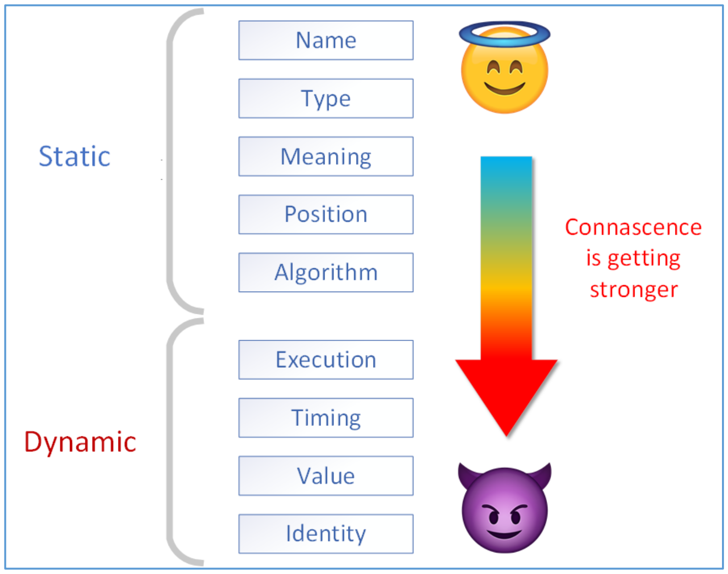
Connascence thinking

Types of Connascence

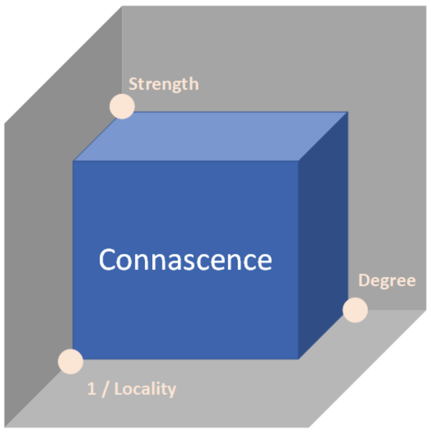


Properties of connascence

There are three identified properties of connascence:

1. **Strength**. The higher level of connascence, the higher the strength. Example, the strength of connascence of Name is lower than the strength of connascence of Algorithm.
2. **Locality**. It describes how close the coupled components are. The higher connascence locality, the better. Coupled methods that are in different modules are much worse than coupled methods within same module.
3. **Degree**. It describes how many components are coupled. Example, if we change implementation of one component, how many other components do we need to change.

```
Connascence = Strength x Degree / Locality
```

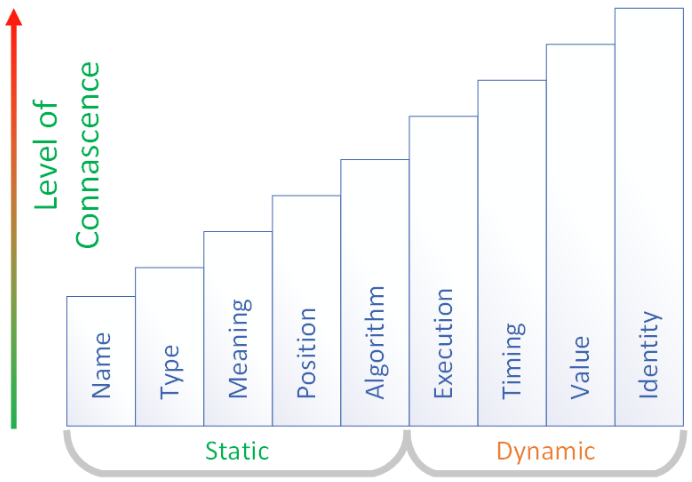


Fixing Static Connascence

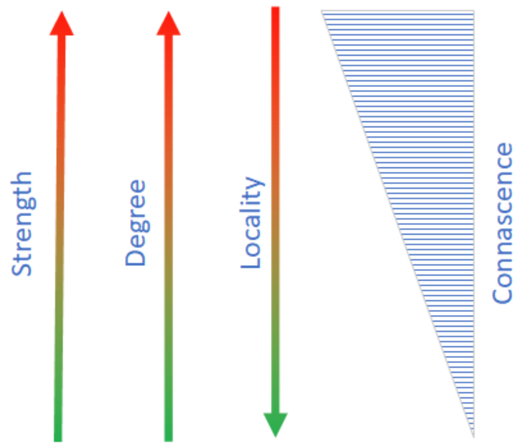
Source: https://programhappy.net/2020/07/19/fixing-static-connascence/

Introduction

In the previous post we described connascence and explained why it's so important to understand for software developers and architects. In this post we will show practical approach on how to fix different levels of static connascence. As to summarize, connascence has nine levels (five static and 4 dynamic). The higher level the less is code quality.



Connascence has three properties. Think of connascence properties as toggles that we can be turned in order to improve code quality withing the same level of connascence. All three properties contribute to connascence at the same time. For example, connascence of position withing private members of a class is not as bad as connascence of position that spans through entire codebase.



Connascence of Name

Connascence of name is the least evil level. It's also hard to avoid as it happens every time we have a named construct that is referenced by other code. Simple examples, invoke a function by name, instantiate a class by name, pass named parameter, reference global constant or variable. The main idea captured by connascence of name is when a name changes, all code that reference the name needs to change as well.

Code below shows different variations of connascence of name, comments explain which is where.

```
public class ScheduleItem
{
    public string Flight { get; set; }
    public string Origin { get; set; }
    public string Destination { get; set; }
    public DateTimeOffset DepartureDt { get; set; }
    public DateTimeOffset ArrivalDt { get; set; }

    public string Print()
    {
        // 1. We are referring properties here.
        //    If a property name chages, we have to change it
        //    in this line as well.
        return $"Flight {Flight} from {Origin} on {DepartureDt} to {Destination} on
{ArrivalDt}";
    }
}

public class Itinerary
{
    // 2. If we rename ScheduleItem class we will have to change
    //    the class name here to make code compilable.
    // 3. We may also need to change _scheduleItems field names
    //    to keep it consistent with the ScheduleItem class name.
    private readonly IList<ScheduleItem> _scheduleItems;

    public void Print()
    {
        foreach (var item in _scheduleItems)
        {
            // 4. If we change Print methond name, we will have to
            //    change it in this line as well.
            Console.WriteLine(item.Print());
        }
    }
}
```

Most of modern IDEs support renaming capabilities therefore changing name is least of a problem. For statically typed languages like C#, Golang, C++ and other connascence of name is not a problem because of compiler. If we introduce an error while renaming then compiler will not be able to resolve the name and compilation will fail. We will know about the error right away and the error will not make its way into production.

Different story is for dynamically typed languages like Python, JavaScript and other alike. Unless rigorous linters and extensive automated testing is used a trivial renaming may introduce an error that may end up in production and cause a failure on runtime.

Fix Connascence of Name

This is very basic level of connascence that does not have a fix and inevitable. The only thing we can do is to work with connascence properties, e.g. increase locality, reduce degree and strenght.

Connascence of Type

This connascence level appears quite often when code makes statements or assumptions about the types it operates on. Simple example, a function takes an argument of an integer type, so we can’t pass a string value to the function.

```
void Sum(int a, int b);
```

For statically strongly typed languages (C#, C++, Golang and alike) compiler takes care of most issues that can occur as result of type violations. The code below is in C# and shows few examples of connascence of type and how compiler helps to catch the issues.

```
public class ScheduleItem
{
    public string Flight { get; set; }
    public string Origin { get; set; }
    public string Destination { get; set; }
    public DateTimeOffset DepartureDt { get; set; }
    public DateTimeOffset ArrivalDt { get; set; }

    public string Print()
    {
        return $"Flight {Flight} from {Origin} on {DepartureDt} to {Destination} on
{ArrivalDt}";
    }
}

public class Itinerary
{
    private readonly IList<ScheduleItem> _scheduleItems;

    public string Print()
    {
        List<string> printedSchedules = new List<string>();
        foreach (var item in _scheduleItems)
        {
            // COMPILE ERROR
            // printedSchedules is a collection of strings,
            // therefore we can not add ScheduleItem instance
            // into the collection.
            printedSchedules.Add(item);

            // FIXED
            // A fix is simple, developer just forgot to call Print() on item.
            // Print() returns a string that can be added to printedSchedules.
            printedSchedules.Add(item.Print());
        }

        return string.Join(Environment.NewLine, printedSchedules);
    }
}
```

In dynamically typed languages (Python, JavaScript and other) the responsibility falls on a developer, so she needs to follow the rules, write automated tests and ensure linters run before code gets deployed. Below is functionally similar code, but re-written in Python. If compared to the above we can see differences of how statically and dynamically typed languages handle issues caused by connascence of type.

```
class ScheduleItem:
def __init__(self):
self.flight = ""
self.origin = ""
self.destination = ""
self.departure_dt = None
self.arrival_dt = None

def print(self):
        return (
f"Flight {self.flight} from {self.origin} on {self.departure_dt} "
f"to {self.destination} on {self.arrival_dt}"
        )


class Itinerary:
def __init__(self):
self._schedule_items = []

def print(self):
printed_schedules = []
        for item in self._schedule_items:
        # No error, item is added to a collection that is supposed
            # to be a collection of strings. When code runs user will see that
            # itinerary is not printed correctly.
        printed_schedules.append(item)

            # A fix is simple, developer just forgot to call print() on an item.
        # However the error was discovered after the code was run.
        printed_schedules.append(item.print())

        return "\n".join(printed_schedules)
```

Fix Connascence of Type

Connascence of type is thought to be a little stronger than connascence of name, however it is very basic level of connascence that does not have a direct fix and also hard to avoid. The only thing we can do to improve connascence is to work with properties: locality, strength and degree.

Connascence of Meaning

Connascence of meaning appears when specific meaning is assigned to a value. Example, a function returns string `"SUCCESS"` when it is executed successfully. Code needs to check the function return value and compare it to `"SUCCESS"` string value. If we ever need to change the value `"SUCCESS"` we will have to change it everywhere in the code.
The code below is full of examples of connascence of meaning.

1. `"SUCCESS"` string value is returned when function `BookFlight()` executes successfully.
2. `0` is returned from `BookItinerary()` when entire itinerary is booked.

```
public class ScheduleItem
{
    public string Flight { get; set; }
    public string Origin { get; set; }
    public string Destination { get; set; }
    public DateTimeOffset DepartureDt { get; set; }
    public DateTimeOffset ArrivalDt { get; set; }
}

public class FlightBookingService
{
    private static readonly Random _rand = new Random();

    public string BookFlight(string flight, DateTimeOffset departureDt)
    {
        if (_rand.NextDouble() > 0.95)
        {
            return "FAILURE";
        }

        return "SUCCESS";
    }
}

public class Itinerary
{
    private readonly IList<ScheduleItem> _scheduleItems;

    public int BookItinerary()
    {
        var bookingService = new FlightBookingService();
        foreach (var scheduleItem in _scheduleItems)
        {
            var status = bookingService.BookFlight(
                    scheduleItem.Flight,
                    scheduleItem.DepartureDt);
            // Check if BookFlight completed successfully by
            // comparing retuned result against string value.
            if (status != "SUCCESS")
            {
                return 1;
            }
        }
        return 0;
    }
}
```

How to fix connascence of meaning

Usually a fix is very simple. We can assign a value to global constant with proper name, or create an enum to map group of values to proper enum members. This way we reduce level of connascence by converting connascence of meaning to connascence of name.

Connascence of Position

Connascence of position appears when order of elements assumes certain meaning. Example, user entity is represented as an array where position of an element carries meaning: `[FirstName, LastName, PhoneNumber]`. Another example, a function takes a long list of arguments where of course the order of arguments matters.

```
class UserFactory
{
public string[] CreateUser(
        string firstName,
        string lastName,
        string phoneNumber)
{
    // User is represented as an array where
    // each element position has meening.
    return new string[] {
            firstName,
            lastName,
            phoneNumber
    };
}
}
```

How to fix connascence of position

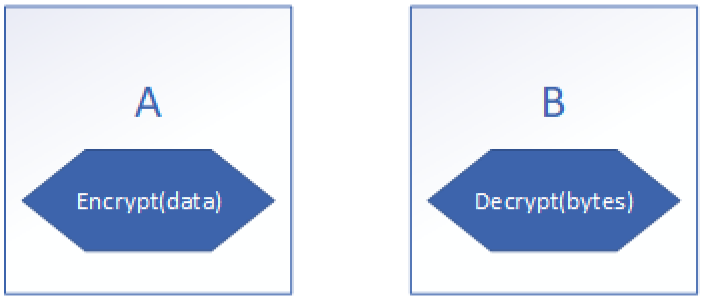
Connascence of position can be converted to connascence of name by employing proper data types, e.g. classes to represent domain entities.

Functions that take long list of arguments are most likely too large in size and are responsible for too many things. In this case refactoring to multiple functions that are more focused will not only reduce connascence but also improve code readability and maintainability. Also, there may be a proper type hiding behind a long list of arguments, so grouping arguments into a proper data type may be a good solution that will reduce level of connascence to connascence of type.

```
public class User
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public string PhoneNumber { get; set; }
}

public class UserFactoryFixed
{
    public User CreateUser(
            string firstName,
            string lastName,
            string phoneNumber)
    {
        return new User
        {
            FirstName = firstName,
                    LastName = lastName,
                    PhoneNumber = phoneNumber
        };
    }
}
```

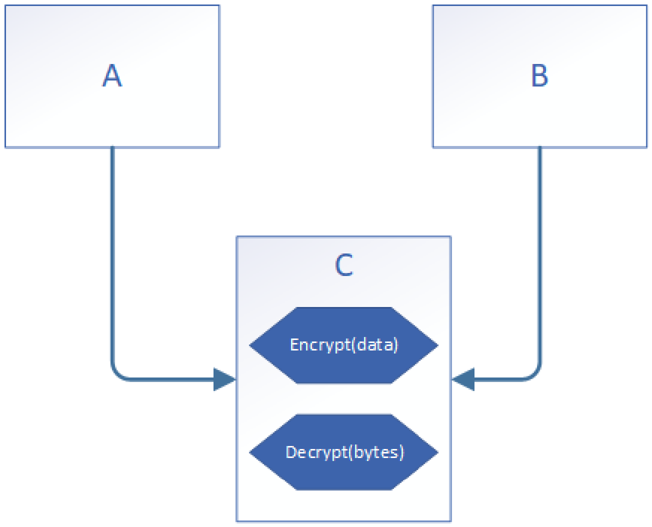
Connascence of Algorithm

Connascence of algorithm appears when components need to agree upon algorithm being used. Example, component A encrypts data and component B decrypts it. Components A and B need to agree upon encryption algorithm, otherwise component B will not be able to decrypt data that’s encrypted by component A. We can also think of string encoding (utf8, base64 and etc.) and serialization format (JSON, XML, protobuf and etc.) that need to be agreed by both sides.



How to fix connascence of algorithm

The best solution is to extract an algorithm into a separate component and share the component with other. To fix the example above we can extract encryption and decryption logic into a component C so that components A and B would use component C to encrypt and decrypt data.



Summary

We walked through all five static levels of connascence and discussed possible fixed for each of them. Most of the fixes can be distilled down to two categories:

1. Refactor code to reduce connascence level where possible. For example, convert connascence of position to connascence of type.
2. Refactor code to adjust connascence properties, increase locality and decrease degree and strength.

Fixing Dynamic Connascence

Source: https://programhappy.net/2020/09/19/fixing-dynamic-connascence/

Introduction

In the previous posts we learned about connascence and explained why it's so important to understand for software developers and architects, we looked at all 5 levels of static connascence and different approaches to improve code quality. In this post we will focus on 4 levels of dynamic connascence and will take practical approach to reduce the degree of dynamic connascence.

Connascence of Execution

Connascence of execution is the first dynamic level connascence. It appears when order of operation is important. For example, when brewing coffee, it's important to follow basic preparation steps: grind coffee, prepare, turn on a coffee machine, wait and then serve freshly brewed coffee.

Example: We have a `Barista` class that can brew given a coffee recipe. Each coffee recipe has sequence of steps that is critically important to follow in certain order. The steps may vary from one recipe to another, but basic steps and their order remain the same. Problem with this design is that `Coffee` class needs to follow the conventional sequence of recipe steps in order to prepare each coffee. When we add new class of coffee we will have to replicate the sequence of basic steps of each recipe. The need to duplicate steps creates an opportunity to make a mistake and mess the order of recipe steps.

```
public class Cappuccino
{
    public void Grind() { }
    public void PressCoffee() { }
    public void AddWater() { }
    public void AddMilk() { }
    public void TurnOnMachine() { }
    public void Wait() { }
    public void Serve() { }
}

public class Americano
{
    public void Grind() { }
    public void AddCoffee() { }
    public void AddWater() { }
    public void TurnOnMachine() { }
    public void Wait() { }
    public void Serve() { }
}

public class Barista
{
    public void BrewAmericano(Americano coffee)
    {
        // Sequence of steps is important and
        // is duplicated in every coffee class.
        coffee.Grind();
        coffee.AddCoffee();
        coffee.AddWater();
        coffee.TurnOnMachine();
        coffee.Wait();
        coffee.Serve();
    }

    public void Brew(Cappuccino coffee)
    {
        coffee.Grind();
        coffee.PressCoffee();
        coffee.AddWater();
        coffee.AddMilk();
        coffee.TurnOnMachine();
        coffee.Wait();
        coffee.Serve();
    }
}
```

How to fix connascence of Execution

This type of connascence is hard to avoid completely as most of algorithms require certain order of steps. However, we can improve our design by increasing locality of connascence. Template pattern can help us to extract the order of steps into a separate function or a class. Having algorithm implemented in one place also decreases degree of connascence by reducing number of components that duplicate the order of recipe steps.

```
public interface ICoffee
{
    void Prepare();
    void AddExtras();
}

public class Cappuccino : ICoffee
{
    public void Prepare()
    {
        PressCoffee();
        AddWater();
        AddMilk();
    }

    public void AddExtras()
    {
        MakeCinnamonPrint();
    }

    private void PressCoffee() { }
    private void AddWater() { }
    private void AddMilk() { }
    private void MakeCinnamonPrint() { }
}

public class Americano : ICoffee
{
    public void Prepare()
    {
        AddCoffee();
        AddWater();
    }

    public void AddExtras() { }

    private void AddCoffee() { }
    private void AddWater() { }
}

public class Barista
{
    // Sequence of basic steps is implemented as template method.
    public void Brew(ICoffee coffee)
    {
        Grind();
        coffee.Prepare();
        TurnOnMachine();
        Wait();
        coffee.AddExtras();
        Serve();
    }

    private void Grind() { }
    private void TurnOnMachine() { }
    private void Wait() { }
    private void Serve() { }
}
```

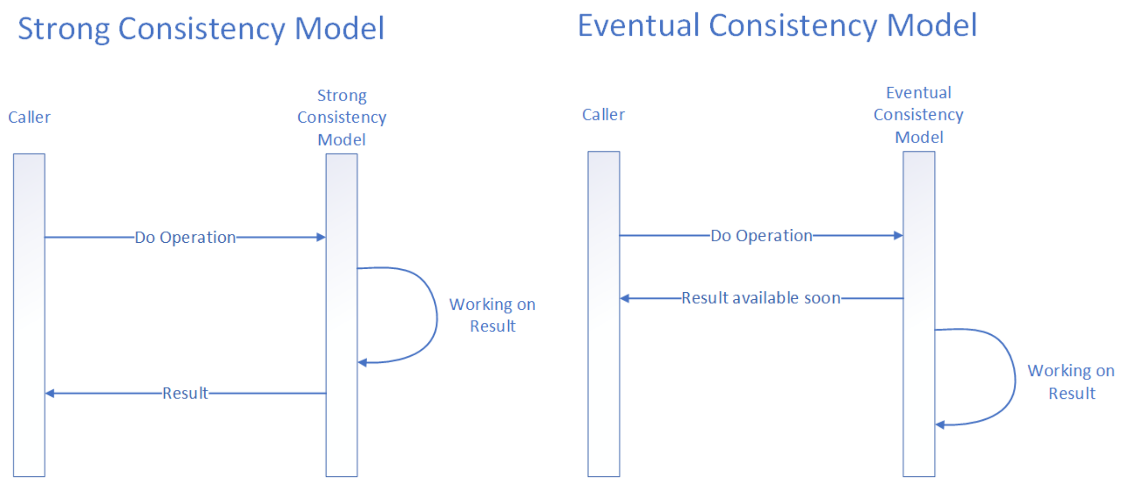
Another approach is to make order of steps hard to mess up. For example, if a class requires certain initialization steps in a particular order, we can employ Factory pattern and lock the order withing factory method.

Connascence of Timing

Connascence of Timing appears when timing of operations becomes important. This level of connascence may appear when communication between components is asynchronous. For example, one component begins an asynchronous operation, while the other component needs the result of previous operation to continue execution. If second component does not wait for the result it may start execution before the result is ready which may cause sporadic exceptions or undefined behavior on runtime.

```
public class User
{
    public int Id { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public string PhoneNumber { get; set; }
}

public abstract class UserRepository
{
    public async Task<User> SaveUser(string firstName, string lastName, string
phoneNumber)
{
    var user = new User();
    var saveUserTask = SaveUserAsync(user);
    // We need to await saveUserTask completeion.
    // If we do not, then saveUserTask may not be completed
    // before we attempt to load user. This may cause
    // undefined behavior or exception.
    var savedUserId = await saveUserTask;
    // User must be saved before we attempt to load.
    var loadedUserTask = LoadUserAsync(savedUserId);
    return await loadedUserTask;
}

    public abstract Task<int> SaveUserAsync(User user);

    public abstract Task<User> LoadUserAsync(int id);
}
```

Connascence of timing may be found on higher design levels as well, for example when strong consistency models need to communicate with eventual consistency model. With strong consistency model a result is guaranteed to be available when operation returns. Eventual consistency model guarantees operation to be completed, but has no guarantees on when this is going to happen. Therefore, strong consistency model has timing dependency on eventual consistency model.



How to fix connascence of Timing

In general, whenever we have asynchronous code, we have connascence of timing. The way we can fix connascence of timing is to keep it as much local as possible (high locality) with few components involved (low degree). Luckily there is powerful pattern at our disposal – Promises. Multiple languages have support for promises: JavaScript Promises, C# Task Parallel Library and `async / await`, Python asyncio and many other. Promises as asynchronous pattern allows to chain asynchronous operations declaratively in one place, hence achieving high locality, and also allows code reuse helping to lower the degree of connascence.

Connascence of Value

Connascence of value appears when multiple components depend on a changing value to be the same on runtime. For example, initial value of `User.Id` property is -1. If `User.Id == -1`, then `UserRepository.SaveOrUpdateUser()` method assumes that user is new and needs to be inserted, otherwise existing user needs to be updated.

```
public class User
{
    public User()
    {
        // If user is new, Id property has
        // vlaue -1
        Id = -1;
    }

    public int Id { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public string PhoneNumber { get; set; }
}

public abstract class UserRepository
{
    public int SaveOrUpdateUser(User user)
    {
        // We make an assumption if user.Id property is -1
        // then user is new and needs to be inserted.
        // Othereise we update existing user.
        if (user.Id == -1)
        {
            return InserttUser(user);
        }
        else
        {
            return UpdateUser(user);
        }
    }

    public abstract int InserttUser(User user);
    public abstract int UpdateUser(User user);
}
```

Another example is error handling through error messages and string comparison. For example, one component returns an error message and the other component checks whether error massage contains certain sub-string to identify success or failure of the operation. This type of code can be found when components communicate over network, e.g. user interface communicates with a service, or microservices communicate with each other. Dependency on error message is dangerous because if the error message changes, it can break other components. In the worst case the introduced bug can make its way to production and stay undetected for some time, in the best case it can be detected by integration tests.

```
public class User
{
    ...
}

public class UserServiceProxy
{
    public string SaveUser(User user)
    {
        // Returns result of the operation as string.
    }
}

public class UserUiController
{
    public void SaveUserData(User user)
    {
        var userService = new UserServiceProxy();
        var response = userService.SaveUser(user);
        if (!response.Contains("error", StringComparison.OrdinalIgnoreCase))
        {
            return;
        }
        if (response.Contains("database", StringComparison.OrdinalIgnoreCase))
        {
            throw new DatabaseException("Can't save user");
        }
        else
        {
            throw new ApplicationException("Fialed to save user, unknown reason");
        }

    }
}
```

How to fix connascence of Value

In many cases a fix is pretty obvious and boils down to spelling out unobvious assumptions clearly. First example can be fixed by adding `IsNew` property to `User` class which removes dependency on -1 to be initial value of `User.Id`.

```
public class User
{
    // 1. Give name to default value of User.Id
    private const int NewUserId = -1;
    public User()
    {
        Id = NewUserId;
    }

    public int Id { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public string PhoneNumber { get; set; }

    // 2. Add property to check if user is new
    public bool IsNew => Id == NewUserId;
}

public abstract class UserRepository
{
    public int SaveOrUpdateUser(User user)
    {
        // 3. Use User.IsNew
        if (user.IsNew)
        {
            return InserttUser(user);
        }
        else
        {
            return UpdateUser(user);
        }
    }

    public abstract int InserttUser(User user);
    public abstract int UpdateUser(User user);
}
```

Error handling example can be fixed by introducing standard error codes that are part of UserService interface and can only change in backward compatible way.

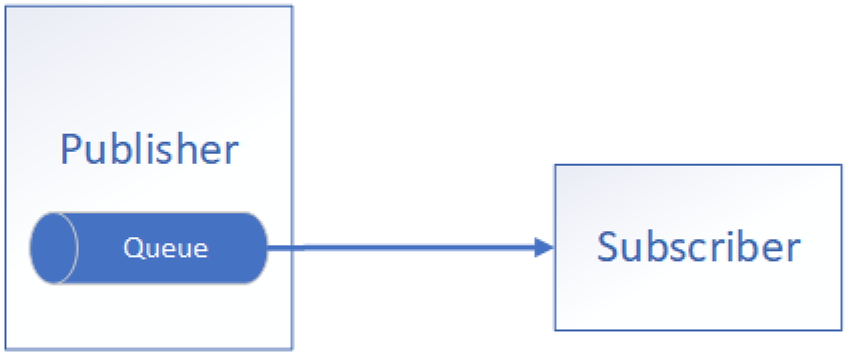
```
public class User
{
    ...
}

// 1. Define error codes explicitly
public enum UserServiceResponseCode
{
    Success = 0,
    DatabaseError = 2
}

public class UserServiceResponse
{
    public const int DatabaseError = 2;
    public UserServiceResponseCode ErrorCode { get; set; }
    public int ErrorMessage { get; set; }
    public bool IsSuccess => ErrorCode == UserServiceResponseCode.Success;
}

public class UserServiceProxy
{
    public UserServiceResponse SaveUser(User user)
    {
        ...
    }
}

public class UserUiController
{
    public void SaveUserData(User user)
    {
        var userService = new UserServiceProxy();
        var response = userService.SaveUser(user);

        // 2. Now check for success result is easy
        if (response.IsSuccess)
        {
            return;
        }

        // 3. Check for the type of error is easy too
        if (response.ErrorCode == UserServiceResponseCode.DatabaseError)
        {
            throw new DatabaseException("Can't save user");
        }
        else
        {
            throw new ApplicationException("Fialed to save user, unknown reason");
        }
    }
}
```

Connascence of Identity

Connascence of identity appears when two or more components require to have a reference to the same instance of an entity (object). The first example that comes in mind is a singleton, however with singleton we know by fact that we are dealing with shared object by design. This fact is explicit and well understood. Singleton is responsible for maintaining only one instance of its class. Therefore, when we reference a singleton this creates static connascence very similar to referencing a variable by name. So Singleton is not a good example.

Well, when can we have connascence of identity then? The most common case that I've encountered is when one object (A) creates an instance (I), the other object (B) gets access to the instance (I) through public interface of object A.

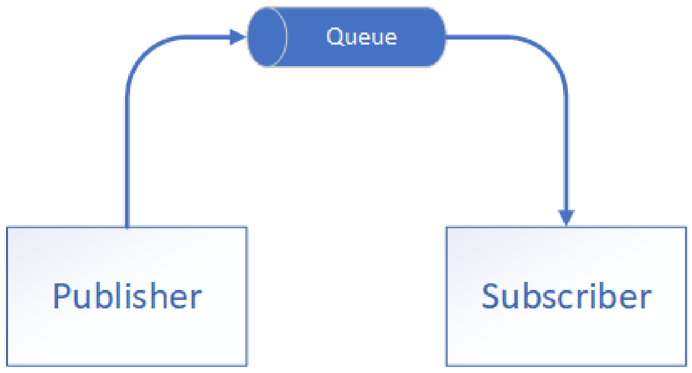
Example: we have a publisher-subscriber model. Publisher creates a queue and exposes it as public property. Publisher pushes messages to the queue. Subscriber accesses the queue through Publishers public property and pulls messages from the queue.



```
public class Publisher
{
    public Queue<string> Queue { get; private set; } = new Queue<string>();

    public void Publish(string message)
    {
        Queue.Enqueue(message);
    }
}

public class Subscriber
{
    public void Consume(Publisher publisher)
    {
        Console.WriteLine(publisher.Queue.Dequeue());
    }
}
```

How to fix connascence of Identity

To fix connascence of identity we can make shared reference explicit. In our example we can move responsibility of creating and managing queue to a broker or to a Dependency Injection container. This approach converts dynamic connascence into static.



```
public class Publisher
{
    private readonly Queue<string> _queue;

    public Publisher(Queue<string> queue)
    {
        _queue = queue;
    }

    public void Publish(string message)
    {
        _queue.Enqueue(message);
    }
}

public class Subscriber
{
    private readonly Queue<string> _queue;

    public Subscriber(Queue<string> queue)
    {
        _queue = queue;
    }

    public void Consume()
    {
        Console.WriteLine(_queue.Dequeue());
    }
}
```

Summary

This post concludes the connascence series. We covered essentials of connascence as software quality metric and walked though multiple approaches to improve code quality for all levels of connascence. Now it’s time to apply this knowledge in the real world!