Refactoring

Definition
Refactoring consists of improving the internal structure of an existing program's source code, while preserving its external behavior.

Common Pitfalls
Refactoring does "not" mean:
- Rewriting code
- Fixing bugs
- Improve observable aspects of software such as its interface

Refactoring in the absence of safeguards against introducing defects (i.e. violating the "behaviour preserving" condition) is risky. Safeguards include aids to regression testing including automated unit tests or automated acceptance tests, and aids to formal reasoning such as type systems.

Expected Benefits
The following are claimed benefits of refactoring:
- Refactoring improves objective attributes of code (length, duplication, coupling and cohesion, cyclomatic complexity) that correlate with ease of maintenance
- Refactoring helps code understanding
- Refactoring encourages each developer to think about and understand design decisions, in particular in the context of collective ownership / collective code ownership
- Refactoring favors the emergence of reusable design elements (such as design patterns) and code modules

Signs Of Use

**Version Control** records (such as CVS or git logs) include entries labeled "Refactoring" the code modifications corresponding to such entries can be verified to be behaviour-neutral: no new unit tests or functional tests are introduced, for example

Skill Levels

**Beginner**
- Knows the definition of "refactoring"
- Can use some automated refactorings from the ide
- Can perform some refactorings by hand
- Is aware of the risks of regression from manual and automated refactorings
- Is aware of code duplication and can remove it by refactoring

**Intermediate**
- Knows and is able to remedy a broader range of "code smells"
- Can chain several refactorings to carry out a design intention, in awareness of the dependencies between refactorings
- Refactors continuously, rather than in sporadic and lengthy sessions
- Advanced
- Has an acute sense of code duplication and coupling
- Applies refactorings to non-code elements such as database schema, documents, etc.
- Uses refactoring to guide large bodies of code toward design styles intentionally chosen from a broad palette: object-oriented, functional, or inspired by known design patterns
Further Reading

Refactoring, by Martin Fowler

The Mikado Method

Origins

1984: the notion of "factoring", an anticipation of refactoring, is described in Brodie's "Thinking Forth", where it is presented as "organizing code into useful fragments" which "occurs during detailed design and implementation".

1990: Bill Opdyke coins the term "refactoring" in an ACM SIGPLAN paper with Ralph Johnson, "Refactoring: An aid in designing application frameworks and evolving object-oriented systems"

1992: a comprehensive description of "refactoring" is presented in Opdyke's thesis, "Refactoring object-oriented frameworks"

1999: the practice of "refactoring", incorporated a few years earlier into Extreme Programming, is popularized by Martin Fowler's book of the same name

2001: refactoring "crosses the Rubicon", an expression of Martin Fowler describing the wide availability of automated aids to refactoring in IDEs for the language Java
**Definition**
Refactoring consists of improving the internal structure of an existing program's source code, while preserving its external behavior.

The noun "refactoring" refers to one particular behavior-preserving transformation, such as "Extract Method" or "Introduce Parameter."

**Common Pitfalls**
Refactoring does "not" mean:
- Rewriting code
- Fixing bugs
- Improve observable aspects of software such as its interface

Refactoring in the absence of safeguards against introducing defects (i.e. violating the "behaviour preserving" condition) is risky. Safeguards include aids to regression testing including automated unit tests or automated acceptance tests, and aids to formal reasoning such as type systems.

**Expected Benefits**
The following are claimed benefits of refactoring:
- Refactoring improves objective attributes of code (length, duplication, coupling and cohesion, cyclomatic complexity) that correlate with ease of maintenance
- Refactoring helps code understanding
- Refactoring encourages each developer to think about and understand design decisions, in particular in the context of collective ownership / collective code ownership
- Refactoring favors the emergence of reusable design elements (such as design patterns) and code modules

**Signs Of Use**
Version Control records (such as CVS or git logs) include entries labeled "Refactoring".
The code modifications corresponding to such entries can be verified to be behaviour-neutral: no new unit tests or functional tests are introduced, for example

**Skill Levels**
**Beginner**
- knows the definition of "refactoring"
- can use some automated refactorings from the IDE
- can perform some refactorings by hand
- is aware of the risks of regression from manual and automated refactorings
- is aware of code duplication and can remove it by refactoring

**Intermediate**
- knows and is able to remedy a broader range of "code smells"
- can chain several refactorings to carry out a design intention, in awareness of the dependencies between refactorings
- refactors continuously, rather than in sporadic and lengthy sessions

**Advanced**
- has an acute sense of code duplication and coupling
- applies refactorings to non-code elements such as database schema, documents, etc.
- uses refactoring to guide large bodies of code toward design styles intentionally chosen from a broad palette: object-oriented, functional, or inspired by known design patterns

**Further Reading**
Refactoring, by Martin Fowler
The Mikado Method
Origins
1984: the notion of "factoring", an anticipation of refactoring, is described in Brodie's "Thinking Forth", where it is presented as "organizing code into useful fragments" which "occurs during detailed design and implementation".
1990: Bill Opdyke coins the term "refactoring" in an ACM SIGPLAN paper with Ralph Johnson, "Refactoring: An aid in designing application frameworks and evolving object-oriented systems"
1992: a comprehensive description of "refactoring" is presented in Opdyke's thesis, "Refactoring object-oriented frameworks"
1999: the practice of "refactoring", incorporated a few years earlier into Extreme Programming, is popularized by Martin Fowler's book of the same name
2001: refactoring "crosses the Rubicon", an expression of Martin Fowler describing the wide availability of automated aids to refactoring in IDEs for the language Java