Refactoring

Refactoring is...

...the technique of improving the design of existing code, without attempting to add new functionality at the same time;

...extremely fashionable right now (but very valuable anyway :-);

...especially valuable in large, hard-to-understand systems but still useful in one-person systems.

A refactoring is a transformation you make on the code that improves its design without attempting to add functionality.

Refactoring can be supported just by a reasonably powerful editor (e.g. emacs) or by a special purpose tool (e.g. the Refactoring Browser: Smalltalk only at present).

Slide 3

Why is this useful?

Software decays – or more long-windedly, it suffers architectural degradation. There are two main ways in which this happens:

1. Even if no-one touches the code, things around it change. E.g. operating system, libraries, other systems with which it interacts, business processes, the century... Eventually the system doesn't work any more.
2. When people do change code, the code tends to be less well-structured after the change than before. This may be, for example:
   - because the person doing the chance doesn't understand the design, or
   - because timeliness pressures persuade them to make the change in the quickest way for now, regardless of how this "hack" messes up the integrity of the system, or
   - because the design that was suitable for the original demands on the system is not ideal to support the change.

Most systems are developed over at most a few years, but they may have lifetimes of many years or decades in maintenance. One definition of a *legacy system* is "a system which is valuable to the organisation which owns it, but which has become hard to maintain or evolve to meet changing business requirements". Most organisations of any size have significant legacy systems, and maintaining them can absorb a high proportion of the organisation's IT budget. It would be nice to think that we have legacy systems today because decades ago people didn't know how to design maintainable systems, and to conclude that there is no new supply of legacy systems being built. Unfortunately, this does not seem to be true. Techniques such as structured programming and (later) object oriented design have helped to some exent, but there are many legacy systems only a few years old; even the best design can be obscured by a few years' changes.

So what can be done about this? The obvious answer is "whenever you make a change, think carefully about the effect on the design, change the design as necessary and leave the system after your change as well designed as it was to start with".

This would work fine, if everyone did it all the time. However, it's extraordinarily hard to do (correctly, or indeed at all). If you're focusing on adding new functionality or fixing a bug, thinking about altering the design tends to be a distraction: you end up with one or other task half done. Moreover if between your well-understood starting point and the next time you test you've done some changes *and* some redesign, it's hard to tell whether you've broken anything and even harder to track down any bugs you may have introduced.

Why refactor?

To counteract architectural degradation.

To avoid producing new legacy systems.

To make systems more readable.

To increase your chances of making changes correctly:

– to decrease the defect density of the product;

– so that you spend less time with a debugger

– so you and your colleagues are more productive

– to decrease your stress levels.

Being confident with refactoring arguably helps you to keep designs as simple as possible: if you need something more complex later on you have a manageable way to add it when you need it, so there's no need to guess all the complications you need up front.

Slide 4

The laughably simple basic idea behind refactoring is that it works better to separate out the redesign from the enhancement. If when you think about a change that has to be made you realise that the design as it stands doesn't support the change ("I wouldn't want to start from here"):

Steps for carrying out a refactoring

1. Consider where you wish you were starting from. What design *could* the system have which would support your change better (whilst, of course, still satisfying all the other requirements).
2. Without attempting to change the existing functionality, *refactor* the system into this new design.
3. Rerun all the tests: because you haven't attempted to alter functionality, the system should pass all the tests it passed before (possibly with routine modifications, but ideally without). If not, fix it!
4. Now you have a system with a design which makes your change straightforward. Make the change; add any new tests to check that the change has been correctly added; rerun all the tests; fix if necessary.

Slide 5

Simplest possible refactoring

Rename a class or a method (to something more helpful to the reader). Consider this any time you find yourself confused or mistaken about the essential purpose of a class or method.

Steps:

1. Use query replace in an editor to change the names in code.
2. *Very carefully* change any names that need to be changed in test code.
3. Recompile and rerun tests.

What could possibly go wrong?

If there are no name clashes, very little.

If you're changing to or (more likely) from a name which is also used for something else – so that a global replace wouldn't do – it's easy to make a few wrong changes among hundreds of occurrences.

The compiler and/or your tests should catch this (if not, you need more tests).

Slide 6

More interestingly, refactorings include things like:

- creating a new class to capture the commonality between two or more classes;

- removing a class that isn't pulling its weight;
- moving a method from one class to another;
- changing a case statement to a send of a polymorphic message.

Extract Superclass (from Fowler, p336)

You have two classes with similar features.

Create a superclass and move the common features to the superclass.

1. Create blank abstract superclass.
2. One by one, pull up the shared fields and methods to the superclass. Compile and test after each one.
3. See whether the non-shared methods have shared parts that should also be extracted, if so extract them.
4. Finally check the clients of these classes. If they only use the common interface, consider making them refer to the new superclass.

Slide 7

The arguments against refactoring

1. All that stuff about rerunning tests... what tests?
2. I don't understand how and when to do it; I'm afraid I might go round in circles.
3. The benefits are long-term and I'm not interested in the long term. I'm not going to be maintaining this code. (Either: (a) no-one is; (b) someone else is, but it won't be my problem.)
4. Refactoring code is an overhead activity. I'm paid to write new features.
5. Anything I do to this code base, including refactoring, might break it.

Slide 8

Responses in very brief:

1. You need tests whether or not you're refactoring! The emphasis placed on them is partly because refactoring began in the Smalltalk world where the compiler is no help. But even in e.g. Java the compiler can't catch all errors. There are automatic tools (e.g. JUnit) to help.
2. Refactor when you're confident the result would be an improvement. This is partly, inevitably, about understanding what good design is. A book like Fowler's Refactoring may help. It lists warning signs (Bad Smells in code), e.g. long methods, classes with no behaviour, etc. and suggests refactorings to help. If you can't understand it, it's probably badly designed (or badly coded, or probably both).
3. Refactoring appropriately makes it quicker to add new features, because you're more likely to do so correctly.
4. Good design matters more in a long-lived system than a throw-away prototype, it's true. But remember many systems live longer than expected. And if you still have to do more work on the system before it is thrown away, it's possible some refactoring will make your life easier.

Relationship between refactorings and patterns?

1. Many of the same people are interested in both so they have a family ressemblance.
2. Refactorings need not be described in a standard form, but they can are (e.g. in Martin Fowler's book).
3. One of the ways in which you might improve the design of code is by introducing an appropriate design pattern.

Slide 9

Further comment on the slide points:

1. That is, the people involved have the same kind of idea about what constitutes good design. Within the OO community this is not too controversial though.
2. How valuable is such a catalogue? You might like to look at the book. My feeling is that most of the refactorings described are obvious: listing them like this might be useful if you're developing a tool to help with refactoring, but I think that, unlike patterns, what's really valuable about refactoring is the idea, rather than any particular refactoring. YMMV.
3. Many design patterns are intended to make the system support certain kinds of changes easily. So if you're faced with such a change and a system that does not support the change easily, you can refactor to introduce the design pattern, then add the change.

References

Martin Fowler *Refactoring* Addison-Wesley Object Technology series.