Refactoring

prepared for COMP314,
Bernhard Pfahringer
see Code Complete, Chapter 24

Refactoring definition

- "change to the internal structure of software to make it easier to understand and cheaper to change, WITHOUT changing the observable behaviour" (Martin Fowler, 1999)
- I.e. changes "inside the black box" only

Key points

- Program changes are a fact of life
- Change => degrade or improve
- “code smells” indicate need for change
- know different refactorings (esp. IDE support …)
- use safe strategy
- do it early on

“Code smells”

- code is duplicated
- a method is too long
- a loop is too long or too deeply nested
- class has poor cohesion
- class interface does not provide consistent level of abstraction
- parameter list has too many parameters
- changes within a class are compartmentalized
- change requires parallel mods in mult.classes
- inheritance hierarchies must be modified in parallel

more code smells

- case statements modified in parallel
- related data not organized into classes
- method uses more features of another class than its own
- primitive data type is “overloaded”
- class does not too very much
- chain of methods passes “tramp data”
- middleman object doing nothing
- class relies on internals of another
- method has a poor name
- public data members/fields

and even more code smells

- subclass uses only small parts of its parent's methods
- comments explain too complex code
- use of global variables
- method must use complex setup code before and/or takedown code after calling another method
- code not needed now, but maybe in the future

Data level refactorings

- replace magic number with named constant
- rename variable to clearer informative name
- move expression inline
- replace expression with method call
- introduce intermediate variable
- replace multi-use variable with single-use variables

More Data level refactorings

- Use local variable instead of parameter (use final in parameter list)
- convert primitive data to class
- convert type codes to enumeration
- or to class with sub-classes
- change array to an object
- encapsulate collection

Statement-level refactoring

- decompose boolean expression
- replace complex boolean exp. with well-named method
- consolidate code fragments of different parts of conditional statement
- Use break/continue/return in loops
- Return/break early
- use polymorphism instead of switch
- use “null” objects

Method-level refactoring

- extract a method
- move method call inline
- turn a long routine into a class
- replace complex algorithm with a simple one
- add a parameter
- remove a parameter

more Method-level refactoring

- separate query from modification
- combine similar methods by parameterization
- separate methods
- pass one whole object instead of many specific fields (cf “Addressee”)
- pass specific fields instead of object
- encapsulate downcasting

Class implementation refactoring

- change value object to reference object
- change reference object to value object
- replace method with data initialization
- change member data or method placement
- extract specialised code into subclass
- combine similar code into superclass

Class interface refactoring

- move method into another class
- split class into two
- remove a class
- hide a delegate
- remove a middleman
- replace inheritance by composition
- replace composition by inheritance
- introduce “foreign” method
- introduce extension class

more Class interface refactoring

- encapsulate exposed data fields
- remove unwanted set methods
- hide methods that are intended for use outside a class
- encapsulate unused methods
- collapse sub with superclass if very similar

System-level refactorings

- create reference source for data outside your control
- change bi-directional class association to uni-directional
- change uni-directional class association to bi-directional
- Provide factory method instead of a constructor
- replace error code with exceptions and v.v.

Safe refactoring (similar for code performance tuning)

- ensure rollback (e.g. via svn)
- small steps, one at a time
- always test (unit tests, and more)

- bug fixing is NOT refactoring
- refactor early