Refactoring

CS 580 Software Specifications
Robert Stehwien

Outline

• What is Refactoring?
• Why Refactor?
• When to Refactor?
• How to Refactor?
• Tips and Opinions
• Examples

What is Refactoring?

• “Refactoring(noun): A change made to the internal structure of a software to make it easier to understand and cheaper to modify without changing its observable behavior”

• Disclaimer: Most slides are derived from this book
• www.refactoring.com
Outline

• What is Refactoring?
• Why Refactor?
• When to Refactor?
• How to Refactor?
• Tips and Opinions
• Examples

Why should you Refactor?

• Improves the design of software
  – the design of a program will decay over time as people change code to realize short-term goal
• Makes software easier to understand
  – by refactoring, software gets cleaned up
• Helps you to find bugs
  – clarifying the structure exposes bugs
• Helps you to program faster
  – as a result from the previous three points

Why people hesitate

• Conventional wisdom would discourage modifying a design
  – You might break something in the code
  – You have to update the documentation
  – => Both expensive
• But, there are longer term concerns sticking with an inappropriate design
  – Makes the code harder to change
  – Makes the code harder to understand and maintain
  – Very expensive in the long run
Why Refactoring is Important

• Only defense against software decay.
• Often needed to fix reusability bugs.
• Lets you add patterns after you have written a program; lets you transform program into framework.
• Lets you worry about generality tomorrow; just get it working today.
• Necessary for beautiful software.

Organic Growth

• Code is a liquid, not a solid!
• The way living things grow
• The way software grows
• More than just addition – transformation

Extreme Normal Form

• Your classes are small and your methods are small; you’ve said everything OnceAndOnlyOnce and you’ve removed the last piece of unnecessary code.
• Your code is in ExtremeNormalForm.. tuned to today and poised to strike at tomorrow.
• All the code is tested. The code passes all the tests.
• No one on the team can think of code in the system that they could remove or simplify without reducing the feature set.
• You never have to go more than one place to change one thing.
• http://c2.com/cgi/wiki?ExtremeNormalForm
Refactoring to Extreme Normal Form

- Extreme Normal Form is a result of refactoring
- Refactoring: changes to the organization of a program, not its function.
- Behavior preserving program transformations.

Outline

- What is Refactoring?
- Why Refactor?
- **When to Refactor?**
- How to Refactor?
- Tips and Opinions
- Examples

When to Refactor?

- If you do something similar the third time...
- When you add a feature
  - to understand the code or improve design
- When you need to fix a bug
  - clear code => no bugs :)
- As you do a code review
- If a new feature seems hard to implement
- If a new feature created some ugly code
- When you can’t stand to look at your code
- When you encounter a “bad smell”…
Bad Smells!

- “Bad Smells” in code give a hint to refactor
- Duplicate Code
  - The #1 bad smell!
- Long Method
- Large Class
- …
- Switch Statements (… breed duplication …)
- Comments (… not bad by itself …)
- … many more (23 in total)
- http://wiki.java.net/bin/view/People/SmellsToRefactorings

Smells Between Classes

- Primitive Obsession
- Data Class
- Data Clumps
- Refused Bequest
- Inappropriate Intimacy
- Lazy Class
- Feature Envy
- Message Chains
- Middle Man
- Divergent Change
- Shotgun Surgery
- Parallel Inheritance Hierarchies

Smells Within Classes

- Duplicated Code
- Comments
- Long Method
- Long Parameter List
- Large Class
- Type Embedded in Name
- Uncommunicative Name
- Inconsistent Names
- Dead Code
- Speculative Generality
Outline

• What is Refactoring?
• Why Refactor?
• When to Refactor?
• How to Refactor?
• Tips and Opinions
• Examples

How To Refactor

• Make changes as small as possible.
• Test after each change.

• Many small changes are easier than one big change.
• Each change makes it easier to see that more changes are needed.

Refactoring Steps

• Use write unit test suite
• Run unit tests and “keep your light green”.
• Plan change so it is a series of small refactoring.
• Run tests after every refactoring, once every 5 minutes.
List of Refactors


List of Refactors (1)

- Add Parameter
- Change Bidirectional Association to Unidirectional
- 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
- Encapsulate Collection
- Encapsulate Downcast
- Encapsulate Field
- Extract Class
- Extract Interface
- Extract Method
- Extract Package by Gerard M. Davison
- Extract Subclass
- Extract Supernode
- Form Template Method

List of Refactors (2)

- Hide Delegate
- Hide Method
- Hide presentation tier-specific details from the business tier
- Inline Class
- Inline Method
- Inline Temp
- Introduce A Controller
- Introduce Assertion
- Introduce Business Delegate
- Introduce Explaining Variable
- Introduce Foreign Method
- Introduce Local Extension
- Introduce Null Object
- Introduce Parameter Object
- Introduce Synchronizer Token
- Localize Disparate Logic
- Merge Session Beans
- Move Business Logic to Session
- Move Class by Gerard M. Davison
- Move Field
- Move Method
- Parameterize Method
- Preserve Whole Object
List of Refactors (3)

- Pull Up Constructor Body
- Pull Up Field
- Pull Up Method
- Push Down Field
- Push Down Method
- Reduce Scope of Variable by Mats Henricson
- Refactor Architecture by Tiers
- 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 Henricson
- 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

List of Refactors (4)

- 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
- Replace Temp with Query Replace Type Code with Class
- Replace Type Code with State/Strategy
- Replace Type Code with Subclasses
- Reverse Conditional by Bill Murphy and Martin Fowler
- Self Encapsulate Field
- Separate Data Access Code
- Separate Query from Modifier
- Split Loop by Martin Fowler
- Split Temporary Variable
- Substitute Algorithm
- Use a Connection Pool
- Wrap entities with session

Outline

- What is Refactoring?
- Why Refactor?
- When to Refactor?
- How to Refactor?
- Tips and Opinions
- Examples
Refactoring and Design Patterns

- “Refactoring to Patterns” by Joshua Kerievsky, Addison Wesley, 2005
- Provides insights into the combination of refactoring and design patterns
- Watch out for design-pattern smells!
  - e.g. Singletonitis: “addiction to the Singleton pattern”
- Refactoring can be used towards or away from certain design patterns.

Shortcomings with Refactoring

- External systems that can't be refactored
  - If you are using an external database or large code framework, those can't be refactored, limiting the improvements that can be made.
  - Note however that with shared code you can refactor your teammates’ code; this is one of the main strengths of having a shared codebase
- Published interfaces should change as little as possible
  - Refactoring often wants to change those interfaces.
  - There is a significant tension here.
- If the design is too awful
  - Its not worth trying to refactor
  - Throw it out!

Refactoring and Design

- Refactoring is almost as important in design as in implementation
  - Same principles apply, but at higher level.
- Don't overdesign, spending too much time on design
  - some problems are more clear when code takes shape.
Testing and Refactoring

- Unit testing is critical to refactoring. Fowler chapter 4 covers JUnit, the unit testing framework. We covered this in the TDD lecture.
- Here is the sequence of events to follow to test in the context of a refactoring:
  - Make sure you have unit tests covering the code you are about to change; if not write them
  - Make sure you are 100% compliant with tests before refactoring
  - Refactor
  - Re-run tests and get back to 100%
- If you have a series of refactorings in mind, re-test between each change.

Personal Opinion

- Most important software engineering technique since Design Patterns
- One of several Extreme Programming practices that can stand out alone (like e.g. automatic testing)
- Refactoring tools in IDE are almost a must
  - Eclipse

Hints

- Automated tests are required
- Refactoring is best done continuously
  - Trial & Error => Code & Refactor & Test
- Checking the long list of refactorings might help once in a while
- CVS does not support renaming, use SVN to keep file history
Outline

• What is Refactoring?
• Why Refactor?
• When to Refactor?
• How to Refactor?
• Tips and Opinions
• Examples

Refactoring Template

Name
• Summary
  – Situation in which you need the refactoring
  – What the refactoring does
  – Helps find appropriate refactoring quickly
• Motivation
  – Why the refactoring should be done
  – Circumstances in which it shouldn't be done
• Mechanics
  – Concise, step-by-step description on how to carry out refactoring
• Examples
  – Simple use of refactoring to illustrate how it works

Encapsulate Field

• Summary
  – There is a public field.
  – Make it private and provide accessors.
• Motivation
  – Encapsulation – data hiding
• Mechanics
  1. Create getter and setter
  2. Replace use of data in clients
  3. Compile and test after each change
  4. Declare field private
  5. Compile and test
• Example
  ```java
  public String _name
  ... Becomes ...
  private String _name;
  public String getName() {return _name;}
  public void setName(String arg) {_name = arg;}
  ```
• For more information see page 206 of Refactoring
Extract Method

• Summary
  – You have a code fragment that can be grouped together
  – Turn the fragment into a method whose name explains the purpose of the method

• Motivation
  – Method too long
  – Method needs a comment to understand purpose

• Mechanics
  1. Create new method, name after intention
  2. Copy extracted code into method
  3. See which variables need to be parameters
  4. See which variables are local to method
  5. See if parameters are changed
  6. Pass in parameters
  7. Compile
  8. Replace extracted code
  9. Compile and test

• For more information see page 110 of Refactoring

Extract Method Example

```java
void printOwing() {
    printBanner();
    //print details
    System.out.println("name: "+_name);
    System.out.println("amount "+getOutstanding());
}

... Becomes ...

void printOwing() {
    printBanner();
    printDetails(getOutstanding());
}

void printDetails (double outstanding) {
    System.out.println("name: "+_name);
    System.out.println("amount "+outstanding);
}
```