JJ’s Reference Architecture

Author: Jan-Joost van Zon
Date: December 2014 – July 2017
[Under Construction]

Practices & Principles

Contents

Introduction

The design concepts mentioned here are abstract principles that you can use in software design. Many of the principes have pros and cons that need to be weighed off in every decision about the software design.

This section also describes good and bad pratices while coding your software. They are concrete technical problems you may face in your work.

There are similar terms for bad practices. *Anti-patterns* are coding techniques that are not recommended. They are not forbidden, but they are an indication that there may be a better way to do it. *Code smells* are vague indications that there might be something wrong with your code and a better option may be available.

Code smells, anti-patterns and design principles can help you review (your own) code.

The topic are organized into groups of related topics.

Responsibilities

<TODO: Write an intro on the kind of topics you will find here.>

Separation of Concerns

<TODO: Use this phrase: Keep it separated.>

(Relatesed SOLID principles Single Responsibility Principle (SRP), Interface Segregation Principle (ISP) and Do Not Repeat Yourself (DRY)).

This is the concept that you split your code into pieces and create separate classes and methods. It is perhaps the single most important design principle of this software architecture.

Separation of concerns can be a split up into functionalities, such as code that handles a whole order and code that handles a separate product. The split up into functional concerns is usually similar to the split up into entities, for instance entities like Order, Product, Customer, but this is not necessarily leading for the split up into functionality.

Separation of concerns can also be applied to technical aspects, such as validation, calculation and security. For instance: you can split up the code to check the validity of an order's data from the code that calculates the total price of the order. The split up into technical concerns is usually similar to the split up into design patterns.

Classes

In this architecture we apply both a split up into functional and technical aspects, creating a 2-dimensional separation of concerns. This produces a matrix of classes:

	Dto	**Mapping**	**Validator**	**ViewModel**	**Presenter**	**…**
Order	OrderDto	OrderMapping	OrderValidator	OrderViewModel	OrderPresenter	…
Product	ProductDto	ProductMapping	ProductValidator	ProductViewModel	ProductPresenter	…
Customer	CustomerDto	CustomerMapping	CustomerValidator	CustomerViewModel	CustomerPresenter	…
…	…	…	…	…	…	…

Plus: you can have specialized variations of these classes, for intance: OrderEditPresenter, SubscriptionProductValidator.

This is a very maintainable structure, because everybody that understands the system of organization, can find the code exactly where he would expect it. A change has low impact, because it can at most impact code that references that specific class. Code is better reusable and recombinable. E.g. you can reuse the same validations in multiple places or use specific validations in specific situations.

Using this split up into classes could impact data integrity, since every class can be independently used, and there is not one thing that guards all the rules. The solution is to use facades that guard (most of) the integrity rules by delegating to smaller business logic objects. The separate concern itself is actually better guarded, since a small class does not get entangled with other code, because it handled totally separately.

Assemblies

The separation into technical and functional concerns extends further than the class split-up. It can also be noticed in the assembly subdivision. Those also have a 2-dimensional separation of concerns:

These are the functional concerns:

JJ.Data.**Ordering**.NHibernate
JJ.Business.**Ordering**.Validation
JJ.Presentation.**Cms**
JJ.Presentation.**Cms**.Mvc

And these are the technical concerns:

JJ.**Data**.Ordering.**NHibernate**
JJ.**Business**.Ordering.**Validation**
JJ.**Presentation**.Cms
JJ.**Presentation**.Cms.**Mvc**
JJ.**Presentation**.Cms.**Mobile**

The assemblies are split up by functional domains.
There are separate assemblies for presentation, business and other main layers.
There are separate assemblies for specific protocols and technologies.

It is clear from the assembly name which technique is used, and what the functional domain is and the position within the software layers.

The result of this split up is that we are not stuck with a 1-to-1 relation between an application and its platform.

For instance: if an Ordering back-end was programmed to use a very specific persistence technology, you might only use it with NHibernate and an SQL Server database. You could not use the entity model on a platform that does not support this (e.g. mobile platforms). If a Cms front-end is programmed to specifically use MVC, it can only be deployed as a web site and not as a Windows application or mobile app.

By further splitting up our assemblies we can reuse the Ordering back-end in multiple front-ends. Furthermore: a single front-end could be deployed to either web or mobile platform and we can store entity models differently depending on the infrastructural context. On a mobile platform we might store an entity model in XML, while in a web environment we might store things in SQL Service using NHibernate.

Framework Assemblies

- <TODO: Describe: Split up in Framework assemblies: when a framework assembly extends a .NET API, make a different framework assembly for each part of .NET you extend. A framework API should only hook into one specific part of .NET. That way we have control over what parts of .NET we make our application dependent on, which makes it possible to develop for multiple platforms.>

Combination of Concerns

After separating all different aspects of both functionality and technique, you can recombine these separated aspects in specific spots in the code: facades or presenters or in very specific classes that are a machine to perform very specific functionalities in a completely controlled way. See 'Facades' under 'Aspects ' and 'Manager / Facade' under 'Patterns'.

2 API's for the Same Thing (bad)

Choose one API and stick with it. It is not recommended to use e.g. two different XML API's in your application.

Copy-Paste Programming (bad)

Also called code duplication, copy-paste programming is the practice of implementing new functionality by copying a old code and then slightly changing it. This is very bad practice and creates a lot of slightly different copies of code, that make it difficult to change their mutual functionality. The alternative is to create one generic piece of code that can be used in multiple different ways and not repeat it (see also: 'Do not repeat yourself (DRY)').

Do note that you are allowed to repeat trivial code and two copies of code that must not affect eachother if one of them gets changed.

Do Not Repeat Yourself (DRY)

This is the principe that you should not repeat code. It is a general rule that you must then put the shared code in a separate class or method, and reuse the same code in multiple places.

But keep in mind that there are exceptions. Very trivial things can be repeated, and the same code might be repeated in placed where a change to one copy of the code, should not affect other parts of the system.

Dump Code Line Here (bad)

<TODO: Strategic mistake: Don't just put a line of code somewhere that gets rid of the symptom. Ask yourself the question if it is the responsibility of that class or whether it really belongs elsewhere. TODO: Describe that in more detail.>

God-Object (bad)

<TODO: Find synonym.>

An object that is used everywhere and can do anything. Consider splitting it up into multiple classes and only using the classes you need where you need them. It is an indication of bad separation of concerns.

God Base Class (bad)

<TODO: Find synonym.>

A base that that is a union of the functionality needed in the derived classes, instead of containing just the basic functionality.

Consider moving code to the derived classes that need it, delegating to different helper classes instead of putting everything in the base class, or as a last resort add intermediate inheritance levels, gradually extending functionality.

<TODO: Rephrase this and make it part of the main text: You should use inheritance to share behavior / private implementation, not as a way make methods available from multiple places, or to give a generalized name to a set of types, even though they still have separate unique behavior. You should also not create a base class, that is about a side-issue, because then you have reserved your one inheritance slot with something unimportant. >

Granularity

Granularity can be compared to sand. Large pebbles are large granules, while fine sand is made up of a small granules. In code it means that a piece of code might tap into a big object, while it really only needs to depend on a smaller object. A piece of code may use a specialized object, while it can tap into a more generalized form. It can also be expressed as 'defined at the wrong level'.

Helperitis (bad)

Helpers are static methods with static functions that support a specific aspect of programming for which no more than a flat list of methods is required.

Helpers are ofcourse helpful, but sometimes you can end up with code in which everything is delegates to helpers, obscuring what is actually going on.

Spread Responsibility (bad)

Keep the responsibility for one thing in one piece of code and do not spread it across multiple pieces of code. E.g. when setting defaults for a new object, try to keep that in one spot in the code.
(Not to be confused with the 'single responsibility principle'.)

Too Many Responsibilities (bad)

When a class does too many different things, you might want to split it up into separate individual classes.

When a method does different things depending on its input, you might want to split it up into different individual methods. Most of the time in the code calling your method, it is already clear which of those different things are needed there, and it does not need to be generic.

Two Solutions for the Same Thing (bad)

If you see that a single problem area has two different ways of handling it, this is an indication that your design lacks elegance and you should go look for a single solution to solve the problem. Two solutions is maintenance overhead and it has the same downsides as other types of code duplication.

Vague Justification (bad)

<TODO: Describe this. Do not try to justify putting a piece of code in a spot where it does not seem to belong with some vague explanation. The separation of responsibilities in a system should be crystal clear. People have a tendency to look for vague reasoning that justifies what they just did, and will take that very far. It is called rationalization. Be critical. Can you clearly explain why some piece of code belongs there? If not? Where should it belong? Try the stategy of 'Ideal Solution'. (There is a topic for that in this document.) That could help you figure out a better separation of concerns.>

Error Checking

<TODO: Write an intro on the kind of topics you will find here.>

Be Stict

<TODO: Describe the 'be strict' concept.>

Error Hiding / Null-Tollerance (bad)

A lot of code contains too much null-tollerance, possibly because of being paranoid about getting exceptions. This is the wrong way to go. You MUST throw an exception. If you build in a lot of null-tollerance you will run into the problem that 'nothing happened, and we have no error message'. Or 'the data is corrupted and we have no error message'. What otherwise would have been a clear error message just turned into a horrible problem to solve, in the worst case we will not even be able to solve it at all.

It can also result in a lot of complex code that tries to recover from a faulty situation, that should never occur in the first place and really should result in an error.

If something is null, that should not be null, an exception MUST be thrown. You have to make your code strict when it comes to faulty data and throw an exception when it is encountered. Exceptions are there to tell us what's wrong.

Null-Checks

<TODO: The rule 'no null checks on lists' is not true for the canonical model. >
<TODO: If GetByID returns null and you expected something, throw EntityNotFoundException<T>(id).>
<TODO: null-checks on parameters that the class uses directly, not if it is only passed to a method that does a proper null-check itself.>

The most common programming error is a missing null-check.

Code should be strict when it comes to nullability. The general message is: check for null if you expect it. But sometimes you can omit the null-checks to keep your code clean. Here are the rules:

- Entity models are anti-encapsulated: none of the data is protected.
- A data store often guards nullability.
- Validators should also guard nullability.
- Those two things determine whether an entity property is not nullable.
- You might mark the entity property with the word 'not nullable' in its summary.
- For not nullable properties you never have to do null-checks in your business logic, even though in theory null could be assigned.
- List-properties in the entity models require no null checks at all. They should be created in the constructor of the entity class and we simply assume nobody will assign null to it.
- This means that serious business logic should not be executed on entities that have not been validated yet. When you just retrieved entities from the data store, you may assume the data is valid.
- This saves us a lot of null-checks, which makes code more readable.
- The other entity properties are considered nullable.
- For nullable properties, business logic must have an alternative flow. Some business logic could throw an exception if a null is encountered. Other business logic might have to be null-tollerant and skip certain things. (Reflect this in the code by using words like 'Try' and 'IfNeeded'.)
- Null-checks can be omitted if you know that a variable was verified before. For instance: if you throw an exception in the constructor in case an argument is null, you can leave out null-checks in the rest of your class.

See also: 'Error hiding'.

Allow nulls as little as possible. Similar rules also apply to other integrity constraints (e.g. "> 0"), but null-checks are the most common.

Here are rules for null-checks for other constructs:

DTO's	Usually the same rules apply to DTO's as do for entities. Especially if they just transfer data from SQL statements to application logic.
Strings	To check if a string is filled in, use IsNullOrEmpty: if (String.IsNullOrEmpty(str)) throw new NullOrEmptyException(() => str); So this is wrong: if (str == null) throw new NullException(() => str); Another common mistake is this: obj.ToString() This will crash if the object can be null. This is the solution: Convert.ToString(obj)

Value types	Value types, for instance int and decimal, cannot be null unless they are nullable, for instance 'int?' and 'decimal?', so do not execute null-checks on non-nullable value types.
Parameters	Execute null-checks on arguments of public methods. Use NullException (out of Framework.Reflection). You can omit null-checks on arguments of private methods, if the rest of the class already guarantees it is not null. Avoid allowing null parameters. But if a parameter *is* nullable, you can denote this in several ways. - Assign null as the default value of the parameter, so it is clear that it can accept null: `private void MyMethod(object myParameter = null)` - Document with XML tags, that it is nullable: `/// <param name="myParameter ">nullable</param>` `private void MyMethod(object myParameter)` - Add an overload that does not have the parameter: `private void MyMethod() { }` `private void MyMethod(object myParameter) { }` Ideally in the overload with the parameter, do a null-check.
ViewModels	ViewModels that are passed to Presenters may contain nulls. You can use the NullCoalesce pattern to resolve the nulls before processing the view model object, so that null-checks can be omitted from the rest of the code.
Our own framework API's	For API's in our own framework you can count on an object when you call a Get method. You have to take null into consideration when you call TryGet.
Your own application code	Conform to that same pattern in your own application code, so you know when you can expect null.
Third-party API's	Some .NET API's and third party API's may return null when you call a Get method. Some do not. You have to learn which methods can return null and do null-checks appropriately.

Alternatives

- Microsoft's C# Language Design team is currently (2017-06-28) working hard at coming up with compile-time analyisis of nullability problems, that may make null-checks a thing of the past. They are not finished with it yet, and we have to wait and see what the outcome will be.

Process that Checks Itself (generally bad)

It is an anti-pattern for a process that just ran, to check if the data it wrote was correct. Why would it check what it just did? Shouldn't the code just be correct? Those are questions you want to ask yourself. Also, perhaps it should just be unit tested, or functionally tested instead.

It is like you are writing the same logical steps twice: once for writing the data, once for checking the data. The effort put into the complexity of the error checking code, would have been better spent making the actual processing code work well. Also: the error checking code might incur an extra maintenance burdon; you just have more code to keep working as the system evolves. It would also have performance overhead. You spread the responsibility of one thing over two different pieces of code. Another problem that could occur is that you might be adapting one piece of code and then you get false error messages, because you forgot that you still had that other piece of code to adapt, that does almost the same thing.

In certain cases with very sensitive, error-prone code you could opt for the process to verify its own data afterwards, but this should be the exception rather than the rule.

Reject, Don't Correct

Do not correct input data, but require that input data is correctly entered. Code that creates tollerance towards user entry errors can quickly get out of hand, while simply rejecting the use input with a validation message would suffice. It also gives the user more control over what happens, instead of the system's wrongly interpreting the user input.

Interfacing

<TODO: Write an intro on the kind of topics you will find here.>

'All' and 'Many'

The word '`All`' is often misused in repository method names. Once just a selection is returned, it is not '`All`' anymore. Use the word '`Many`'. So not `GetAllBySearchText`. Instead use `GetManyBySearchText`.

<TODO: If multiple items are returned by for instance repository methods, remember that the multiplicity should be reflected in the method name. For instance a method GetByCritia could be intended to return a list, but you cannot really see it from the name. One could assume it returns a single item. You could for instance call it GetManyByCriteria in that case, so you can see from the name that can return more than one. There are other ways to express multiplicity, such as the word 'List' or 'Collection' or a plural name, but as long as it is clear.>

Blackboxing and Whiteboxing

Blackboxing means that you put something in a little machine, something comes out, but you cannot see how it was processed exactly.

Blackboxing is the use of encapsulation to hide complexity. It is also a concept of creating an interface in front of an implementation: hiding what is exactly done.

Blackboxing can create a lower degree of dependence between different parts of a system, making things easier to change. It is also present in the concept of interfaces, allowing you to hide multiple implementations behind a similar interface.

Sometimes blackboxing creates problems, because not seeing exactly what is going on is a big downside. Being open about what happens could give a programmer the overview and control he needs. This is the concept of whiteboxing.

Both concepts are important and have their place. Whether blackboxing or whiteboxing is the right way to go, should be evaluated on a case-by-case basis.

Simply giving something a name that reveals its inner workings is a common form of whiteboxing.

Clarity over Brevitity

A longer name in code is better than a short, inspecific one. Even through you may think brevity supports readability, if it creates ambiguity, a longer, unambiguous name usually works out better.

For entity models, consider the name Order.OrderProducts, not Order.Products if the entity types are Order and OrderProduct, even though the first part of the expression 'Order.Products' already seems to imply it that it would be an OrderProduct. Because next to an OrderProduct entity, the model probably also has a Product entity and it would be very confusing that Order.Products would be a list of OrderProducts, as you would sooner think it is a list of Products from the name. Also it makes it harder to 'guess' what an entity model property is, if you abbreviate the names. Just use the full entity type name for property names and it will be far less confusing, especially to the ones that did not program your model. Again: yes, even when it seems obvious to *you*.

Conceptual Names (bad)

'Conceptual names' are bad practice. It is hard to define what that means. But it has to do with the names not being specific enough or only vaguely related to what it is really about.

Often combining a domain term with a design pattern gives you a more specific name:

```
CustomerViewModel
```

This instead of calling it just `Customer` or just `ViewModel`.

Here are a few more examples:

- `CustomerListReload`
    - It was a controller action name intended to be a AJAX variation of the Index action. Not only should `CustomerList` be replaced with `Index`, but also the word `Reload` is not clear. It may have something to do with reloading some piece of index, but it really is the AJAX variation of Index, so perhaps a suffix `Ajax` would be more appropriate. `IndexAjax` would have been a better name.
- A class name `Cooking` is also a good example of an conceptual name. Cooking? If you have to ask 'What about it?', you got a conceptual name, that should be made more specific.
- A view named `_CollectionListAction.cshtml`: The name `Action` is a conceptual name. It has something to do with an `Action`. More specifically: multiple actions, and more specifically: it is an `ActionBar`. The word `Action` is too general. It can refer to a Controller Action, the .NET Action<T> class, etc. Perhaps `_IndexActionBar.cshtml` would have been better.
- Conceptual names are also ones which do not include the pattern name at the end.

CRUD

<TODO: Explain.>

Delegitis (bad)

This is over-use of delegates, mostly as parameters. Here is an example of a method with too many delegates passed to it. The problem is that it is very unreadable:

```
public void MyDelegator(
    Func<Child, Parent> getParent, Action<Child, Parent> setParent,
    string parentPropertyName, string listPropertyName,
    Func<Parent> createParent, Func<Child> createChild)
{
    // ...
}
```

Solution: create a base class and a derived class that overrides the methods with specific implementations. This may create more classes, but it might be better overviewable.

```
abstract class MyBase
{
    private string _propertyName;
    private string _listPropertyName;

    public MyBase(string propertyName, string listPropertyName)
    {
        // ...
    }

    public void Execute()
    {
        // ...
    }

    protected abstract Parent GetParent(Child child);
    protected abstract void SetParent(Child child, Parent parent);
    protected abstract Parent CreateParent();
    protected abstract Child CreateChild();
}
```

Dependency Injection

<TODO: Change your view on dependency injection and rewrite that section. This is because SimpleInjector takes away a lot of drawbacks that Ninject has.>

For dependency injection we will not use frameworks like Ninject anymore. Ninject uses a 'magic hat' principle: an object came from somewhere and you have no idea where it came from or if the object is even there. NInject allows you to define a set of implementations of several interfaces centrally and retrieve that implementation from arbitrary places in the code:

```
// Bind it
Bind<IMyDependency>().To<MyDependency>();

// Use it
MyClass obj = new MyClass();

class MyClass
{
    private IMyDependency _myDependency;
    public MyClass()
    {
        myDependency = ServiceLocator.Resolve<IMyDependency>()
    }
}
```

We will not use this technique anymore, because it has serious disadvantages and the object oriented paradigm completely falls apart:

- You get null-reference exceptions, since there are no guarantees that the dependency is actually there.
- The setup of dependencies is tricky, because you would need to analyse all the code to actually know which dependencies you need to set up, or use an example, which is probably different for your new application.
- You get problems with using multiple instances of the dependency, and it is really hard to figure out where your object came from.
- It takes considerable hours of trouble shooting to set it up or to solve problems and you have a sense of having no control over what is going on.
- Some variations on the technique even require making members public that really need to be private.
- Using constructor arguments in the dependencies is not type safe.
- It is unclear from an class's members and constructors that it is dependent on something, what it is dependent on.
- It creates spaghetti code where everything is potentially dependent on everything else.
- It gets worse because injected dependencies can be dependent on yet again more injected dependencies.
- Instantiation is slower, because it has to go through a framework.
- Also, dependency injection needs a framework that might not be supported on all platforms.

Here is the proper alternative:

```
IMyDependency myDependency = new MyDependency(); // Or another means of
instantiation.
MyClass obj = new MyClass(myDependency);

class MyClass
{
    private IMyDependency _myDependency;

    public MyClass(IMyDependency myDependency)
    {
        if (myDependency == null) throw new NullException(() => myDependency);
        _myDependency = myDependency;
    }
}
```

It is simply a combination of interfaces and constructor arguments.

Now we have accomplished the same thing, only instantiation is explicit and not magic. You know you need to create the dependency (with dependency injection you did not). The dependency is null-checked (with dependency injection it was not). You can use multiple instances in the right places (with dependency injection multiple instantiation is tricky and has limited capabilities). It is faster, because it does not go through a framework and you do not need to include a framework, that might not work on all platforms.

Entity Design

< Entity design:

- (Dutch) Aangeven dat het beperken van tabellen belangrijker is dan constraints bewaken op database niveau omdat alles wat je in de database structuur aanmaakt, daar kom je 'nooit meer' vanaf en alles wat je in business logic oplost is makkelijker aan te passen.
- Keep models simple clean and stripped of all accessories, in particular entity models and especially canonical models. >

Execution Order Dependence (bad)

If methods only work if you execute them in a particular order, why not have one method that executes them in that specific order?

Handy Extras / Ya Ain't Gonna Need It (bad)

Do not add things to you code (and in particular to interfaces) 'that might be handy for the future'. The opposite is true. Extra code requires maintenance in case of changes. Also: If programmers use these 'handy things', they will be hard to get rid of and then you are stuck with it.

It is better to keep the code minimalistic and add the extras at the time that you actually need them.

Specialized case: Overloads that are never used, should be removed from the code.

Hatch / 'Doorgeefluik' (generally bad)

A method, that does not do anything but delegate to another method. For example: let's say there is a method GetImage in both an ImageRepository and an ImageManager. All ImageManager.GetImage does is call ImageRepository.GetImage.
The thinking error might be that you want to consistently call the ImageManager for everything and that it is a good preparation for the future, because the Manager might add extra rules later.

But it usually a better plan to directly call ImageRepository.GetImage and leave out the method ImageManager.GetImage. If you leave in the method that does nothing, then when a deeper layer changes, you'd have to change a lot of pointless layers above it. Also by adding a method to the Manager class, you create the false illusion, that more is done than just retrieving an image, giving you a lessened sense control what is going on.

If you see a method that does nothing but delegate to another method you have to consider removing this method.

However, you could also consider that in this case maybe the hatch is a good thing. If the general rule is to always go through the manager / facade then a method in the manager or facade may be expected. For a simple get by ID you may be better off using the repository directly, otherwise you get depedencies on facades where you do not need them, and this due to the nature of facades, which can do anything, automatically creates a large degree of dependency on many different parts of the code.

Hollow Interface

A hollow interface is and interface with many implementations in which many members are not even implemented or do not do anything. This is an indication that there is something wrong with its design. It violates the Liskov subtitution principle from the SOLID principles. You may want to split up into multiple interfaces so that the implementation you are making is not hollow and all interface members are properly implemented. Depending on your system, there could be a downside to having multiple interfaces, because it could harm how accessible your code is to others or how accessible it is to new implementors of your interface. (SOLID supporters may deny this with a vengeance.) You can solve this and make Liskov happy by introducing booleans to your interface saying whether a method is actually supported. See 'IsSupported' under 'Patterns'.

Interface Contamination

(Related to the Dependency Inversion Principle from SOLID.)

Interfaces are supposed to be lean. You should keep as much as possible out of an interface. An indication of an interface that is too rich, is for instance that a method has many parameters. You might see a method's name and expect it does not need all those parameters. That indicates a degree of interdependency that is too high. This method's responsibilities might have to be redistributed among different parts of the system. An interface might use a type from an API, while you might expect that interface to be API neutral. You might see a data class passed to an interface that you expected to be independent of that data model. These are all indications of interface contamination and the solution is usually to distribute responsibilities differently over different parts of the system.

Interface contamination becomes a big problem, when an interface is used in many different places. Then all those parts have a high degree of dependence on things they really have nothing to do with. It also makes interfaces difficult to implement, and poorly reusable.

Interface Neutrality

<Describe in general. Also refer to Interface Contamination and Interface Stability, leaky Abstractions>

Interface Stability

(Related to the Stable Abstraction Principle from SOLID.)

You can design the interfaces in such away that they do not tend to change much in the future.

This can be done by thinking about what the core of what you want is, and whether the interface is a dry enough representation of the kind of problems you want to solve with its implementations.

Not making assumptions about its use or its implementation can help.

Input/output transparency can both help and harm. By always passing input as parameters to the method, you take away assumptions about where that data came from, and makes it less hard to adapt when the data comes from elsewhere. But by passing input as parameters, you also increase the interface's awareness of things that should just be implementation details and makes it harder to make the interface that work when implementations change. Leaky abstractions are the worst example of this. It is an art. You cannot apply a single solution to all problems here.

Interface stability can also be improved by choosing neutral collection types over specific ones, for instance using IList<T> and not Dictionary<Something, Something>. This can have a small performance trade-off, but does improve interface stability. Again: it is a trade off, an art of picking the right tool for the job.

(Stable abstraction principle is also one of the 'secondary' SOLID principles. Although, I (Jan-Joost van Zon) personally find anything I can find on the topic vague about whether they mean stability as in 'does not change much', or stability as in 'does not have many bugs'. I also find things poorly explained, often written in a way that makes me feel you already need to understand the topic in order to understand the explanation, as if the author was just writing it for himself.)

IO Transparency

Input / ouput transparency.
<TODO: Describe. >

Kama-Sutra Pattern (bad)

So many overloads you cannot see which to pick.

Leaky Abstractions (bad)

(Related to the Dependency Inversion Principle from SOLID.)

An interface tries to abstract / generalize multiple similar problems into one solution. If an interface exposes the underlying solution, we speak of a leaky abstraction.

For example: this repository interface exposes the underlying NHibernate technology:

```
interface IMyRepository
{
    Entity TryGetByCriterion(NHibernate.Criterion.AbstractCriterion);
}
```

The repository interface should not have shown NHibernate-related types, because it is supposed to hide the underlying technology. You can also do too much with the interface now. AbstractCriterion allows you to build any query you want. That is also leaky about this interface. It is the repository's job is to offer a set of optimal queries. With the leaky interface above, a repository cannot do its job anymore.

The following example is better.

```
interface IMyRepository
```

```
    {
            Entity TryGetByCriteria(string name, DateTime dateCreated);
    }
}
```

You might add extra methods or parameters if more filtering options are needed.

Loose Coupling

Loose coupling or a low coupling is the concept of keeping a low degree of dependence between things. Tight coupling or high coupling means many things are tightly dependent on eachother, making it hard to change one of those things without breaking or changing the other things. The benefit of loose coupling is that a change to one thing affects a minimum of other things.

Low coupling does not mean that the total amount of links between things is lower. The amount of links between things may actually go up. The low degree of coupling is about an individual type's links to other types. That is what gives us the benefit of one change only affecting a limited set of other things.

One technique of limiting the degree of coupling is the use of interfaces. By letting code talk to an interface rather than directly to a specific implementation, this makes the code dependent on that interface without being directly dependent on multiple concrete classes. This makes you able to write one piece of code that handles multiple concrete things, which makes that code dependent on one type rather than the union of concrete implementations.

A symptom of high coupling is what happens if you hit Shift-F12 on a class name ('Find all references'). If you get a whole lot of results, you have a case of high coupling and you might be in trouble. If you get very little results it is a sign or low coupling and you can make a sigh of relief.

There are cases where high coupling is normal. For instance in case of base classes, combinator classes, framework classes, simple types, canonical models.

Lying Names

Names in code should tell the truth. If a method does more than what the name says, then it is a bad name. If a method name only gives an indication what it does, but in fact it does other stuff too, it is a bad name. If a programmer could not come up with a good name, so just typed something vague, then this is bad practice.

Magic (bad)

When a call to a member does more than you would expect and has unintuitive side effects that you do not see.

Solution: execute the side effects explicitly, so you see what is going on, instead of letting the side effects go off automatically.

Magic Defaults (generally bad)

<TODO: Describe. >

Magic Numbers / Magic Strings (bad)

<TODO: Describe. >

Method Self-Sufficient

It can be better to repeat the same work in multiple methods, rather than do the shared work once and pass the result to multiple methods. This might harm performance, but this makes the methods self-sufficient.
It is worse for the method to only work if you do very specific work beforehand. It is usually better to then let each of the methods repeat the same work. Performance is the trade-off here, but it makes the methods more reliable. An alternative is to create an instantiatable class that does the shared work in the constructor to not repeat it in the instance methods.

Ripple-Effect

This is an analogy to throwing a rock in a pond. If you throw a rock in a pond, a wave is created that goes through the whole pond. In software programming, it means that if you change one piece of code it may require you to adapt many other pieces of code. It can be an indication of a bad design choice where too many things directly dependent on eachother. It is not always a bad design choice, but simply a hard to prevent high-impact change.

See from Name, not from Arguments

The name of a method should say what it does. It should not be inferred from the argument list what it does. You may think that if name not clear, see argument list, but having to analyse the arguments it valuable brain time that you loose, that can be prevented by a clearer method name. Closely related to the 'Toilet-Role Principle'.

Spooky Action (at a Distance) / Cause and Effect too Far Apart

<TODO: Describe>

Syntactic Sugar

Not always bad practice, but is can be a cause of confusion.

Syntactic sugar is creating a notation that does not add anything, other than a simpler notation.

Object initializers are an example of syntactic sugar. In C# 2.0 you had to initialize an object's properties as follows:

```
Cat cat = new Cat();
cat.Age = 7;
cat.Name = "Nala";
```

In later versions of C# you are able to do the same thing with the following code:

```
var cat = new Cat
{
Age = 7,
Name = "Nala"
};
```

These two pieces of code do exactly the same thing. The shorter notation introduced is 'syntactic sugar'.

You can also program syntactic sugar into your own code. You can do this by introducing shorter names or a shorter notation, helper classes or implicit conversion operators (which can be very confusing). This may create a concise notation, but breaks the rule 'clarity over brevity'. Often such notations lie a little about what is really going on. So use it sparsely and in most cases go for a more explicit notation.

(The object initializer notation above actually is the recommended notation, not undesirable syntactic sugar.)

Unclear Interfaces

Interfaces of classes, methods and other members must clearly show what the member will do. A method's name should say what it does. Preferrably, if the method needs input, it should be a parameter, if a method has output it should be a return value. If an input parameter has an invalid value assigned to it, you should get an exception. The interface should guide the programmer, so there is really only one way to do it, and not several incorrect ways to do it.

Examples of unclear interfaces are:

- A method's input parameters make the method do nothing, instead of throwing an exception.
- You can assign null to one of the parameters and the method will change its behavior completely. A better solution is to have two separate methods.
- The input of a method is a property, while the method is the only one using the property. A better solution is to remove the property, and pass it as a parameter.
- The output of a method is a property. It may be better to actually return the output, rather than assign a property. A programmer may not be able to guess that the effect of the method is that a property is assigned.
- A method does nothing unless a property is assigned.
- A property could have been a constructor argument. The class does nothing or throws exceptions unless the property is assigned, while really it could have been made a mandatory constructor parameter, so the object's state is valid right after construction.

The Unwritten Agreement ('het onderonsje') (bad)

Two seemingly independent pieces of code only work if one piece of code makes assumptions about the implementation of the other piece of code. Another example of a code smell, that points to an unwritten agreement, is when an interface has elements to it, that are not obviously necessary. For instance a Dictionary parameter where it does not seem to be needed, perhaps because the calling code just so happens to be using a dictionary. You should generally avoid such scenarios. Different pieces of code should be self-sufficient and non-assumptious, but ofcouse it can depend on other trade-offs, such as performance and readability.

<TODO: Consider some of these phrases: 'Onderonsje' anti-pattern: polluting the interface of a method or class with members that break a pattern or with non-neutral types such as dictionaries, simply to make something maybe more efficient or another purpose. Usually the pollution can be prevented, because you can just do it higher in the call stack, rather than just passing things on to other methods.>

Wrapperitis (bad)

Do not make a class that simply wraps another class with no specific reason at all. For instance wrapping an API into a class that supposedly makes it easier, but really adds nothing new to it. You may be better of directly working with the underlying classes that actually do stuff, instead of having wrapper classes that suggest that they add something, but really do not have any additional value. It is annoying when you have a whole lot of classes and many times you wonder "What does this do?" and the answer turns out to be "nothing really".

This is closely trelated to the 'Hatch' anti-pattern.

There can be a good reason to wrap something though: loose coupling and polymorphism: giving multiple things a mutual interface, while originally they did not have a common interface.

Variables and Parameters

<TODO: Write an intro on the kind of topics you will find here.>

Double Negatives (bad)

If you give a variable name the word 'not' in it, then your code is likely to be less readable, since you might get a lot of double negations like "!not" and such. It is usually a better idea to use the 'positive' name as the variable name.

High-Throughput (bad)

It is not recommended to let a parameter be both input and output. Usually it is a better plan to let the parameter be input, and not write to it, and to return new output. A parameter's being 'througput' should be an exception, rather than the rule. The reason is that it is often confusing to a programmer calling your method. You usually do not expect the data you pass to the method to get deformed.

Keep the Names Consistent

Use the same name for something everywhere. For instance if the business domain contains one name, do not reinvent a new name for it. Use the same name everywhere in class names, property names and variable names. Do not come up with a new name for things half way the code.

Here is a code example with a name that changes out of the blue:

```
int id = 1234;
string sourcePageName = PageService.GetPageName(id);
string destPageName = ConvertPageName(sourcePageName);

private string ConvertPageName(string sourceTitle)
{
   // …
}
```

`PageName` all of a sudden turned into `Title`. This confuses everybody. You might not think there is much of a difference in meaning, but for all I know it can be a totally different property and what could also be the case is that someone was mistaken in taking the wrong property. Keep it consistent.

Many Properties (generally bad)

It is not a good plan to substitute parameter passing with the use of properties or fields.

It is better to let methods use their own parameters and return values, than to let many methods simply control the same set of properties.

The reason for this is that preferring properties over parameters, you have less control over where the data comes from and goes to. You have no idea what the output of a method actually is, because it may change any of the properties.

Methods Instead of Parameters (good)

Sometimes you see a lot of parameters in a method, that require (a lot of) if's inside the method. This makes the use of the method harder, and the implementation too. Sometimes the solution is simple: make a separate method for every option. This can make the implementation so much simpler and the use of it too. You might thing this is less flexible, because then you have to know in advance which method to call and you cannot just call one method for different situations. But here's a secret: most of the times that is not a problem. Most of the times the programmer really only needs that one thing in that part of the code, not a single method full of feature switches.

<TODO: A good example would help.>

Pass on and Assign (generally bad)

Polluting a whole call stack of methods with variables only used to assign it to an object, that may just as well be assigned much higher in de call stack.

Returning a Parameter (bad)

Do not return a parameter again. If you write directly to a passed object, there is not need to return it again. So this is wrong:

```
A x = Bla(x);

private A Bla(A x)
{
   x.Y = 10;
   return x;
}
```

Do this instead:

```
Bla(x);

private void Bla(A x)
{
   x.Y = 10;
}
```

Returning a parameter suggests that new output is created and the parameter is not written to, while neither is true: the input and output are the same object and the input object is changed. This is very unintuitive.

Temporary Variables (good)

To clarify yourself in code, and to make expressions better readable. Use temporary variables just to give things a name in between.

For instance this:

```
bool isSameControllerAndAction = string.Equals(names.ControllerName, GetControllerName()) &&
                                 string.Equals(names.ActionName, sourceActionName);
if (isSameControllerAndAction)
{
    return View(names.ViewName, viewModel);
}
```

Is more readable than this:

```
if (string.Equals(names.ControllerName, GetControllerName()) &&
    string.Equals(names.ActionName, sourceActionName);)
{
    return View(names.ViewName, viewModel);
}
```

Because it clarifies your reasoning to the next programmer who reads it.
The problem with programming is not writing code, it is reading it.

Most of the time it is a good idea to first put a returned value in a variable before returning it, because this makes debugging easier, because you can inspect the value before returning it, instead of only being able to see the value several frames up the call stack to conclude that a must deeper, now out of sight method did not do its job. Then you are already not in the piece of code you were trying to debug anymore. From a performance point of view it does not matter, because if compiled for release, the compiler will optimize out such temporary variables:

```
public bool MustExecute(MyClass myParameter)
{
   bool mustExecute = myParameter.IsSpecial || myParameter.Items.Count > 3;
   return mustDoIt;
}
```

It is not black and white when to use temporary variables just for the sake of giving something a name. It is an art to make your code as readable as possible for the next person.

Do not worry about the performance implications of the extra variable. If a temporary variable is immediately used after it is assigned, it will be optimized by the compiler.

Unused Parameter (bad)

Remove parameters from methods if they are no longer used.

Variable not Declared Where it is Used (bad)

For instance: when a variable is used in one place and declare in a totally different place, you might want to move the variable closer to where it is used. This may apply to local variables. This may also apply to using local variables instead of fields.

Variables that Change Meaning (bad)

When you use a variable for one thing and later overwrite it with semantically something else, it can be confusing to someone reading your code. Consider using a second variable instead.

Method Bodies

<TODO: Write an intro on the kind of topics you will find here.>

Auto-Instatiation

Sometimes auto-instatiation on first use can be replaced by initializing a field in a constructor or type initializer. This performs better becasue it prevents the auto-instatiation 'if' and might make the field only initialize once in the lifetime of the app domain.

<TODO: Auto-instantiation variations: benefits and downsides.>

Constructor Calls an Overridable (bad)

Calling an overridable member in a base constructor breaks inheritance principles. It creates a chicken and egg problem. Fields in the derived class need to be initialized before running a method, but the field can only be initialized after the base constructor went off, which runs the method! See the following code:

```
abstract class MyBase
{
  public MyBase()
  {
    Execute();
  }

  protected abstract Execute();
}

class MyDerivedClass()
{
  private int _value;

  public MyDerivedClass(int value)
  {
    _value = value;
  }

  protected override Execute()
  {
   // PROBLEM: _value is not initialized yet!
  }
}
```

The solution is to make Execute() public and insist that that it is called explicitly in the code that creates the instance. Ofcourse if the method is not overridable it would be no problem, and if the method was not called in the base constructor it would be no problem, but calling an overridable member from the base class's constructor could mean trouble.

<TODO: Update the remark below. Another solution is to actually do all 'the work' in the constructor, instead of having a separate Execute method, which you could also document in the alternatives above.>

(Validation framework uses this anti-pattern however, because there is too much danger that someone forgets to call Execute. It uses a trick to be able to initialize the members anyway, but it is quite dirty.)

Cross-Referencing (generally bad)

<TODO: Practices: Cross-referencing prevention. Do not pass 2 arrays and process them side-by-side, but look for a 'singular form' to process and pass along a tuple. Even better: look for something you can execute onto each tuple element separately.>

Empty If-Block (generally bad)

Do not do this:

```
if (condition)
{
   <<No code>>
}
else
{
   <<Some code>>
}
```

It looks like you forgot to write code. It looks weird. Either use negation:

```
if (!condition)
{
   // (Some code)
}
```

Or do an early return:

```
if (condition)
{
   return;
}

// (Some code)
```

Foreach with i

<TODO: Describe: Patterns / code style: for int i met vervolgens het item direct eerst in een variabele.>

Last Loop Item

<TODO: Describe different ways of handling the last array item that you can think of, in cases where the last loop item needs to be handled a little bit different from the others.>

Method too Long / Class too Long

If a method is long, for instance 25+ code lines, consider if it should be split up into multiple methods.
If a class is long, for instance 800+ code lines, consider if it should be split up into multiple classes.

In both cases this usually means that the method or class has too many responsibilities.

This code smell can also apply to other code than C# classes and methods.

Nested Loops (sometimes bad)

This is a nested loop:

```
foreach (var x in list1)
{
   foreach (var y in list2)
   {
      // ...
   }
}
```

Nested loops usually come with a performance penalty, because compared to a single loop with *n* iterations it might have n^2 iterations. It is not always wrong to have a loop in a loop, but you are only comparing two lists, using a hashset or dictionary might be a better solution, changing the n^2 problem back to a *2n* problem:

```
Dictionary<int, X> dictionary = list1.ToDictionary(x => x.ID);

foreach (var y in list2)
{
        var x = dictionary[y.ID];
        // ...
}
```

Nesting too Deep

Too much nesting in code can be confusing. It can be prevented by splitting the code up into multiple methods. It can also be prevented by using early returns. So instead of:

```
if (condition)
{
   // A lot of code…
   // …
   // …
   // …
}
else
{
   validationMessages = …;
}
```

You could do the following:

```
if (!condition)
{
  validationMessages = …;
  return;
}
```

```
// A lot of code
// …
// …
// …
```

This keeps cause and effect closer together, making alternative flows in code less confusing.

Toilet-Role Principle

Someone reading you code will look at it through the hole of a toilet-role, seeing only a small piece of code at a time. This means smaller pieces of code must make sense on their own. Someone maintaining or correcting your code should not first need to understand 1000'nds of lines of code before being able to correct a minor problem. There are many coding style tricks and design techniques to support this goal, that are talked about in this documentation.

Problem Solving

Core of the Problem

<TODO: Write text. Incorporate:
Solve the root of the problem. Do not work around a problem, because it will bite you in the ass later, very soon.>

Hit F5 and See

Hitting F5 and getting rid of symptoms. This is a strategy, where rather than trying to solve all edge cases in your head or on paper, you start debugging the code and tack the problems as you go.

Use Diagnostics / Improve Diagnostics

If you are faced with a problem and you do not really know what the cause is and trying to *Narrow the Scope* is not getting you anywhere, you can also adopt a strategy of improving diagnostics.

You could add some *Logging* or improve *DebuggerDisplays.* You could also improve *Error Checking*. You can also improve user input validation, which could give you a clue as to what's wrong.

(*DebuggerDisplays* are handy things that can also improve your debugging experience.)

You could also try and look for the already existing diagnostics to help you. Are the loggings, is there something in the windows event log. Can I see error messages in Windows Task Scheduler. Can you get a clearer error message?

Narrow the Scope

If faced with a problem you do not know the cause of and looking around, guessing and reasoning does not seem to get you any further, one strategy you can adopt is to narrow the scope. This means that you go through a process of eliminating possible causes. This can take the form of zeroing in on the specific piece of data that makes it go wrong. Another way is to try and exclude parts of the code that could cause the problem.

Another strategy that you can combine this with is making a *Smaller Test*.

Refactor to Solve

<TODO: Explain how refactoring code can actually help you solve a bug.>

Reproduce the Bug

<TODO: Write something about it. Look if other texts are present already in this document and use them / move them.>

Smaller Test

You could be faced with a problem and not be exactly sure what the cause is, but you do have some sort of clue. You could try and find out using the original data, where the problem emerged. But that situation can have so many other variables influencing the outcome, that it may be a good strategy to make a smaller test, with less data and less code involved.

This only really works if you already have some sort of clue as to where the problem lies. A smaller test is a good step after you have narrowed the scope already and want to test if one of your remaining hypotheses is true.

Even if your hypothesis does not turn out to be true, this is still useful information, because it further excludes possible causes, which narrows the scope of the problem. You can then say: “Well we can know for sure, it's not that.”

Strategy

<TODO: Write an intro on the kind of topics you will find here.>

Abstract / Concrete

Abstract means that instead of referring to specific items in a set, you refer to a set of items by stating what they have in common. Concrete means talking about a very specific item in the a set.

In software programming abstraction means you can generalize multiple problems and offer a single solution for it.

<TODO: Describe this well: There is a second meaning of abstract, that also has a place is software development: leaving out details. In software development in particular: making something out of smaller building blocks and hide the details behind a simpler view on it.>

It is an art to pick when to abstract problems or when to handle a concrete problem. Both have their benefits. In one case abstraction may prevent complexity, while in another case being specific prevents complexity.

A concrete problem is easier to work out, better to understand, and can tap into specific requirements. On the other hand, abstracting a problem makes you able to write code once and apply it to many different situations. It is harder to write, but might be more maintainable, because it is much less code.

Analysis Paralysis

It is good to think in advance about how you are going to do things. But there is a limit to how useful it is to spend more time preparing for the job, rather than just go ahead and do it. You can keep in mind that there is a balance to be found here.

Anti-programming

It seems more efficient to reuse many third party software components instead of programming them yourself.

However, your own code might actually be better than that of the third party component. Third party components often come with overhead and bugs and problems with integration so that a custom solution might actually be more efficient.

The choice for a third party component might be related to your own inability to program it. That is why it is anti-programming.

You always start out as an anti-programmer. However, as you progress, you might start getting better.

Do not fall into the trap of thinking that using a third party component is always more efficient than programming it yourself.

Asymmetry (bad)

When several pieces of code that do similar things are differently structured, this is an indication that you should make these pieces of code consistent. If these pieces of code do not have a similar structure, you should have a good reason for it and understand this reason and be able to explain it.

Even though this may seem a vague point, symmetry in code is very important for good software design.

Blind Faith Methodology (bad)

<TODO: Describe.>

Bottom-Up and Top-Down

Bottom-up design means you first design the lower layers of a system, for instance the data model and gradually work your way up to the front-end. You can also say bottom-up design is starting with the smaller parts and working your way up to creating bigger and bigger parts out of it.

Top-down design means you first desing the higher layers of the system, for instance the front-end and gradually work your way down to the data model the little details. You can also say top-down design is starting to think about the bigger parts first and then gradually working out smaller and smaller details.

No method is best. They are simply two different strategies to attack a problem.

Bug Solving

To solve a bug, first reproduce it.

<TODO: Make more extensive description.>

Cartesian Product of Features Problem

Say you have some behaviors that you want a class to either have or not have. What if you want some derived classes that either have or do not have that feature in it. Then you would get as many derived classes as 2 to the power of the number of features. If you have 4 features, you would need 24 = 16 derived classes, with each of the features either turned on or turned off. In cases like this it is hard to come up with a good inheritance structure, because neither feature builds on top of eachother. You could make class variations WithFeature1, WithoutFeature1, WithFeature2WithFeature1, WithFeature2WithoutFeature1. All very awkward. Arbitrarily Feature1 was picked to be more basic than Feature2. Also: you would have to repeat the code of Feature2 in two derived classes! Another alternative is also not so good: building a base class that simply has all features in it and derived classes having the feature either turned on or off. This would be called the 'god base class' anti-pattern. It would break the way you work with base classes, since base classes should be more basic with less features in it than derived classes; base classes should not have more features than derived classes.

It does not just apply to turning features on and off. If you have 4 variation on a feature and you want to combine it with one out of 4 variations of another feature, then you have 42 base classes and which feature will be in the deeper base class? It is the same situation as the problem as described above.

This is a weaknesses of inheritance, that makes it so that inheritance should not always be your first choice in constructs to solve your problem.

The Inheritance-Helper pattern may solve some of the issues.

Chicken and Egg

<TODO: Describe.>

Consistent Stupidity

Too much effort into making code consistent can result in nonsensical code when looking at the individual cases. In a worst case scenario it even results in code that works incorrectly.

It is better for each piece of code to make as much sense as possible individually.

"Consistent stupidity is still stupidity."

Delete + Insert != Update

If you want to update a set of records, you can sometimes get away with deleting all of them and then inserting the new ones again, but you have to realize that this is generally bad practice. You are better off checking if the item exists and then choose whether to insert or update the item based on that. The reason for this is that often the existing items are linked to by other items. If you bluntly remove them, those links will be corrupted. Even when items do not seem to have any links to them, it is still a better idea to check for existence, then insert or update, because of various reasons:

- It performs better if those statements will end up executed onto a database.
- You might link to those entities in the future, which makes your code prepared for that, just by following best practice.
- The entity might not be linked to by other entities, but the entity's ID may very well be present in URL's someone might send to someone in an e-mail.
- Also there could be links to an entity in the UI, even though there are not links to the entity from other entities.
- The entity might not be linked to through a foreign key but could be linked to externally by a loosely linked key. (It can be said that it is very easy to overlook that there are links to an entity.)

Hopefully this will give you an idea of how soon you run into problems if you pretend an update equals a deletion + an insertion and make you think twice and do it another way.

See also: Patterns, TryGet-Insert-Update.

Distortion (usually bad)

When you diverge from a pattern, you are probably not using it right. Find a way to keep the use of a pattern clean. It is an indication that your separation of concerns is not right or another design mistake. Perhaps you are using the wrong pattern, perhaps you are putting the responsibility for something in the wrong spot.

Do It Right, Or Don't Do It At All

This is a phrase that can help you prevent a mess of half-baked things that do not work well on their own, let along work well together. This strategy can also be applied to whole features, but also separate classes and methods. If you do not have the time to program it right, you have several options. For instance postpone until you do have time or find an alternative that offers limited capabilities, but still does what it says on the tin. Still better than half-baked stuff. Do not litter the code with all sorts of stuff that does not work.

Double Stitch

'A double stitch holds better.'

Combatting a problem by implementing security against it at multiple levels. Sometimes this protects against a problem better, but on the other hand, you introduce spread responsibility and potentially code repetition.

Here is an example. Say there is a process that writes away data. The process is in principle responsible for writing away correct data, but the validation part of the architecture may guard the overall rules. This means the intricacies around the correct data is noticable in both the process and the validation, so if one of these subsystems contains a programming error, the other subsystem acts as a fail safe.
If the correctness of data is described by trivial rules, you might not say there is a spread responsibility and rather call it a double-stitched solution. But if writing away correct data and checking the correctness of that data is very intricate, you may be repeating significant logic, which does lead to spread responsibility and code repetition. So even though that solution is safer because of double-stitchedness, it is unsafe due to excessive code repetition. This goes to show that it all depends on the situation what the best strategy is.

First Try Specific, Then Try Generic

It is often a good solution to design something generic, rather than something that only works for one specific situation. But sometimes it is hard to do this. A strategy can be that when you have the feeling a generic solution is appropriate, but you cannot figure it out, to first develop a specific solution, and then refactor it to become more general.

Fluff

A lot of code that does not add much functionality is an indication that a more elegant solution might be possible.

Ghost Hunt

Not so much programming practice, rather than a practical concern. Seeing problems where they do not exist or investigating a potential problem that might not be a problem at all. Early detection of seeing problems where there are none can prevent such excessive time spenditure of time.

GNUID (bad)

An anti-pattern where a GUID (Globally Unique Identifier) is used as an identifier, but is not globally unique: each occurrence of the same thing, for instance spread over different databases, will have a different ID, even though a Globally Unique Identifier has in its name that it should be the global identifier. This can easily result in programming errors, since one might make assumptions, that an object can be referenced by the same ID always, no matter what copy you are working on.

Hard-Coding and Soft-Coding

<TODO: Describe.>

Ideal Solution

It's good to have the ideal situation in your head and then come up with a less than perfect alternative that approaches that ideal more or less.

Inheritance not Always Good

<TODO: Point to other pieces of this document that explains this point and perhaps add a few more arguments to it.>

It Works, Doesn't It? ('Maar het werkt toch?') (bad)

This is the false conclusion that when the output is OK, it must mean that the program is coded well. This is often paired with the argument that additional coding work is a waste of time.

A senior programmer might tell a junior programmer to change his code, and a junior programmer might think it is a whole lot of unnecessary work, because his program already worked.

Things that could still be wrong with the program are for instance that:

- Poor performance
- Crashes in exceptional cases
- Corrupts data in exceptional cases

- Code not easy to understand. So that another programmer does not easily understand.
- Poor maintainability so that a change takes ages
- or possibly means it has to be rewritten completely
- or a change is error-prone.
- Not adapted to contextual changes such as culture or when variables change. They may be hard-codedly interwoven into the code.
- Assumptions that databases will be available when they are not.
- Security leaks
- Crashes when services are offline.
- Does not give validation or exception messages when something goes wrong.
- Does not integrate well with other technologies.
- Etcetera.

These are all examples of what could still be wrong with the code if a program 'seems to work'.

Another false argument against doing all this work is the claim that these are just unimportant details.

Lack of Choice = Guarantees

<TODO: Describe.>

Least Possible Effort Strategy (bad)

<TODO: Describe. Something like laziness may in the end cost you more, then it is not efficiency.>

Liskov-Substution Principle (SOLID)

<TODO: Describe my take and my solutions on it.>

Use It or Lose It

<TODO: Describe:
- Unused functions (bad)
- (Dutch) Ik heb de policy om niet gebruikte code weg te halen en pas als het echt opgelost moet worden opnieuw te bouwen, omdat de situatie dan weer dusdanig veranderd is en ideeën veranderd zijn, dat er toch niets over blijft van de oude oplossing, die dan in de tussentijd alleen maar in de weg staat en onderhoudslast tot gevolg heeft.
- It also never gets tested.
- Go into how the people might be afraid to throw away something valuable and that's why they keep unused stuff lying around. But counterintuitively it is more economic to just throw it away and rebuild it later.
- Things that are not used are also not tested very well and may well not work anymore when you take another look at it a year later. Also you could be releasing buggy functionality, that nobody tests, but you can still access the faulty functionality, so the customer finds this buggy thing in your programs. Not good>
- All of this can be such a hindrance, that it pays off better to just put that code in an Archive and just rebuild if ever needed again.
- Flip side: reusable framework components. There 'might use' is enough, especially if it makes a tool complete. But there it almost takes on a similar form as a user interface. The user interface should be complete and tested. For a programmer, its being in a user interface means it is in use. Whether a user will actually use and appreciate this function is up to other

people. If they do not use it, and will not ever use it or think it is unlikely they will use it in the foreseeable future, this can better be scratched too. So 'used' has more fine definition. Anyway, exposure through an API might be interpreted as 'used' code. Who is the judge? The 'user', which is not always the same people.>

One Extra Step

If you are done with a task, take one extra step to make it a little better. Often this is the time when you can make it much easier for the future with relatively little effort.

Open/Closed Principe (SOLID)

"Open for extension, closed for modification." is one of the SOLID software design principles.

This is my personal take on it, and I am very sceptical about this principle. And I am not alone. I think it is outdated, like the waterfall method, or only works in theory, but is not practical at all. It sounds like people that believe in this principle are blindly following some authority, without having an opinion of their own.

If you interpret this litterly it says that you cannot ever change code, which is something I am going to completely have to disagree with. That sounds something theoretical there will never work in practice and in its contains the assumption that code that is written, can be made perfect before moving on to something else. Or a mess of an inheritance hierarchy, or protected modifiers that will break encapsulation, just because someone is afraid to change his code. I use design principles so that my code is easy to change to new situations, so that interpretation does not fly with me.

But if interpreted more loosely, it could be something I can agree with:

- That you should not be required to change code, in order to use it.
- Or that a new version of a component should be compatible with existing software, that uses the component. It sounds to me like something you can work around if all of that client code is your own. And backward compatibility can be achieved with other techniques than not changing code.

There is much debate about how to interpret this design principle. It is like people try to give new explanations to this principle, because the original idea does not work in practice.

Open Ends (bad)

Or rather: not solving open ends is usually bad and will meet up with you in the near future.

Power of Abstraction / Power of Generalization

When you are able to generalize multiple problems into a single solution, you can code something once and solve multiple problems at the same time.

The other side of it is, that it is difficult to abstract multiple problems into a more general problem. Sometimes it is also difficult to understand the solution, because it requires the same abstract thinking.

However, by doing it you can save a lot of work and complexity.

Quick and Dirty / Dirty (usually bad)

<TODO: Describe.>

Readable, Writable and Rewritable Code

<TODO: Use this phrase: Code should be about expressing your intentions as much as instruct the computer what to do. >
<TODO: Incorporate this phrase: "the difficult part of programming is not writing code, but reading it.">

Code should be written so clearly that it is easy to read, which also makes it easy to adapt. It is more important that code is easy to read than easy to write. It is not the startup cost, but the maintenance cost that will kill you. Following good design principles, such as loose coupling, good naming, can make code adaptable. You should write code that you should not be afraid to change. You should not be afraid to change code or refactor. If you are afraid to change the code, then it might be poorly written and a digital time bomb. Time to change that. Or if you are afraid to change code, then there might be something wrong with the testing phase of your software development lifecycle, or lack thereof?

Reflection after a Task

After completing a task, take a critical look at your code and reflect on whether it is readable, what might still be wrong with it, what could be made better.

Subtractive and Additive

Also called 'inclusive' or 'exclusive'.

Subtractive or additive can be a strategy in programming. Subtractive starts with everything and then you start excluding things. Additive starts with nothing and then you start adding things.

An example is security. It is often better to use an additive approach and start with no user rights at all and gradually add rights.
Another example might be storing object structures. You might create a storage mechanism that stores everything unless you exclude something, or you might create a storage mechanism that stores nothing unless you explicitly specify it is included.

Testing

Some bad practices regarding testing:

- Not testing something
- No unit testing if favorable.

Too Difficult / Disproportional Effort

'Don't be too hard on yourself' principle / 'It can't be that hard' principle / 'It is not allowed to be hard'.

Allow yourself to admit, that implementing it a certain way is just going to be too difficult (for you). Look at it another way: if it is that difficult, perhaps there is a simpler solution, that you have overlooked. If a potential solution to a problem takes a lot of effort, be it noticed up front, or noticed while trying to implement, take a step back and look for simpler solutions.

This may help you keep an open mind for other solutions. Do not let this be an excuse for laziness. Just keep an open mind.

Those simpler solutions may come with limitations from a functional point of view, or require trade offs in other areas, but it may be worth the time you save. Do keep in mind that you do not just create more work in the future. If doing it 'wrong' now will give you an overload of work later, it does not fall into the category of 'less effort' or 'simpler solution' at all. That time in the future where it becomes a problem, is nearer than you think. It is always a gray area.

If things become difficult to implement, think back to the core of the problem and that if the problem sounds simple, the solution might be too. (Large gray area.)

When you code keeps producing errors when you make a change, this could be an indication that your solution is too difficulr.

Tooleritis

Too many tools / separate little programs to fix little things, rather than having a coherent system with all the needed capabilities. A lot of single-run, sometimes-run processes, started separately in separate little console and WinForms apps… while one coherent management application might be better. Tooleritis can also be the result of the system's having too many ifs, ands and buts.

Trade-Offs

Every technique in software development has pros and cons. It is the job of the software designer to weigh off all the pros and cons of every possible design choice and come up with a balance best suited to the situations, that will make us run into the least problems in the future.

A striking example is the principe of generalization which is good, and the principe of low coupling which is good.

There are cases where high coupling is normal. For instance in case of base classes, combinator classes, framework classes, simple types and canonical models, a high degree of coupling with these types can be expected.

Basically when you generalize and make something very reusable, you can automatically expect a high degree of coupling with it, because it is reused so often. That is another reason why generalized solutions should be of such high quality.

There are techniques that in general are bad, and techniques that in general are good, but there also good principles that contradict eachother, that need to be weighed off in every design decision.

The danger of such a large gray area, is that people thing they can just do whatever. But you should not do just whatever. You should learn the pros and cons of things and do a careful weigh-off every time. However, some things are generally bad and some things are generally good.

Unforeseen Problems

It is always harder than you think. The general gist of it might take you 30 minutes, but you can expect edge cases or rework to pop up that will make it 2 hours easily. You might encounter something you have less experience with, you name it. Do not stress out. This is part of the job.

Whirlpool Anti-Pattern / Inappropriate Conversions

The architecture contains multiple layers that require converting one type to another, for instance converting a view model to an entity. However, additional conversions such as converting one type of view model to another type of view model are not recommended.

<TODO: Describe that it is also called the Whirlpool anti-pattern. Related to Inappropriate conversions. It is when data get converted in one form to another to another to another with very little need, not even for abstraction layers. You could consider moving more of the conversion logic that is spread into a single place instead and refactor away some of the conversions. You could also consider that instead of converting from source to dest and then reprocessing dest and then reprocessing dest, you just convert source to multiple dest items, not relying on intermediate data transformations.>

SOLID

<TODO: Move all comparisons to SOLID to here, because I want to cover it once centrally.>
<TODO: write my take on it. Admit that it might be a single sided view and that you are open to different opinions. It will show you know them and thought about them.?
<TODO: Interface seggregation principle from SOLID.>