

Refactoring

Paul Jackson

School of Informatics
University of Edinburgh

The Problem

As code evolves its quality naturally decays

- ▶ Initially code implementing a good design
- ▶ Changes often local, without full understanding of the context
- ▶ With loss of structure, code becomes harder to follow, harder to modify

Refactoring is about restoring good design in a disciplined way

- ▶ Expertise on refactoring captured in *refactoring patterns*
- ▶ Enables rapid learning
- ▶ Enables tool support

Refactoring definition

Refactoring (noun) is a change made to the internal structure of software to make it

- ▶ easier to understand, and
- ▶ cheaper to modify

without changing its observable behaviour

Refactor (verb) to restructure software by applying a series of refactorings *without changing its observable behaviour*

Fowler, *Refactoring*, 2000

Refactoring (noun) also used to refer to the general activity

Why refactor?

Refactoring

- ▶ makes software easier to understand
 - ▶ Your code, by you,
 - ▶ Your code, by others,
 - ▶ Others code, by you
- ▶ helps you make subsequent modifications quicker
- ▶ helps you find bugs
 - ▶ Design becomes clearer and bugs easier to see

The **result**: refactoring helps you program *faster*

When to refactor?

Refactoring was once seen as a kind of maintenance. . .

- ▶ You've inherited legacy code that's a mess.
- ▶ A new feature is required that necessitates a change in the architecture.

But can also be an integral part of the development process

Agile methodologies (e.g. XP) advocate continual refactoring (XP maxim: "Refactor mercilessly").

What does refactoring do?

A refactoring is a *small* transformation which preserves correctness.

There are many examples.

For a catalogue of over 90 assembled by Martin Fowler, see <http://refactoring.com/catalog/>.

A sample:

- ▶ Add Parameter
- ▶ Change Bidirectional Association to Unidirectional
- ▶ Extract Variable (Introduce Explaining Variable)
- ▶ Replace Conditional with Polymorphism

Extract Variable

Change

```
if ( (platform.toUpperCase().indexOf("MAC") > -1) &&
      (browser.toUpperCase().indexOf("IE") > -1) &&
      wasInitialized() && resize > 0 )
{
    // do something
}
```

to

```
final boolean isMacOs      = platform.toUpperCase().indexOf("MAC") > -1;
final boolean isIEBrowser = browser.toUpperCase().indexOf("IE")  > -1;
final boolean wasResized  = resize > 0;

if (isMacOs && isIEBrowser && wasInitialized() && wasResized)
{
    // do something
}
```

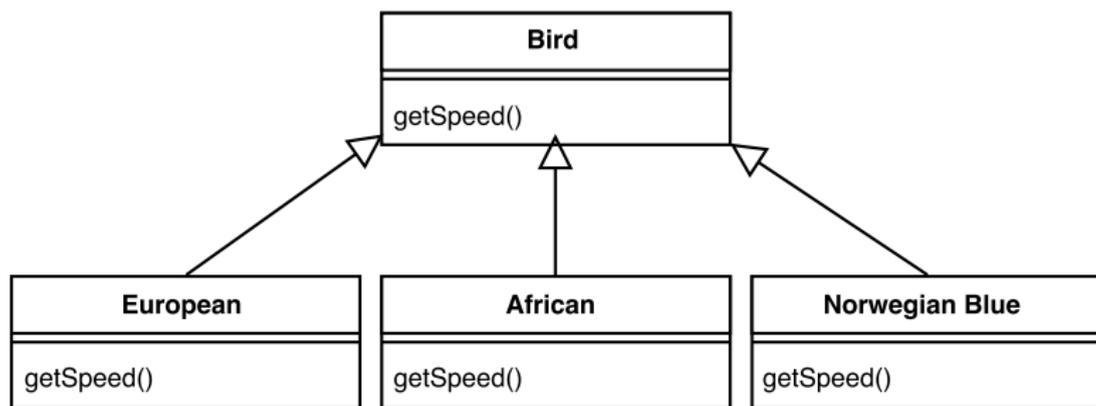
Replace Conditional with Polymorphism I

Change

```
double getSpeed() {
    switch (_type) {
        case EUROPEAN:
            return getBaseSpeed();
        case AFRICAN:
            return getBaseSpeed() - getLoadFactor() * _numberOfCoconuts;
        case NORWEGIAN_BLUE:
            return (_isNailed) ? 0 : getBaseSpeed(_voltage);
    }
    throw new RuntimeException ("Should be unreachable");
}
```

Replace Conditional with Polymorphism II

to



Eclipse Refactoring

Eclipse has a built-in refactoring tool (on the Refactor menu).

Many of its refactoring operation can be grouped in three broad classes . . .

Eclipse Refactoring I:

Renaming and physical reorganization

A variety of simple changes.

For example:

- ▶ Rename Java elements (classes, fields, methods, local variables)
 - ▶ On class rename, `import` directives updated
 - ▶ On field rename, getter and setter methods also renamed
- ▶ Move classes between packages

Eclipse applies these changes *semantically*

- ▶ Much better than syntactic search-and-replace

Eclipse Refactoring II: Modifying class relationships

Heavier weight changes. Less used, but seriously useful when they are used. E.g.

- ▶ Move methods or fields up and down a class inheritance hierarchy.
- ▶ Extract an interface from a class
- ▶ Turn an anonymous class into a nested class

Eclipse Refactoring III: Intra-class refactorings

The bread-and-butter of refactoring: rearranging code within a class to improve readability etc. E.g.

- ▶ Extract Method: pull code block into new method.
 - ▶ Good for shortening method or making block reusable
 - ▶ Also can extract local variables and constants
- ▶ Encapsulating fields in accessor methods.
- ▶ Change the type of a method parameter or return value

Safe refactoring

How do you know refactoring hasn't changed/broken something?

Perhaps somebody has *proved* that a refactoring operation is safe.

More realistically:

test, refactor, test

This works better the more tests you have: ideally, unit tests for every class.

Bad smells in code

- ▶ Duplicated code
- ▶ Long method
- ▶ Large class
- ▶ Long parameter list
- ▶ Lazy class
- ▶ Long message chains

Smell documentation explains how to recognise them and what refactorings can help.

Reading

- Required:** The article 'Refactoring for everyone' at <http://www.ibm.com/developerworks/opensource/library/os-ecref/>. Aim to remember: what refactoring is, and a few examples, not the details of the refactorings discussed here.
- Suggested:** Look at the *Reference - Refactor Actions* section of the *Eclipse Java development user guide* for full information on Eclipse's current capabilities.
- Suggested:** Browse around Fowler's page at <http://refactoring.com/>. Some of his book *Refactoring* is available on Google Books e.g., details of some of the refactorings in the catalogue.
- Suggested:** Search *code smells*. One catalogue can be found at <http://wiki.c2.com/?CodeSmell>.