Why Refactor?

Imagine you’ve written a piece of code but then accidentally deleted and lost it.

Questions:
• How much time would it take you to reconstruct from scratch what you had – the same amount, or more, or less?
• Would the code have a better design the second time you write it?

Imagine you are doing software maintenance.

Observations:
• It’s harder to maintain (someone else’s) code than it is to write new code.
• Most developers hope that they won’t have to deal with code maintenance.
• Evolving/maintaining code is what most developers do most of the time.

Good engineering: keep code simple and easy to understand.

Problem: code is hard to evolve and maintain
Solution: refactor to keep it simple

“Any fool can write code that a computer can understand. Good programmers write code that humans can understand.”
Martin Fowler
What is Refactoring?

“[Refactoring is] the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure” – Martin Fowler

Changes made to a system that:

• Do not change observable behavior
• Remove duplication or needless complexity
• Enhance software quality
• Make the code easier and simpler to understand
• Make the code more flexible
• Make the code easier to change, etc…

NOTE: the importance of –
• Tests
• Metrics
Why Refactor? .... Revisited

• Long-term investment in the quality of the code and its structure

• No refactoring may save costs / time in the short term but incurs a huge penalty in the long run

• Why fix if it ain’t broken?
  Every module has three functions:
  1. To execute according to its purpose
  2. To afford change
  3. To communicate to its readers
  If it does not do one or more of these, it is broken.
When to Refactor?

1. NOT: 2 weeks every 6 months

2. Do it as you develop - Opportunistic Refactoring

3. Boy Scout principle: leave it better than you found it.

4. If you recognize a warning sign (a bad smell)
   I. When you add a function
      a) Before, to start clean and/or
      b) After, to clean-up
   II. When you fix a bug
   III. When you code review
   IV. You can use The Rule of Three (Fowler – XP)
      a) The first time, just do it!
      b) ! Need it somewhere else? Cut and paste it!
      c) ! The third time, refactor!
When **Not** to Refactor?

1. When the tests are failing  
2. When you should just rewrite the code  
3. When you have impending deadlines

How to Refactor?

![Refactoring Diagram]
Bad Smells in Code: *(Refactoring, chp3, Martin Fowler)*

- **Duplicated Code:** The same code structure in two or more places is a good sign that the code needs to be refactored: if you need to make a change in one place, you’ll probably need to change the other one as well, but you might miss it.
- **Long Method:** Long methods should be decomposed for clarity and ease of maintenance.
- **Large Class:** Classes that are trying to do too much often have large numbers of instance variables. Sometimes groups of variables can be clumped together. Sometimes they are only used occasionally. Over-large classes can also suffer from code duplication.
- **Long Parameter List:** Long parameter lists are hard to understand. You don't need to pass in everything a method needs, just enough so it can find all it needs.
- **Divergent Change:** Software should be structured for ease of change. If one class is changed in different ways for different reasons, it may be worth splitting the class in two so each one relates to a particular kind of change.
- **Shotgun Surgery:** If a type of program change requires lots of little code changes in various different classes, it may be hard to find all the right places that do need changing. Maybe the places that are affected should all be brought together into one class.
- **Feature Envy:** This is where a method on one class seems more interested in the attributes (usually data) of another class than in its own class. Maybe the method would be happier in the other class.
- **Data Clumps:** Sometimes you see the same bunch of data items together in various places: fields in a couple of classes, parameters to methods, local data. Maybe they should be grouped together into a little class.
Bad Smells in Code: *(Refactoring, chp3, Martin Fowler)*

- **Primitive Obsession:** Sometimes it's worth turning a primitive data type into a lightweight class to make it clear what it is for and what sort of operations are allowed on it (e.g., creating a date class rather than using a couple of integers).
- **Switch Statements:** Switch statements tend to cause duplication. You often find similar switch statements scattered through the program in several places. If a new data value is added to the range, you have to check all the various switch statements. Maybe classes and polymorphism would be more appropriate.
- **Parallel Inheritance Hierarchies:** In this case, whenever you make a subclass of one class, you have to make a subclass of another one to match.
- **Lazy Class:** Classes that are not doing much useful work should be eliminated.
- **Speculative Generality:** Often methods or classes are designed to do things that in fact are not required. The dead-wood should probably be removed.
- **Temporary Field:** It can be confusing when some of the member variables in a class are only used occasionally.
- **Message Chains:** A client asks one object for another object, which is then asked for another object, which is then asked for another, etc. This ties the code to a particular class structure.
- **Middle Man:** Delegation is often useful, but sometimes it can go too far. If a class is acting as a delegate, but is performing no useful extra work, it may be possible to remove it from the hierarchy.
- **Inappropriate Intimacy:** This is where classes seem to spend too much time delving into each other's private parts. Time to throw a bucket of cold water over them!
- **Alternative classes with different interfaces:** Classes that do similar things, but have different names, should be modified to share a common protocol.
- **Incomplete Library Class:** It's bad form to modify the code in a library, but sometimes they don't do all they should do.
- **Data Class:** Classes that just have data fields, and access methods, but no real behaviour. If the data is public, make it private!
- **Refused Bequest:** If a subclass doesn't want or need all of the behaviour of its base class, maybe the class hierarchy is wrong.
- **Comments:** If the comments are present in the code because the code is bad, improve the code.
Refactoring Drawbacks

• When taken too far
  • Incessant tinkering with code
  • Trying to make it *perfect*

• Attempting refactoring when the tests don’t work – or without tests – can lead to dangerous situations!

• Refactoring published interfaces propagates to external users relying on these interfaces
Why Developers Fear Refactoring?

1. “I don’t understand the code enough to do it”
2. Short-term focus (Adding a new working feature is cooler!)
3. Not paid for overhead tasks such as refactoring?

Solutions:

1. Test, test and test again, with good test code.
2. Learn to appreciate elegant code (and find good metrics)
3. Teach the benefits of better code (to your colleagues)
Recommended Reading


Recommended Reading


Java Refactoring – Built-in to Eclipse

Source -

Eclipse provides a general-purpose API for implementing refactorings that can be applied to any Eclipse workspace elements, from text resources to whole projects.

Several plugins for Eclipse draw upon the API to implement refactorings for specific languages; here we will focus on refactorings implemented for Java.

The Eclipse refactoring API, part of the Language Toolkit (LTK), is implemented within the org.eclipse.ltk.core.refactoring and org.eclipse.ltk.ui.refactoring plug-ins since R3.0

The API for refactoring provides a process-level abstraction upon which specific refactorings may be built.
Java Refactoring – Built-in to Eclipse

Dependencies between refactoring API elements
Java Refactoring – Built-in to Eclipse

A refactor implemener must extend the abstract class org.eclipse.ltk.core.refactoring.Refactoring

There is a life cycle associated to such refactorings:

A refactoring ultimately produces a single implementation of the abstract class Change that describes the workspace changes necessary to accomplish the refactoring. Implementers of refactorings must re-use or implement their own classes derived from Change to specify the behavior of a change.
Java Refactoring – Built-in to Eclipse

The Java part of Eclipse, JDT, is able to perform several types of automatic refactorings on Java projects, classes, and their members. There are several ways to quickly select a refactoring for an element in a Java project.

If you look in the Refactor dropdown menu in Eclipse, you will notice four sections:

The first section has Undo and Redo in it

The other three sections contain the three different types of refactorings available in Eclipse

1. contains refactorings that change the physical structure of the code and classes such as Rename and Move.

2. contains refactorings that change the code structure on a class level such as Pull Up and Push Down.

3. contains refactorings that change the code within a class such as Extract Method and Encapsulate Field
Java Refactoring – Built-in to Eclipse

Type 1 – Physical Structure
- Rename
- Move
- Change Method Signature
- Convert Anonymous Class to Nested
- Convert Nested Type to Top Level (Eclipse 2 only)
- Move Member Type to New File (Eclipse 3 only)

Type 2 – Class Level Structure
- Push Down
- Pull Up
- Extract Interface
- Generalize Type (Eclipse 3 only)
- User Supertype Where Possible

Type 3 – Structure inside a Class
- Inline
- Extract Method
- Extract Local Variable
- Extract Constant
- Introduce Parameter (Eclipse 3 only)
- Introduce Factory (Eclipse 3 only)
- Encapsulate Field

TO DO: Experiment with all the refactorings that are already in your version of Eclipse. You should become familiar with all of them.
Refactoring Catalog (http://refactoring.com/catalog/index.html)

- Add Parameter
- Change Bidirectional Association to Unidirectional
- Change Reference to Value
- Change Unidirectional Association to Bidirectional
- Change Value to Reference
- Collapse Hierarchy
- Consolidate Conditional Expression
- Consolidate Duplicate Conditional Fragments
- Convert Dynamic to Static Construction by Gerard M. Davison
- Convert Static to Dynamic Construction by Gerard M. Davison
- Decompose Conditional
- Duplicate Observed Data
- Eliminate Inter-Entity Bean Communication (Link Only)
- Encapsulate Collection
- Encapsulate Downcast
- Encapsulate Field
- Extract Class
- Extract Interface
- Extract Method
- Extract Package by Gerard M. Davison
- Extract Subclass
- Extract Superclass
- Form Template Method
- Hide Delegate
- Hide Method
- Hide presentation tier-specific details from the business tier (Link Only)
- Inline Class
- Inline Method
- Inline Temp
- Introduce A Controller (Link Only)
- Introduce Assertion
- Introduce Business Delegate (Link Only)
- Introduce Explaining Variable
- Introduce Foreign Method
- Introduce Local Extension
- Introduce Null Object
- Introduce Synchronizer Token (Link Only)
- Introduce Transaction
- Localize Disparate Logic (Link Only)
- Merge Session Beans (Link Only)
- Move Business Logic to Session (Link Only)
- Move Class by Gerard M. Davison
- Move Field
- Move Method
- Parameterize Method
- Preserve Whole Object
- Pull Up Constructor Body
- Pull Up Field
- Pull Up Method
- Push Down Field
- Push Down Method
- Reduce Scope of Variable by Mats Henrikson
- Refactor Architecture by Tern (Link Only)
- Remove Assignments to Parameters
- Remove Control Flag
- Remove Double Negative by Ashley Frieze and Martin Fowler
- Remove Middle Man
- Remove Parameter
- Remove Setting Method
- Rename Method
- Replace Array with Object
- Replace Assignment with Initialization by Mats Henrikson
- Replace Conditional with Polymorphism
- Replace Conditional with Visitor by Ivan Mitrovic
- Replace Constructor with Factory Method
- Replace Data Value with Object
- Replace Delegation with Inheritance
- Replace Error Code with Exception
- Replace Exception with Test
- Replace Inheritance with Delegation
- Replace Iteration with Recursion by Dave Whipp
- Replace Magic Number with Symbolic Constant
- Replace Method with Method Object
- Replace Nested Conditional with Guard Clauses
- Replace Parameter with Explicit Methods
- Replace Parameter with Method
- Replace Record with Data Class
- Replace Recursion with Iteration by Ivan Mitrovic
- Replace Static Variable with Parameter by Marian Vittek
- Replace Subclass with Fields...