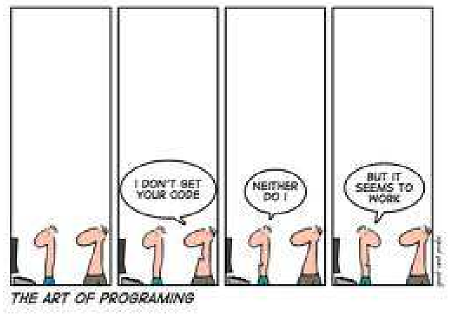


Refactoring ➔ Introduction

www.cs.uoi.gr/~zarras/soft-devII.htm

from Refactoring by Martin Fowler

Definition

Refactoring (noun):

a change made to the internal structure of software to make it *easier to understand and cheaper to modify without changing its observable behavior*

Refactor (verb):

to restructure software by applying a series of refactorings without *changing its observable behavior.*

Definition

Is refactoring just cleaning up code?

In a way the answer is yes, but refactoring goes further because it provides techniques for cleaning up code in a more efficient and controlled manner.

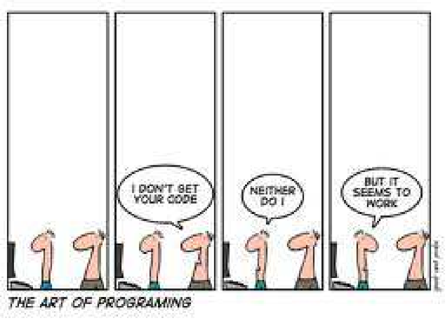
Why do it ?

Refactoring Improves the design of software

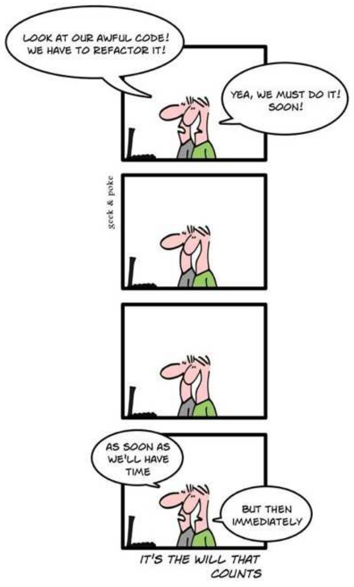
Refactoring makes software easier to understand

Refactoring helps find bugs

Refactoring makes you program faster



When to refactor ?


LOOK AT OUR AWFUL CODE! WE HAVE TO REFACTOR IT!
YEA, WE MUST DO IT! SOON!
geek & poke
AS SOON AS WE'LL HAVE TIME
BUT THEN IMMEDIATELY
IT'S THE WILL THAT COUNTS

Refactoring is not an activity you set aside time to do.

Refactoring is something you do all the time in little bursts.

You don't decide to refactor, you refactor because you want to do something else, and refactoring helps you do that other thing.

When not to refactor ?

LOOK AT OUR AWFUL CODE! WE HAVE TO REFACTOR IT!
YEA, WE MUST DO IT! SOON!
geek & poke
AS SOON AS WE'LL HAVE TIME
BUT THEN IMMEDIATELY
IT'S THE WILL THAT COUNTS

There are times when the **existing code is such a mess** that although you could refactor it, it would be easier to start from the beginning.

A clear sign of the need to rewrite is when the current code just **does not work**. You may discover this only by trying to test it and discovering that the code is so full of bugs that you cannot stabilize it.

Remember, code has to **work mostly correctly** before you refactor.

When not to refactor ?

LOOK AT OUR AWFUL CODE! WE HAVE TO REFACTOR IT!
YEA, WE MUST DO IT! SOON!
geek & poke
AS SOON AS WE'LL HAVE TIME
BUT THEN IMMEDIATELY
IT'S THE WILL THAT COUNTS

A compromise route is to **refactor** a large piece of software into **components** with strong encapsulation. Then you can make a **refactor-versus-rebuild decision** for one component at a time.

The other time you should avoid refactoring is when you are **really close to a deadline**.

At that point the productivity gain from refactoring would appear after the deadline and thus be too late.

Refactor in small steps

"... you start digging in the code …
soon you discover new opportunities and you dig deeper …
eventually you dig yourself into a hole …"
(M. Fowler)

Refactor in small steps

Composing methods

Moving features

Simplify conditionals

Simplify method calls

Generalizations

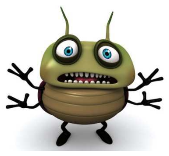
Fowler’s Testing Advices

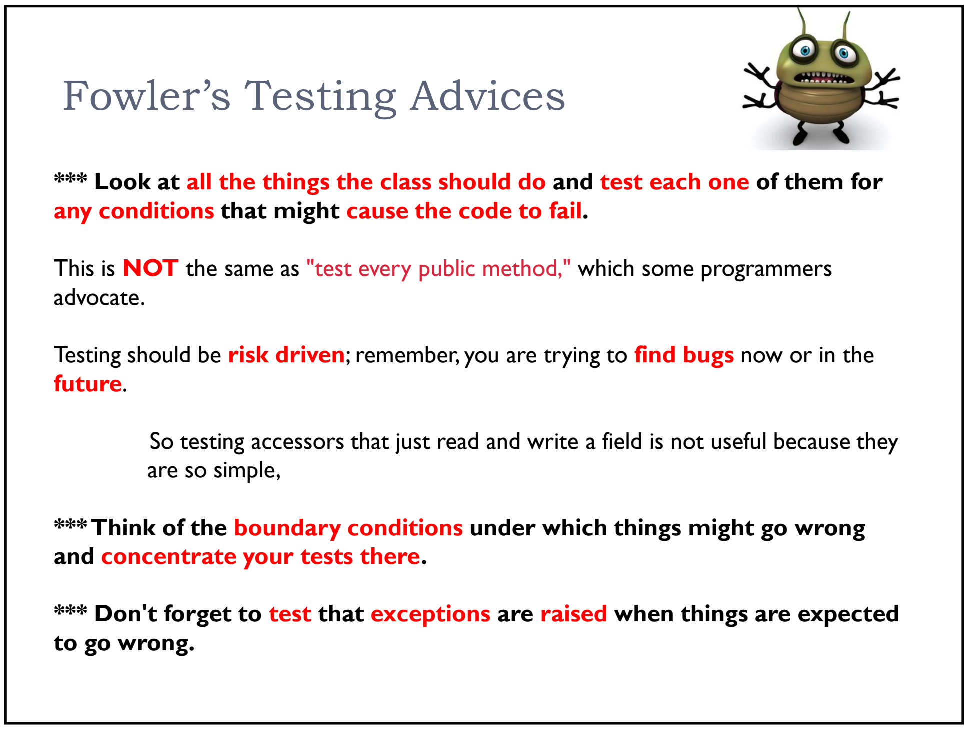
***** If you want to refactor, the essential precondition is having solid tests.**

Even if you are fortunate enough **to have a tool that can automate the refactorings, you still need tests.**

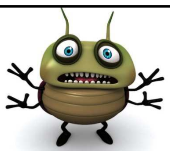
It will be a long time before all possible refactorings can be automated in a refactoring tool.

***** Make sure all tests are fully automatic and that they check their own results.**





Fowler’s Testing Advices



***** Look at all the things the class should do and test each one of them for any conditions that might cause the code to fail.**

This is **NOT** the same as "test every public method," which some programmers advocate.

Testing should be **risk driven**; remember, you are trying to **find bugs** now or in the **future**.

So testing accessors that just read and write a field is not useful because they are so simple,

*****Think of the boundary conditions under which things might go wrong and concentrate your tests there.**

***** Don't forget to test that exceptions are raised when things are expected to go wrong.**