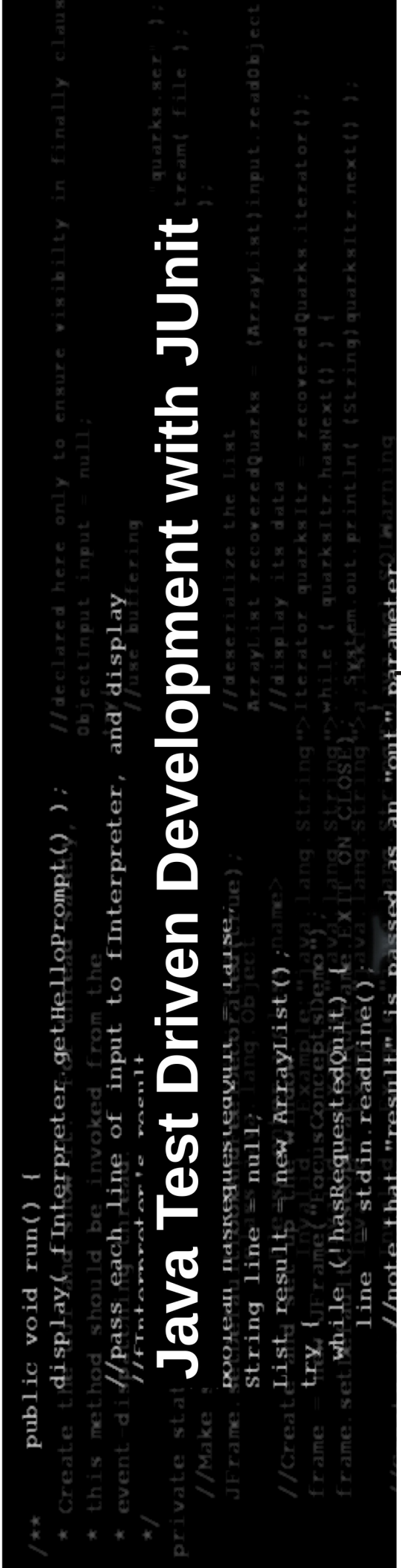


Module Seven

Refactoring

Controlling complexity is the essence of computer programming.

Brian Kernigan

Perfection (in design) is achieved not when there is nothing more to add, but rather when there is nothing more to take away

Antoine de Saint-Exupery

To change and to change for the better are two different things.

German Proverb

There are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies. The first method is far more difficult.

C.A.R Hoare

Why Refactoring Works

Kent Beck

Programs have two kinds of value: what they can do for you today and what they can do for you tomorrow. Most times when we are programming, we are focused on what we want the program to do today. Whether we are fixing a bug or adding a feature, we are making today's program more valuable by making it more capable.

You can't program long without realizing that what the system does today is only a part of the story. If you can get today's work done today, but you do it in such a way that you can't possibly get tomorrow's work done tomorrow, then you lose. Notice, though, that you know what you need to do today, but you're not quite sure about tomorrow. Maybe you'll do this, maybe that, maybe something you haven't imagined yet. I know enough to do today's work. I don't know enough to do tomorrow's.

But if I only work for today, I won't be able to work tomorrow at all. Refactoring is one way out of the bind. When you find that yesterday's decision doesn't make sense today, you change the decision. Now you can do today's work. Tomorrow, some of your understanding as of today will seem naive, so you'll change that, too.

What is it that makes programs hard to work with? Four things I can think of as I am typing this are as follows:

- Programs that are hard to read are hard to modify.
- Programs that have duplicated logic are hard to modify.
- Programs that require additional behavior that requires you to change running code are hard to modify.
- Programs with complex conditional logic are hard to modify.

So, we want programs that are easy to read, that have all logic specified in one and only one place, that do not allow changes to endanger existing behavior, and that allow conditional logic to be expressed as simply as possible.

Refactoring is the process of taking a running program and adding to its value, not by changing its behavior but by giving it more of these qualities that enable us to continue developing at speed.

Natural Course of Refactoring – a Refactoring Work-flow

Posted by Mariusz Sieraczkiewicz on Nov 30, 2014

https://www.infoq.com/articles/natural-course-refactoring

Current state

Refactoring is not a new technique yet still quite a hot topic. It became an indispensable tool in a programmer toolbox. At least in theory. In practice surprisingly I still see that this practice is abandoned in many teams. “We don’t have time”. “We are not allowed”. I think that one of the problems is a false dichotomy thinking here. Refactoring is not a zero-one decision: do it or not. This is why I differentiate two types of refactoring: everyday refactoring and strategic refactoring.

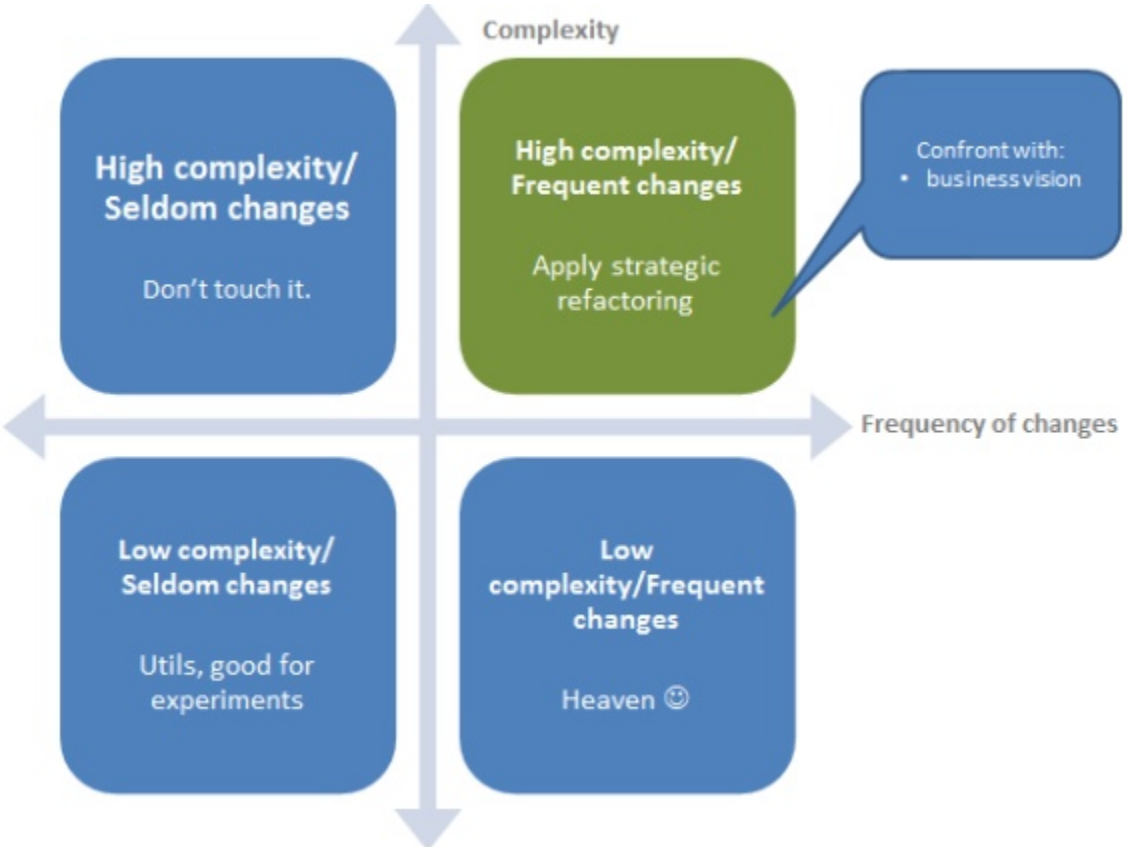
Everyday refactoring is within a reach of every programmer, can be done in minutes, these are mostly safe, IDE-base automatic refactorings, is done for local health of the code, it is a part of programming practice and there is no excuse for not doing it.

Strategic refactoring is a team longer term effort, requires aligning with iteration planning, generates items in backlog, is a risky activity that requires intensive testing (including good tests suite), is difficult and time-consuming. Check carefully if refactoring gives you enough value (eg. challenging with Feather’s Quadrant). A tool that can be helpful to evaluate the potential value of refactoring is Feather’s Quadrant (I am not sure if Michael knows that it was named after him).

In order to analyze your code you should consider two dimensions:

1. code complexity (cyclomatic complexity metric);
2. frequency of changes in repository.

Having so your code will be divided into four parts:



High complexity – seldom changes – in most cases it is messy but stable code of the core of the system, so don't touch it.

Low complexity – seldom changes – simple code not changing much, such as utils or not that often used functionality. Also no need to touch unless you want to do some safe to fail experiments.

Low complexity – frequent changes – if you have such code then you are in heaven, this is what you would desire – quite simple code that you deal with often.

High complexity – frequent changes – this is a place where evil lives, big classes, big methods that are changed often. There parts of code scream for refactoring.

But this kind of clustering has one drawback – it shows you the past, how it was in the past, so it doesn't guarantee that your guess will be correct. This is why you should challenge results of this heuristic with the business strategic vision – to find out what is likely to be changed in the future.

This is great mental tool but I haven't seen any production tool supporting this heuristic. On the other hand it shouldn't be difficult to write script getting this data for your environment.

Natural Course of Refactoring

It is time to get to the point. What is Natural Course of Refactoring? I am sure you are familiar with TDD process: red-green-refactor. It is a very simple process. You are right – practice is a little bit more difficult. But the process itself is easy to understand and use. The same is with Natural Course of Refactoring.

I have been working for years dealing with legacy code and doing "Refactoring to design patterns" trainings. During one of such consultancy projects I realized that there is some kind of pattern occurring very often, a process pattern. What you mostly do while doing refactoring: you try to understand the code, split methods into smaller pieces (smaller methods), then you move them around to find a better place for it using single responsibility response, you introduce patterns when some kind of flexibility is needed and evolve your architecture.

First solution

Let's assume that you work with code that is a mess. Spaghetti. Big ball of mud. You try to do something with one of those big methods. And this is the place where we start.

Step 0. Understand it

First do something to understand it. It is many times the most difficult step in the process. Especially when dealing with legacy code. How to understand a five hundred lines method having cyclomatic complexity higher 50? No matter what you do it is always pain. How to relieve this pain? Here you can find some tips.

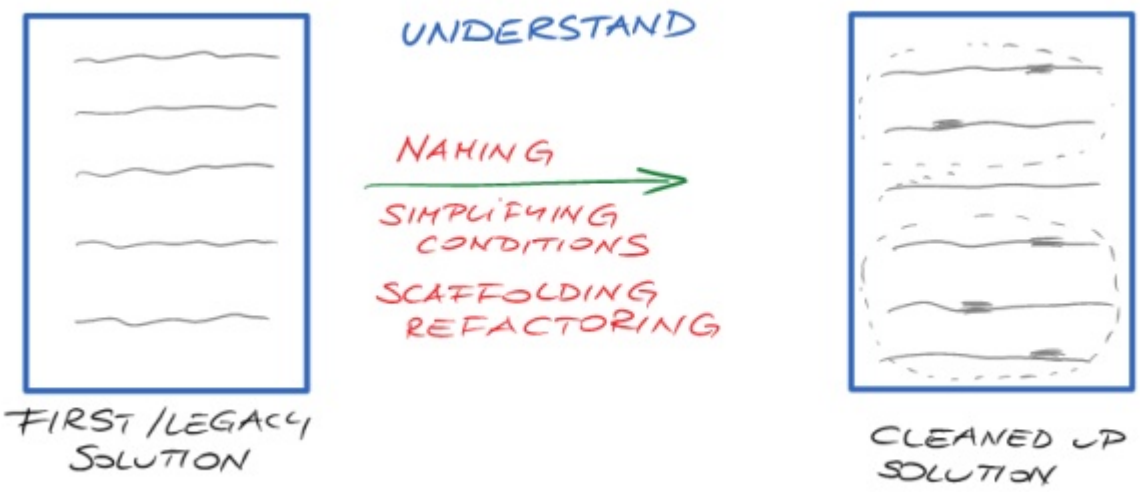
1. Find somebody who wrote the code or is familiar with it and ask for help. This would be the recommended solution but usually not doable. Sometime even the author is not able to explain the code.
2. Start reading and comment the code putting some extra tags.

   Reading a large piece of code is exhausting and tedious task. You have to process a lot of information and it is easy to get lost. You can find some helpful task tags which can be used for the knowledge gathered from the understanding process. Beware: these comments are temporal and should be removed right after refactoring is finished.

   Comments are particularly important in further steps of the Natural Course of Refactor-

ing and are used to extract steps of algorithm.

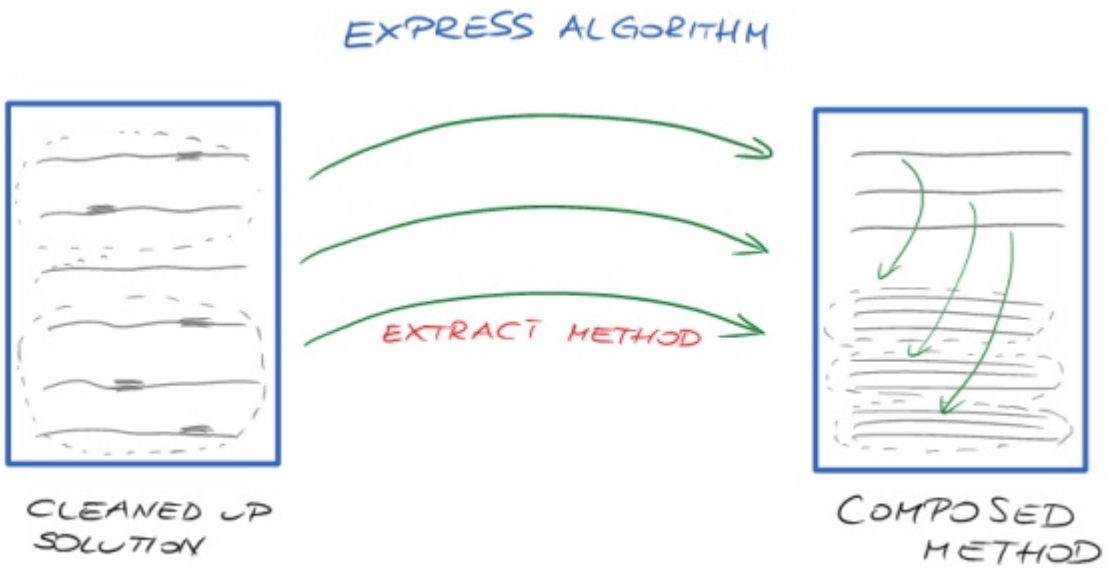
3. Do some sort of scratch refactoring – a technique described by Michael Feathers in his book “Working with legacy code”. It is a rough refactoring done just for better understanding the code and to be thrown away afterwards.
4. Rename variables and name the conditional to express the intent. Naming is a big power and it usually is the most common source of code smells. You can improve readability of code just changing names.



Step 1. Express algorithm

Every method should express its algorithm. And this algorithm is usually not expressed explicitly in badly written legacy code. Considering you have understood the method, now you should be able to split it into algorithm steps using mostly the Extract method refactoring in case of simpler methods and Introduce Method Object refactoring for more messy methods. Your aim is to get the code that speaks to you.

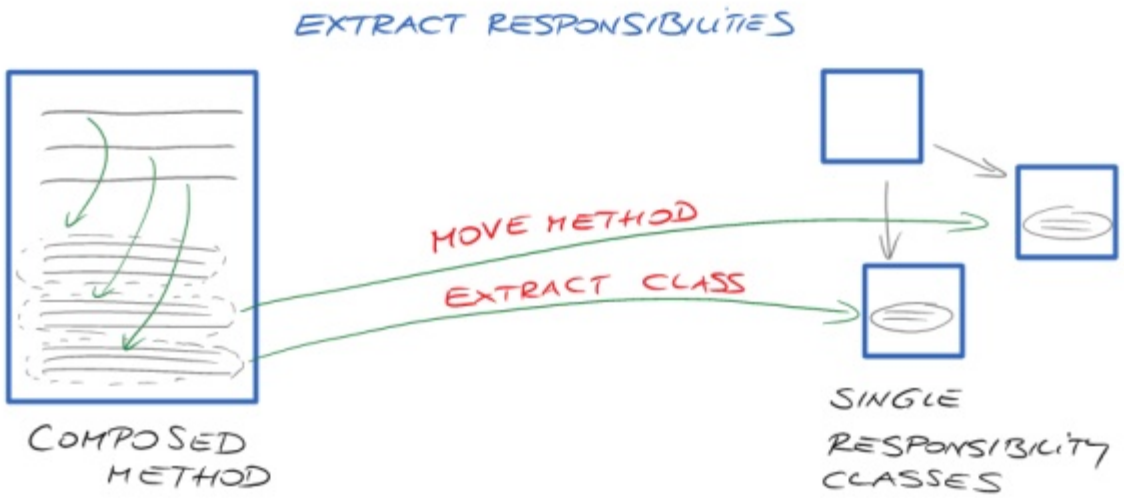
We can visualize this step this way:



This way we have algorithm expressed in the code. Just one look at the code and you can see the steps of the algorithm. Steps should be at the same level of abstraction. The state of the code we are aiming to is called Composed method.

Step 2. Extract responsibilities

After extracting many small methods it is quite likely that you have many of them in your class. It is quite likely, especially in legacy code, that some extracted methods are out of class responsibility. It is time to use Single Responsibity Principle here. Move those method to other classes or extract a new one if needed. In this step are aim is to adjust responsibilities and cut the code a little bit.



You may consider following strategies in this step:

- if there is a private method worth testing (ie. complicated enough) it is likely there is a better place for it in another class. In our case a good example is swapUpperAnd-LowercaseRandomly() method. Then move it or extract a new class for the method.
- check if the method name and the class name match. In our example method named isWord() doesn't match the class named TextObfuscator. It is better fit for TextPart class.
- use metrics indicating detecting class responsibility violations from LCOM family to find candidate methods to move outside.

After step two we should have more balanced responsibilities in our classes. Refactorings that may be helpful in this step: Move method, Extract class, Introduce Domain Object, Introduce Value Object.

Step 3. Introduce flexibility

Just doing steps 0-2 you would heal most codebases on this planet. But of course sometimes you should go further when flexibility is needed. Here are some tips indicating that you may introduce the design pattern:

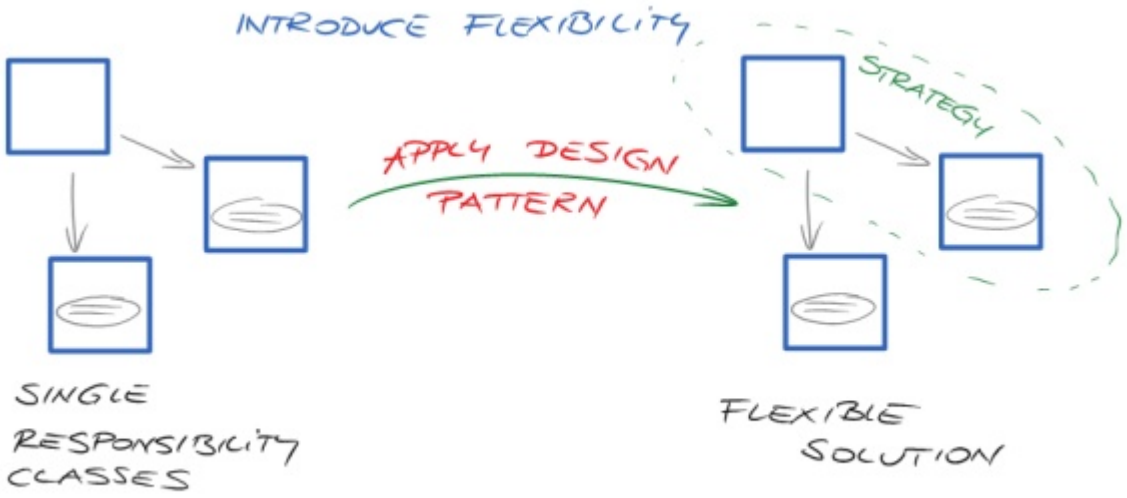
- you have implemented many times similar algorithm

- your requirements state that something may be done in a variety of ways (for example you have to support different file formats, data formats, protocols, third party systems etc.)
- after you more understand the bigger picture, you can recognize better a nature of object interactions and patterns emerge

It is very handy to know what kind you may be looking for:

- data are built in many steps (Builder)
- you domain object has its lifecycle and you can define some sort of state machine (State)
- you need some form of caching data (Flyweight)
- you need to flexibly enhance some behaviour (Decorator)
- you want to hide the real object for some reasons (Proxy)
- you have different variations on the same algorithm (Strategy)

and so on…



In most cases after this step your code would ready for extending (Open-Close principle embedded in many patterns). But beware. There are many situations when you can introduce pattern but there is no rationale to do it. Our example is the case. You can find Builder pattern introduced as an illustration of the next step of the process but let's think for a while. Do we really need it this kind of flexibility in this context? Builder pattern gives us means to build data in different ways but in our example there is only one way to obfuscate the text and we have no more hints that we have to be flexible here. Then don't do it. Flexibility has its cost. Don't pay for something you don't need.

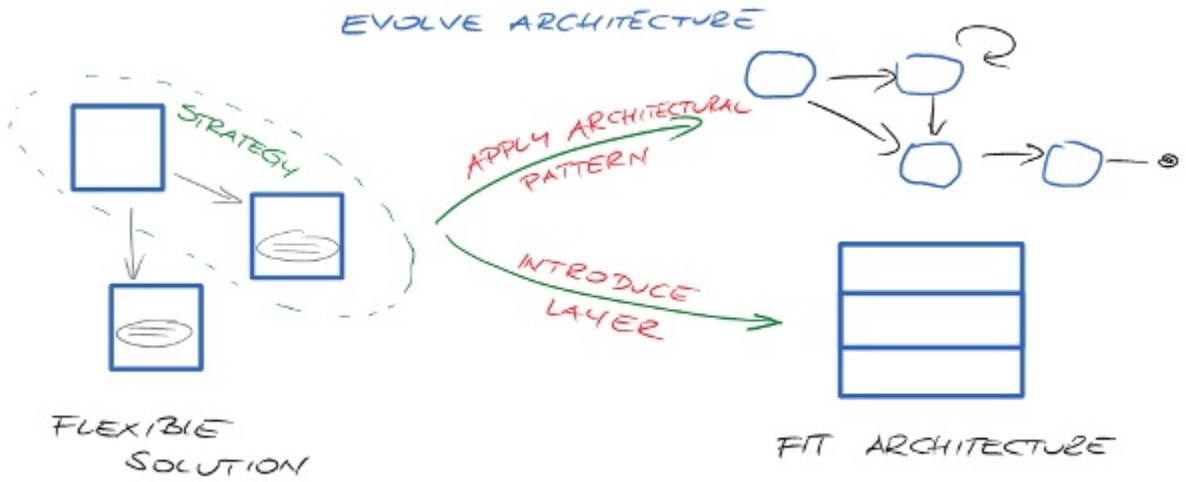
The most important question here is: "Do I really need this flexibility?" and you should have very strong reasons. Mental tools you use in this step: S.O.L.I.D., Design patterns, refactoring to patterns

Step 4. Evolve architecture

After some time some higher level patterns emerge:

- You may notice that state machine is better suited for your solution (or its part)
- You may notice that introducing event style of communication will give you expected system flexibility
- You may want to extract a read-only subdomain according to CQRS so that expensive queries would be much faster
- You may introduce or remove layer in you layered architecture
- You may want to apply some Domain-Driven Design concepts

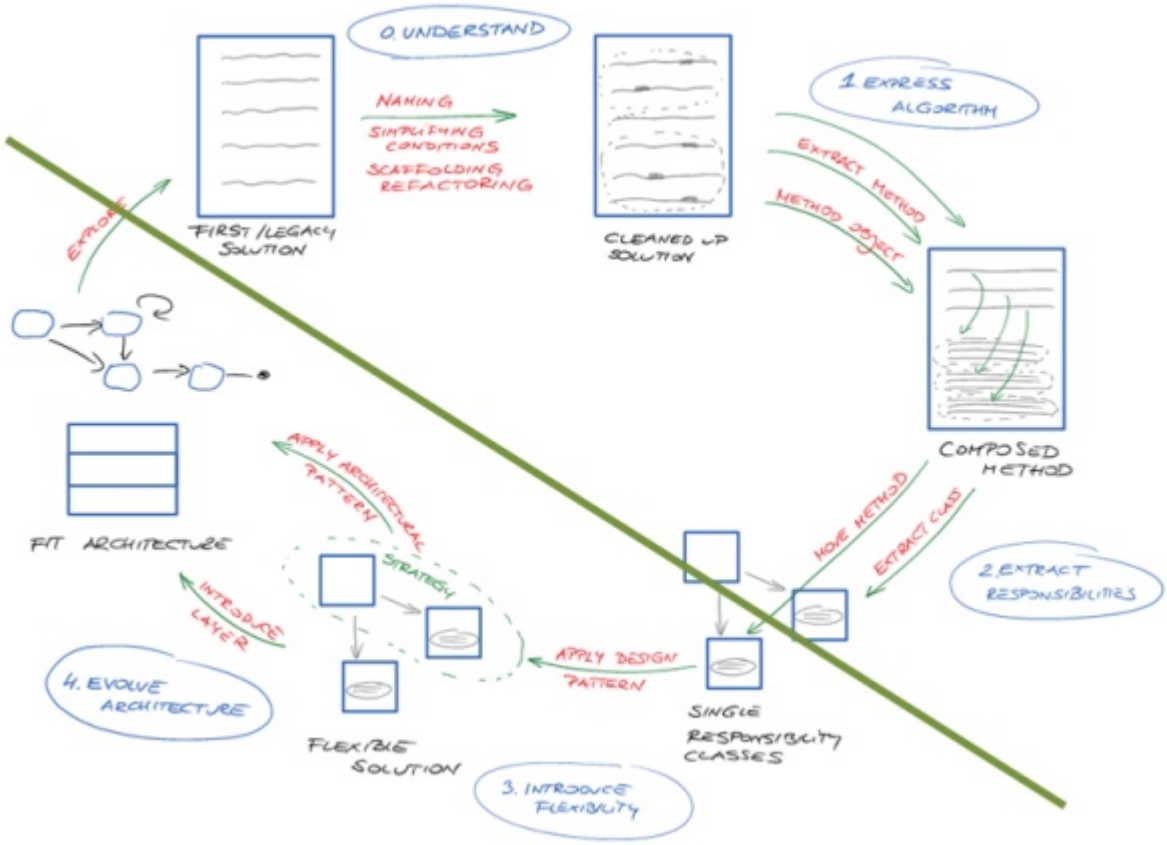
And so on…



Implementing these decisions require a lot of effort and usually cannot be done adhoc. They require longer term planning and are very risky. Thus this step doesn't occur too often but is needed in order to keep system architecture healthy.

This way you should have your architecture balanced just in time. It may require changes in future but now it is reasonably ok. Of course it is a little simplified view of things because evolving architecture is usually takes a long time and balancing architecture is more like a process than a particular state.

The whole process can be depicted this way with the border between everyday and strategic refactoring:



Practical hints

Natural Course of Refactoring is just the model. It is simplified view on how to organize refactoring process and what is the least possible step. It tells us how to do it in small steps and so you won't rush.

Here are some practical hints for using it in real life:

- It doesn't also mean that you should always follow the NCR steps in this order. Sometimes you can do two steps at once, you can omit a particular step.
- Most of your everyday refactoring fall in steps 0-2. It will significantly change readability of your codebase.
- The further the step is the less frequently it is performed. You do steps 3 and 4 less frequently (and this is so called strategic refactoring).
- The border between everyday and strategic refactoring is not strict and it is more a matter of effort needed to do refactoring.
- Legacy code is usually inconsistent so you can find different parts in different NCR phases.