action Methods
}
```

Constructor Injection Testing

```
[TestMethod]
public void TestControllerInitialized()
{
    var home = new HomeController(new MockLogger());
    //
    // Go on with the test...
    //
}
```

Dependency Injection

- Not trivial, but…
- Makes the code modular, flexible and easy to maintain
- Give it a try!

SOLID

SOLID

- Coined by Bob Martin in 2000
- Stands for:
  - Single Responsibility Principle
  - Open/Closed Principle
  - Liskov Substitution Principle
  - Interface Segregation Principle
  - Dependency Inversion Principle

Single Responsibility Principle

“Each class, module or method should have one, and only one, responsibility”

Single Responsibility Principle

Logging Engine

- What should be written?
- Where should it be written?

```
public void Log(string message)
{
     // compose the log message
     .
     .
     .

     // write the log message to a file
     .
     .
     .
}
```

```
public void Log(string message)
{
     // compose the log message
     var composed=Composer.ComposeMsg(message);

     // write the log message to a file
     Writer.WriteMsg(composed)
}
```

Changes affect only a well defined module

Open / Closed Principle

“A software entity should be open for extension but closed for modification”

Open / Closed Principle

- Can be implemented using:
  - Class Inheritance
  - Plug-In Mechanism
  - And more…

Make our code as flexible as possible

Liskov Substitution Principle

"If S is a subtype of T,
then objects of type T may be replaced with
objects of type S,
without altering any of the desired properties
of the program"

Liskov Substitution Principle

- Might look similar to Polymorphism…
- Deals with Behavioral Subtyping

Liskov Substitution Principle

```
private void SendMail(Message msg)
{
    var sender = new MailSender();
    sender.Send(msg);
}
```



```
private void SendMail(Message msg)
{
    var sender = new AdvancedSender();
    sender.Send(msg);
}
```

Behavior Should Not Be Changed!

ie. Don't send to archive

Avoid unmaintainable code

Interface Segregation Principle

“Many client-specific interfaces are better than one general-purpose interface”

Interface Segregation Principle

```
class DataProcessor
{
    public string ReadData()...

    public bool ValidateData(string data)...
}
```

```
interface IDataProcessor
{
    string ReadData();

    bool ValidateData(string data);
}
```

Interface Segregation Principle

```
class DataProcessor : IDataProcessor
{
    public string ReadData()...

    public bool ValidateData(string data)...

    public string EncodeData(string data)...

    public string DecodeData(string data)...

    public void SendDataToExternalSystem(string data)...
}
```

```
interface IDataProcessor
{
    string ReadData();

    bool ValidateData(string data);

    string EncodeData(string data);

    string DecodeData(string data);

    void SendDataToExternalSystem(string data);
}
```

Bloated!

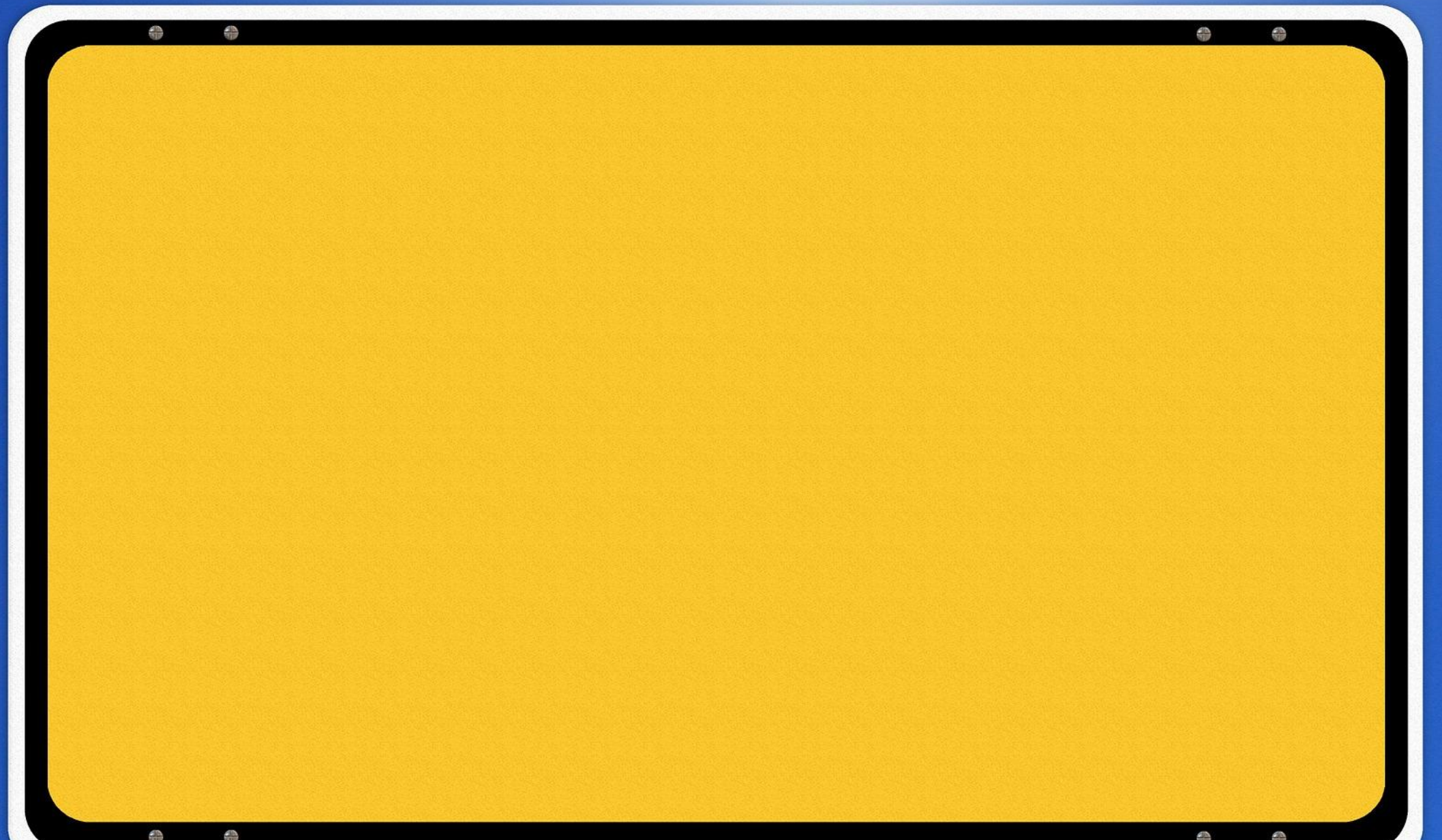
Interface Segregation Principle

```
class DataHandler : IDataHandler
{
    public string ReadData()...

    public bool ValidateData(string data)...

}

class DataEncoder : IDataEncoder
{

    public string EncodeData(string data)...

    public string DecodeData(string data)...

}

class DataSender : IDataSender {

    public void SendDataToExternalSystem(string data)...
}
```

```
interface IDataHandler
{
    string ReadData();

    bool ValidateData(string data);
}

interface IDataEncoder {

    string EncodeData(string data);

    string DecodeData(string data);
}

interface IDataSender {

    void SendDataToExternalSystem(string data);
}
```

Dependency Inversion Principle

Naming Conventions



Naming Conventions

- Define naming rules of code elements
  - Classes, Methods, Variables, Constants etc.
- Make the code more readable and easy to understand
- Not enforced by compilers
- Two types:
  - Structure (casing, underscores, etc.)
  - Content

CamelCase

- First letter of second word onward will be Capitalized

  Upper Camel Case:
  ```
  class CarEngine
  ```

  Lower Camel Case:
  ```
  class carEngine
  ```

- Popular in:

  Java, C#, JavaScript, Swift

  Recommended for class names in: Python, Ruby

lowercase_separated_by_underscore

- Name contains only lowercase letters
- Words separated by underscore

```
int num_of_parts;
```

- Popular in:

    Python & Ruby for variable names

CAPITALIZED_WITH_UNDERSCORE

- Name contains only uppercase letters
- Words separated by underscore

```
class Calendar(object)
    DAYS_IN_WEEK=7
```

- Popular in:

    Java, Python & Ruby for naming constants

Hungarian Notation

- Type information is part of the name

```
string strFirstName;
```

- Popular in:

  Nowhere. Avoid it.

Content

Class Names	Nouns (DataRetriever, Car, Network)
Methods Names	Imperative Verbs (RetrieveData, Drive, SendPacket)

Naming Convention - Summary

- Always decide on convention
- Better stick to a standard
- Do it ASAP
- Follow it!

Exception Handling

Best Practice #1

- Catch exception only if you have something to do with it
  - And no, logging does not count
- Examples:
  - Rolling back a transaction
  - Retry
  - Wrap the exception

Best Practice #2

- Catch only specific exceptions
  - Example: SQLException
- Make sure you handle the right exception

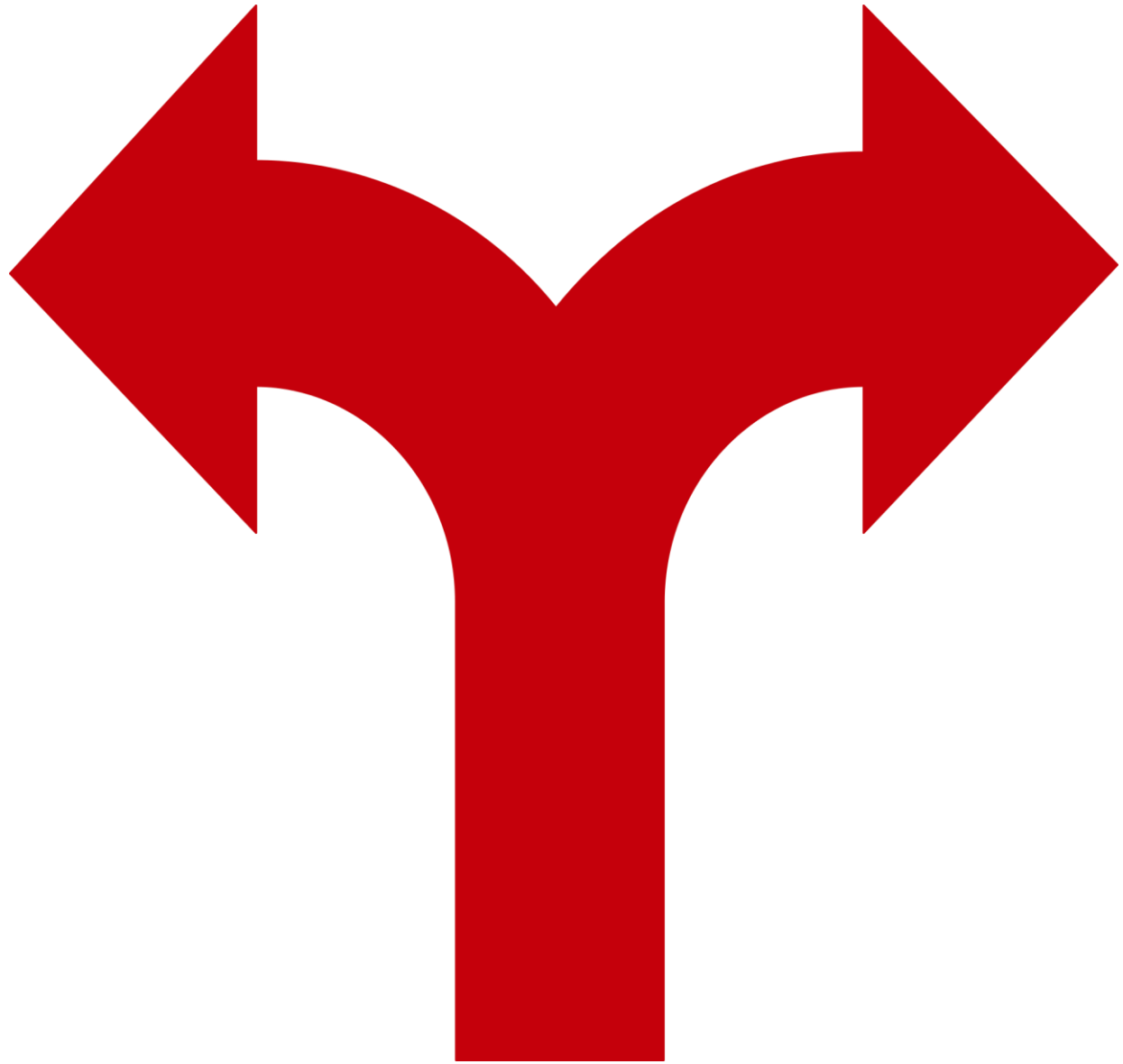
Best Practice #3

- Use try…catch on the smallest code fragments possible
- Locate the code fragments that may raise exceptions,

  and try…catch on them

Logging


4321
5
890




2

Logging Has Two Purposes

- Track Errors
- Gather data
  - Which module is most visited
  - Performance
  - User's flow

Log Storage

- Doesn't really matter
  - Files
  - Database
  - Event Log

Summary

- These are the building blocks of almost any software component
- System Architecture complements it
- Contact me with any question or comment!