http://www.faadooengineers.com/online-study/post/cse/software-engineering/231/refactoring

Refactoring

Introduction:-Coding often involves making changes to some existing code. Code also changes when requirements change or when new functionality is added. Due to the changes being done to modules, even if we started with a good design, with time we often end up with code whose design is not as good as it could be. And once the design embodied in the code becomes complex, then enhancing the code to accommodate required changes becomes more complex, time consuming, and error prone. In other words, the productivity and quality starts decreasing.

Refactoring is the technique to improve existing code and prevent this design decay with time. Refactoring is part of coding in that it is performed during the coding activity, but is not regular coding. Refactoring has been practiced in the past by programmers, but recently it has taken a more concrete shape, and has been proposed as a key step in the Extreme Programming practice. Refactoring also plays an important role in test driven development code improvement step in the TDD process is really doing refactoring.

Basic Concepts:-Refactoring is defined as a change made to the internal structure of softwareto make it easier to understand and cheaper to modify without changingits observable behavior . A key point here is that the change is beingmade to the design embodied in the source code exclusively for improvement purposes.

The basic objective of refactoring is to improve the design. However, note that this is not about improving a design during the design stages for creating a design which is to be later implemented (which is the focus of design methodologies), but about improving the design of code that already exists. In other words, refactoring, though done on source code, has the objective of improving the design that the code implements. Therefore, the basic principles of design guide

the refactoring process. Consequently, a refactoring generally results in one or more of the following:

1. Reduced coupling

2. Increased cohesion

3. Better adherence to open-closed principle (for OO systems)

Refactoring involves changing the code to improve one of the design properties, while keeping the external behavior the same. Refactoring is often triggered by some coding changes that have to be done. If some enhancements are to be made to the existing code, and it is felt that if the code structure was different (better) then the change could have been done easier, that is the time to do refactoring to improve the code structure.

Even though refactoring is triggered by the need to change the software (and its external behavior), it should not be confused or mixed with the changes for enhancements. It is best to keep these two types of changes separate. So, while developing code, if refactoring is needed, the programmer should cease to write new functionality, and first do the refactoring, and then add new code.

The main risk of refactoring is that existing working code may "break" due to the changes being made. This is the main reason why most often refactoring is not done. (The other reason is that it may be viewed as an additional and unnecessary cost.) To mitigate this risk, the two golden rules are:

1. Refactor in small steps

2. Have test scripts available to test existing functionality

If a good test suite is available, then whether refactoring preserves existing functionality can be checked easily. Refactoring cannot be done effectively without an automated test suite as without such a suite determining if the external behavior has changed or not will become a costly affair. By doing refactoring in a series of small steps, and testing after each step, mistakes in refactoring can be easily identified and rectified. With this, each refactoring

makes only a small change, but a series of refactorings can significantly transform the program structure. With refactoring, code becomes continuously improving. That is, the design, rather than decaying with time, evolves and improves with time.

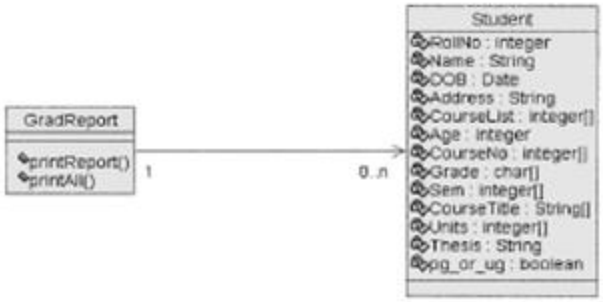

Figure 9.4: Initial class diagram.

With refactoring, the quality of the design improves, making it easier to make changes to the code as well as find bugs. The extra cost of refactoring is paid for by the savings achieved later in reduced testing and debugging costs, higher quality, and reduced effort in making changes.

If refactoring is to be practiced, its usage can also ease the design task in the design stages. Often the designers spend considerable effort in trying to make the design as good as possible, try to think of future changes and try to make the design flexible enough to accommodate all types of future changes they can envisage. This makes the design activity very complex, and often results in complex designs. With refactoring, the designer does not have to be terribly worried about making the best or most flexible design the goal is to try to come up with a good and simple design. And later if new changes are required that were not thought of before, or if shortcomings are found in the design, the design is changed through refactoring. More often than not, the extra flexibility envisaged and designed is never needed, resulting in a system that is unduly complex.

Refactoring is not a technique for bug fixing or for improving code that is in very bad shape. It is done to code that is mostly working the basic purpose is to make

the code live longer by making its structure healthier. It starts with healthy code and instead of letting it become weak; it continues to keep it healthy.

https://www.cs.usfca.edu/~parrt/course/601/lectures/refactoring/refactoring.html

Code Refactoring

Until you've had to live with the same piece of code for a while, you will not experience the need for refactoring. This lecture tries to summarize what refactoring is, when you need to do it, what patterns and tools are available. Further, I provide some jGuru examples to illustrate some of the concepts.

These lecture notes paraphrase or quote most content from **"Martin Fowler's Refactoring Book"**. I use keyword "TJP" to indicate my own thoughts. There are large sections of just TJP stuff.

Refactoring?

What?

Fowler says that refactoring is the

"*... process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure.*"

Just cleaning up code.

Contrary to idealized development strategy:

1. analysis and design
2. code
3. test

At first, code is pretty good but as requirements change or new features are added, the code structure tends to atrophy. Refactoring is the process of fixing a bad or chaotic design.

Amounts to moving methods around, creating new methods, adding or deleting classes, ...

TJP: Sometimes it means completely redoing the entire code base (i.e., throwing stuff away). Avoid the *second system effect*!

Why?

Improve code structure and design

- more maintainable
- easier to understand
- easier to modify
- easier to add new features

Cumulative effect can radically improve design rather than normal slide into decay.

Flip-flop code development and refactoring. Only refactor when refactoring--do not add features during refactoring.

TJP: kind of like an immune system that constantly grooms the body looking for offensive and intrusive entities.

Bad code usually takes more code to do the same thing often because of duplication:

- Usual estimates: 8 to 10% in normal industrial code
- Our Research:

Case Study	*Language*	*LOC*	*Duplication %*
gcc	*C*	*460'000*	*8.7% (5.6%)*
Database Server	*Smalltalk*	*245'000*	*36.4% (23.3%)*
Payroll	*Cobol*	*40'000*	*59.3% (25.4%)*
Message Board	*Python*	*6500*	*29.4% (17.4%)*

http://www.iam.unibe.ch/~scg/Teaching/OOSR/

TJP: from Fred Brook's "mythical man-month" remember that conceptual integrity is one of his big points. Addition of new features can break current system conceptual integrity. Must refactor your concept sometimes as well as your code to make it fit properly. For example, I integrated page snoopers with search so that I could look for

keywords regardless of whether the source was local or on some other page or other some other site's search engine.

Improving design then often includes removing duplicate code. Don't want duplicates because of bloat and also you only want one place to change functionality not multiple.

Fowler says he will refactor existing code just to understand it better and sometimes it helps to find bugs.

In summary, refactoring helps you develop better code, faster!

Kent Beck:

"*Programs have two kinds of value: what they can do for you today and what they can do for you tomorrow.*"

Mostly we are focused on today, but you cannot code today so that you cannot code tomorrow. Actively making the future smoother is a great idea. When you find tomorrow that today you made a mistake, you use refactoring to fix the decision.

Fowler says database refactoring is super hard. See TJP's jGuru example below for entity specifications. :)

When not to refactor?

- Sometimes you should throw things out and start again.
- Sometimes you are too close to a deadline.

jGuru experience

TJP: Locally-inoptimal refactoring-detours generally lead to a more optimal global solution. From personal experience, I can say that refactoring is absolutely and totally required to develop a large piece of software in the face of an ever-changing list of features.

Example 1: I started with a /search/results.jsp page that worked well for the single task. Then, had to show only some stuff for nonpremium folks. Then had to show a "marketing" version of the page. Finally it was too damn much--couldn't read the code and literally couldn't make it work in all combinations. After factoring out basically functionality and making separate pages that used the common functionality, I was able to quickly break apart results.jsp into 3 pages that were obvious.

Example 2: We started out using a search engine by thunderstone.

1. An employee coded some scripts in Thunderstone's proprietary xml-based programming language to do the right thing when we went to a particular page. He parsed the results in Java and then displayed it on a jguru-style page. So far so good.
2. Then we wanted to search other people's sites like slashdot, ibm, sun, etc... The employee wanted to use the Thunderstone language to grab, parse, and return a simple list of results that can be easily reparsed by jguru and displayed in a jguru page. I vetoed this suggesting that he get everything into a set of Java classes that did the same thing but that would integrate better into our Java code. Even though it was more work, I suspected we'd reap a benefit later. Further, I don't like be tied to a particular tool. After the effort, we had some "search resource" type objects that could do searches (either remotely or locally via Thunderstone) and display them on jGuru. We didn't treat our local search differently than a Slashdot search. Cool! Sounds good.
3. At this point, we also had some random code lying around that snooped other people's websites for articles. Bad. Getting a list of articles is same as getting list of search results (just different remote page). Oh well. Duplicated code etc... Oh well. We left it.
4. Sure enough, we decided to get rid of thunderstone. My advice was worth it because no "user level" code changed to incorporate Lucene (new engine). Yeah! Unfortunately, there was a huge amount of code duplication among the snooper and search spiders. I combined these into a nice hierarchy and common functionality. Now, just about anything can pretend to be a source of search results or snooped information.
5. So then we decided to not do remote searches because everybody else's search engines are so slow and we had lots of them to search. So, I added code to occasionally walk foreign (remote) sites to get their content (and only the good stuff) and add it to a local Lucene search database. We get better and faster results because of this, but...
6. Current smelly code: snooper does not add things to search db...I have separate code that fills the foreign search db. This is because you need the article list and search db to operate independently. Still, it would be better if the same code did both the snooping and the search db updates.

Example 3: An employee built a prototype forum manager. It had evolved over months and had naturally decayed in code beauty. I rewrote (since I was only coder employee then) to be better designed and fast etc... Boy was the new thing beautiful. Well, as we added more features and so on, it got uglier and uglier. Still I left it. Then we realized it was taking forever at start up because it loaded all forum messages up front. Ok, so I finally redesigned it to load last 3 months worth and do others

dynamically. Actually I knew that I would eventually have this "popularity" problem and it was hanging over me all the time. Now I don't worry about it. :)

When?

When you can't stand the code anymore or it becomes impossible to add new features or fix bugs.

When your boss isn't looking. ;) There is a lot of pressure not to do work that adds no functionality--shortsighted.

Rule of 3 from Don Roberts:

"*The first time you do something, you just do it. The second time you do something similar, you wince at the duplication, but you do the duplicate thing anyway. The third time you do something similar, you refactor.*"

Refactor when

1. you add new features and the code is brittle or hard to understand. Refactoring makes this feature and future features easier to build.
2. you fix bugs.
3. during code review.

What's that smell?

Refactor (*groom* as TJP calls it) when your code smells. It smells in the following situations.

[From Fowler's book, but summarized at
JHU, **http://www.cs.jhu.edu/~scott/oos/lectures/refactoring.html**]

- *Duplicated Code* extract out the common bits into theirown method (**extract method**) if code is in same class if two classes duplicate code, consider **extract class** to create a new class to hold the shared functionality.
- *Long Methods* **extract method**!
- *Large Class* Class trying to do too much often shows up as too many instance variables.
- *Long Parameter List* **replace parameter with method** (receiver explicitly asks sender for data via sender getter method) Example: day month, year, hour minute second ==> date

- *Divergent Change* If you have a fixed class that does distinctly different things consider separating out the varying code into varying classes **(extract class)** that either subclassor are contained by the non-varying class.
- *Shotgun Surgery* The smell: a change in one class repeatedly requires little changes in a bunch of other classes. try to **move method** and **move field** to get all the bits into one class since they areobviously highly dependent.
- *Feature Envy* Method in one class uses lots of pieces from another class. **move method** to move it to the other class.
- *Data Clumps* Data that's always hanging with each other (e.g. name street zip). Extract out a class (**extract class**) for the data. Will help trim argument lists too since name street zip now passed as one address object.
- *Switch (case) statements* Use inheritance and polymorphism instead (exampleof this was in Fowler Chapter 1; this is one of the more difficult refactorings)
- *Lazy Class* Class doesn't seem to be doing anything. Get rid of it!
    - **collapse heirarchy** if subclasses are nearly vacuous.
        - **inline class** (stick the class' methods and fields in the class that was using it and get rid of original class).
- *Speculative generality* Class designed to do something in the future but never ends up doing it. Thinking too far ahead or you though you needed this generality but you didn't. like above, **collapse hierarchy**or **inline class**
- *Message chains* Say you want to send a message toobject D in class A but you have to go through B to get C and C to get D. use **hide delagate** to hide C and D in B, and add a method to B that does what A wanted to do with D.
- *Inappropriate Intimacy* Directly getting in and munging with the internals of another class. To fix this, move methods, inline methods, to consolidate the intimate bits.
- *Incomplete Library Class* If method missing from library, and we can't change the library, so either: o make this method in yourobject (**introduce foreign method**) If there is a lot of stuff you want to change: o make yourown extension/subclass (**introduce local extension**)
- *Data Class* We have already talked about this extensively: in data-centric design, there are some data classes which are pretty much structs: no interesting methods. first don't letother directly get and set fields (make them private) and don't have setter for things outsiders shouldn't change look who uses the data and how they use it and move someof that code to the data class via a combinationof **extract method** and **move method** (see the Fowler chapter 1 example for several examplesof this)
- *Comments* Comments in the middleof methods are deodorant. You should really refactor so each comment block is its own method. Do **extract method**.

Sample refactoring "patterns"

Here is a catalog of refactoring patterns:

http://www.refactoring.com/catalog/changeUnidirectionalAssociationToBidirectional.html

TJP Refactoring Example

PROBLEM: have BitSet but need an integer set representation that is efficient for unicode chars, makes BitSets huge! Started thinking of a List of ranges as a good representation.

1. First thing I did was to "extract interface IntSet", making BitSet implement it. Intellij automatically converted references (even in comments) to BitSet into IntSet references. In this way I can swap in a different representation w/o altering code.
2. Started IntRangeSet implementing IntSet. Decided that using List/ArrayList etc... would be too slow so I decided to use my own IntArrayList, which is like Vector but implements List. Range i is at indices (2i,2i+1). Efficient: fewer object allocations. Decided not to sort / compact ranges until I needed to in order to avoid shuffling data around in the array like insert range, delete range etc... Code wasn't exactly clear, but it was fast accessing individual ranges etc.. I had decided against a Range object to avoid extra allocations, load on the GC.
3. Started testing and realized that I could really use some upgrades to the testing rig. Added auto runTests() construction: call all methods with "test" prefix. Then added method assertEqual() that will print what was expected vs result when mismatch. Etc...
4. I got IntRangeSet mostly working and tested when I had to implement equals(). Decided that it was horribly expensive and I needed it in my DFA construction. Damn. Ok, better keep ranges sorted and disjoint to make operations efficient.
5. At this point it occurred to me that IntervalSet was kind of a better name than IntRangeSet as I'm really a set of Intervals that can double as a set of integers. Interval arithmetic is a more term anyway. The change was easy globally with Intellij.
6. Anyway, to keep intervals disjoint/sorted, I needed to be able to insert and delete and merge elements in a list. That implied a LinkedList, which implied I should make an Interval object and just use the standard classes and make the code more obvious. I realize that I had falsely (probably) optimized (and too early). Back to readable code as a primary goal and worry about speed later. Now I had IntervalSet with an Interval object; it started out as a class scoped in IntervalSet but as it got bigger I refactored ("move") it into a separate file.
7. As I implement the new mechanism, I notice that the overall structure of the class is sound and all the code that references it does not get affected. Further, I

note that my unit tests are extremely useful when making the changes--I can see the implementation make more and more tests work. I know I'm making progress and I have some idea that the code is working. I code with confidence because I know anything that is "wrong" will instantly show up as a bunch of failed tests.

8. I realize that these classes are in org.antlr.analysis when they should be in the misc package. I move them all trivially thanks to Intellij.
9. Finally I get back to equals() and it's now a trivial array comparison done automatically by List :) Finally got more stuff tested and all is cool.
10. At some point during the implementation of IntervalSet, when it was still IntRangeSet, I realized that due to the fundamental element actually being an Interval, it was hard to make IntRangeSet/IntervalSet conform to the same IntSet interface as BitSet. I abandon the relationship, but for now leave the IntSet/BitSet relationship as I may in fact make a new implementation of that later to make even simple int sets more efficient in a new situation. This is a case where the abstract ideas fit nicely together but in practice the type system fails you, making it hard to fit two things together (like ANTLR's AST interface that has the same name but different return types than the DOM XML interface; incompatible, but the same operations and names: getNextSibling() etc...; did they copy my names?). It also turns out that implementing the IntSet methods like member() are very expensive for IntervalSet and perhaps a pain to write. Now that I'm writing this (May 29, 2004), it seems that perhaps I could make it implement IntSet and then just throw "unimplemented" or whatever for the stuff that I haven't implemented at the moment because it's unneeded for my application. Here is a great case where writing stuff down in your "lab notebook" or "telling it to the teddy bear" really helps!
11. As I start using IntervalSet I realize that I have misnamed difference; it is really subtract. Difference is more like |x-y| for two sets x and y (the "xor" operation). I rename trivially with Intellij and it does the global replace.

Simplifying System Architecture As Refactoring (TJP)

I suppose you could consider simplifications in system architecture refactoring as they improve the system/software design without changing external behavior. This is, again, a restatement of Brook's principle of conceptual integrity.

Patterns TJP used on jGuru

Fowler:

"*Any fool can write code that a computer can understand. Good programmers write code that humans can understand.*"

I did lots of refactoring at jGuru to continuously "groom" the code as I called it. In terms of Fowler, I did some of the following in order of most common to least.

- **"Extract Method"**
- **"Pull up method"**
- **"Add parameter"**
- **"Rename method"**
- **"Replace temp with query"**
- **"Introduce a controller"** TJP calls this a *service* like `LogManager`

Renaming is a great idea because code should communicate clearly and variable names are super important (imagine a program where everything is `v1`, `v2`, ...; that is what obfuscaters do).

The features of Intellij's IDEA refactoring tool I used:

- extract method (does all the flow analysis)
- rename
- change method signature
- extract interface
- implement methods from interface/superclass
- generate getter/setters for instance variables

I also introduced lots of classes to factor out common behavior, but I did this manually.

My first impression was that all of this was not very useful (coming from emacs), but I have come to really rely on the power of these operations. For example, change method signature changes every reference to that method even for messages to all subclasses of the method's class you change etc...

Thoughts on IDEs vs emacs (TJP)

Coding/refactoring is part

1. thinking
2. text editing using well-established idioms and patterns
3. questioning (searching, jumping to a class def, finding implementations of methods etc...)

4. cutting-n-pasting

Emacs really only does a good job of 2, which is not a huge percentage of coding. It is much better to treat your program like a database of code not just plain text. Being able to quickly see the class hierarchy or find implementors or references of a method is very useful.

After switching to Intellij's IDEA dev environment (still the best IDE on any platform in my opinion) I became MUCH more productive. Without code completion of method names and variables, i'd have worse tendonitus i can tell you.