Refactoring and Code Maintenance

slides created by Marty Stepp
http://www.cs.washington.edu/403/

Problem: "Bit rot"

- After several months and new versions, many codebases reach one of the following states:
  - *rewritten* : Nothing remains from the original code.
  - *abandoned* : Original code is thrown out, rewritten from scratch.

- Why?
  - Systems evolve to meet new needs and add new features
  - If the code's structure does not also evolve, it will "rot"

  - This can happen even if the code was initially reviewed and well-designed at the time of checkin

Code maintenance

- **maintenance**: Modification or repair of a software product after it has been delivered.

  Purposes:
  - fix bugs
  - improve performance
  - improve design
  - add features

  - Studies have shown that ~80% of maintenance is for non-bug-fix-related activities such as adding functionality (Pigosky 1997).

Maintenance is hard

- It's harder to maintain code than write your own new code.
  - "house of cards" phenomenon (don't touch it!)
  - must understand code written by another developer, or code you wrote at a different time with a different mindset
  - most developers dislike code maintenance

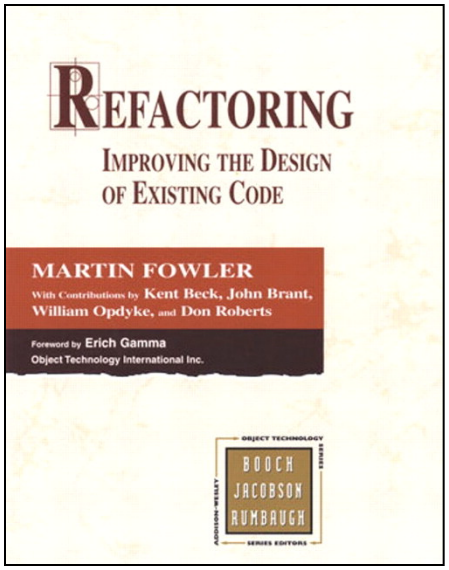
- Maintenance is how developers spend much of their time.

- It pays to design software well and plan ahead so that later maintenance will be less painful.
  - Capacity for future change must be anticipated

Refactoring

- **refactoring**: Improving a piece of software's internal structure without altering its external behavior.
  - Incurs a short-term time/work cost to reap long-term benefits
  - A long-term investment in the overall quality of your system.

- refactoring is not the same thing as:
  - adding features
  - debugging code
  - rewriting code



Why refactor?

- Why fix a part of your system that isn't broken?

- Each part of your system's code has 3 purposes:
  1. to execute its functionality,
  2. to allow change,
  3. to communicate well to developers who read it.

  – If the code does not do one or more of these, it is "broken."

- Refactoring:
  – improves software's design
  – makes it easier to understand

When to refactor?

- When is it best for a team to refactor their code?
  - best done **continuously** (like testing) as part of the process
  - hard to do well late in a project (like testing)

- Refactor when you identify an area of your system that:
  - isn't well designed
  - isn't thoroughly tested, but seems to work so far
  - now needs new features to be added

Signs you should refactor

- code is **duplicated**
- a routine is **too long**
- a loop is too long or **deeply nested**
- a class has poor **cohesion**
- a class uses too much **coupling**
- inconsistent level of **abstraction**

- too many **parameters**
- to **compartmentalize** changes (change one place → must change others)
- to modify an **inheritance hierarchy** in parallel
- to **group related data** into a class
- a **"middle man"** object doesn't do much
- **poor encapsulation** of data that should be private
- a **weak subclass** doesn't use its inherited functionality
- a class contains **unused code**

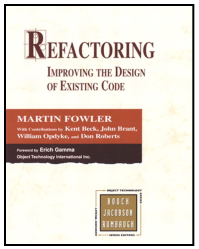
Code "smells"

- Duplicated Code
- Long Method
- Large Class
- Long Parameter List
- Divergent Change
- Shotgun Surgery
  - (change one place → must change others)
- Feature Envy
- Data Clumps
- Primitive Obsession
- Switch Statements
- Parallel Inheritance Hierarchies
- Lazy Class
- Speculative Generality
- Temporary Field
- Message Chains
- Middle Man
- Inappropriate Intimacy
- Alternative Classes with Different Interfaces
- Incomplete Library Class
- Data Class
- Refused Bequest
  - (subclass doesn't use inherited members much)
- Comments

Some types of refactoring

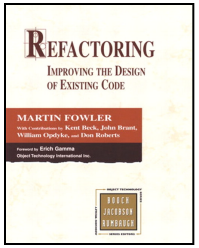
- refactoring to fit design **patterns**
- **renaming** (methods, variables)
- **extracting** code into a method or module
- **splitting** one method into several to improve cohesion and readability
- changing method **signatures**
- performance **optimization**
- **moving** statements that semantically belong together near each other
- naming (extracting) "magic" **constants**
- **exchanging idioms** that are risky with safer alternatives
- **clarifying** a statement that has evolved over time or is unclear

  - See also http://www.refactoring.org/catalog/



Refactoring patterns

- Composing Methods
  - Extract Method
  - Inline Method; Inline Temp
  - Replace Temp with Query
  - Introduce Explaining Variable
  - Split Temporary Variable
  - Remove Assignments to Parameters
  - Replace Method with Method Object
  - Substitute Algorithm
- Moving Features Between Objects
  - Move Method; Move Field
  - Extract Class
  - Inline Class
  - Hide Delegate
  - Remove Middleman
  - Introduce Foreign Method
  - Introduce Local Extension
- Organizing Data
  - Self-encapsulate Field
  - Replace Data Value with Object
  - Change Value to Reference; Change Reference to Value
  - Replace Array with Object
  - Duplicate Observed Data
  - Change Unidirectional Association to Bidirectional
  - Change Bidirectional Association to to Unidirectional
- Simplifying Conditional Expressions
  - Decompose Conditional
  - Consolidate Conditional Expression
  - Consolidate Duplicate Conditional Fragments
  - Remove Control Flag
  - Replace Nested Conditional with Guard Clauses
  - Replace Conditional with Polymorphism
  - Introduce Null Object
  - Introduce Assertion
- Making Method Calls Simpler
  - Rename Method
  - Add/Remove Parameter
  - Separate Query from Modifier
  - Parameterize Method
  - Replace Parameter with Explicit Methods
  - Preserve Whole Object
  - Replace Parameter with Method
  - Introduce Parameter Object
  - Remove Setting Method
  - Hide Method
  - Replace Constructor with Factory Method
  - Encapsulate Downcast
  - Replace Error Code with Exception
  - Replace Exception with Test
- Dealing with Generalization
  - Pull Up Field; Method; Constructor Body
  - Push Down Method; Push Down Field
  - Extract Subclass; Extract Superclass; Interface
  - Collapse Hierarchy
  - Form Template Method
  - Replace Inheritance with Delegation (or vice versa)
- Big Refactorings
  - Nature of the Game
  - Tease Apart Inheritance
  - Convert Procedural Design to Objects
  - Separate Domain from Presentation
  - Extract Hierarchy



Refactoring examples

- **Encapsulate Downcast:** A method returns an object that needs to be downcasted by its callers. Refactor by moving the downcast to within the method.

```
Object lastReading() {
        return readings.lastElement();
}
```

```
Reading lastReading() {
        return (Reading) readings.lastElement();
}
```

- **Consolidate Conditional Expression:** You have a sequence of conditional tests with the same result. Refactor by combining them into a single conditional expression and extract it.

```
double disabilityAmount() {
        if (_seniority < 2) return 0;
        if (_monthsDisabled > 12) return 0;
        if (_isPartTime) return 0;
        // compute the disability amount
```

```
double disabilityAmount() {
        if (isNotEligibleForDisability()) return 0;
        // compute the disability amount
```

- **Consolidate Duplicate Conditional Fragments:** The same fragment of code is in all branches of a conditional expression. Refactor by moving it outside of the expression.

```
if (isSpecialDeal()) {
        total = price * 0.95;
        send();
} else {
        total = price * 0.98;
        send();
}
```

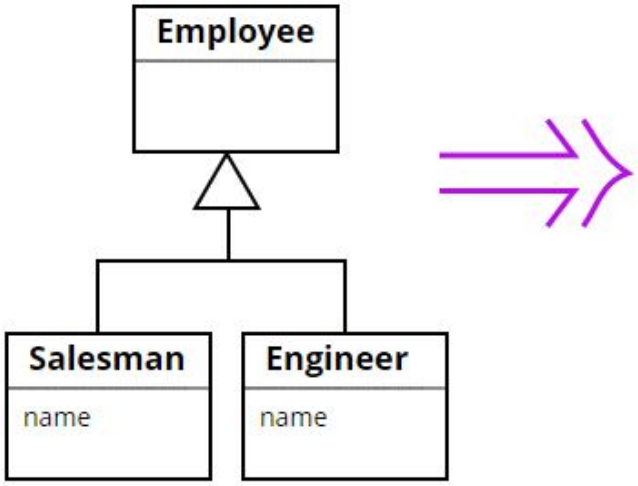
```
if (isSpecialDeal())
        total = price * 0.95;
else
        total = price * 0.98;
send();
```

- **Rename Method:** The name of a method does not reveal its purpose. Refactor it by changing the name of the method.

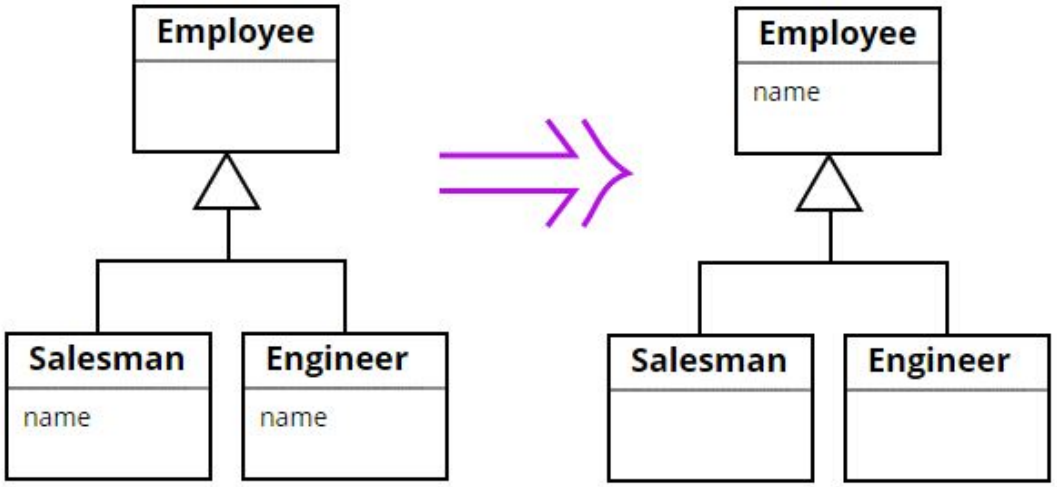
```
int getInvCdtLmt(){
…
}
```

```
int getInvoiceableCreditLimit(){
…
}
```

- **Pull Up Field:** Two subclasses have the same field. Refactor it by moving the field to the superclass.

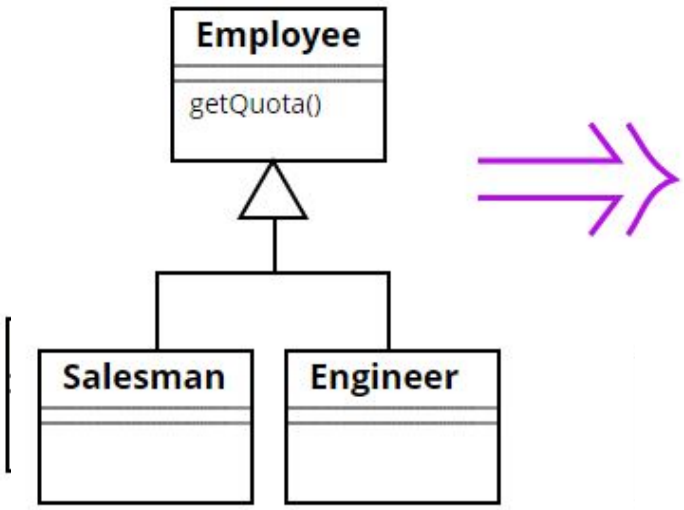


- **Pull Up Field:** Two subclasses have the same field. Refactor it by moving the field to the superclass.



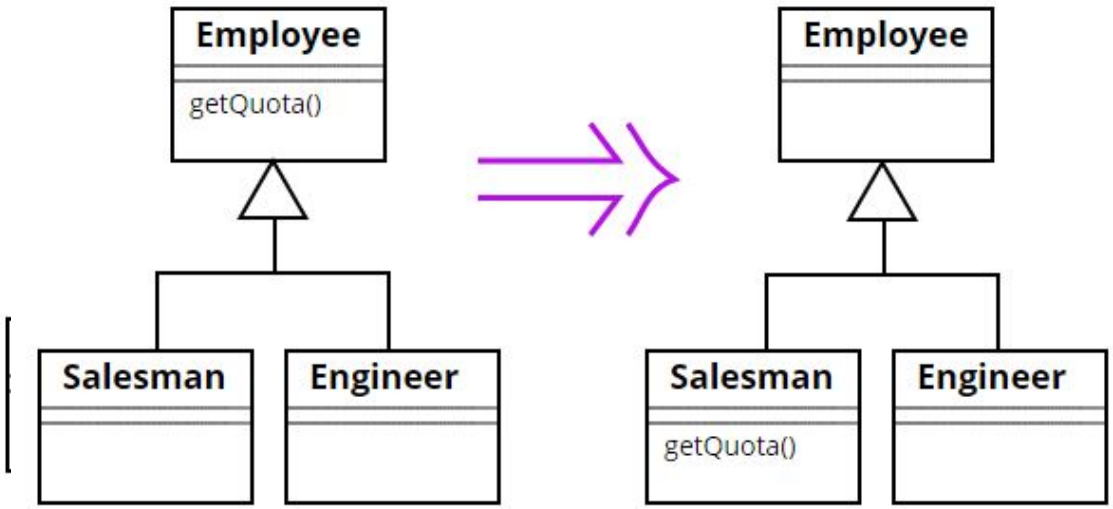
Refactoring: examples

- **Push Down Method:** Behavior on a superclass is relevant only for some of its subclasses. Refactor it by moving it to those subclasses.



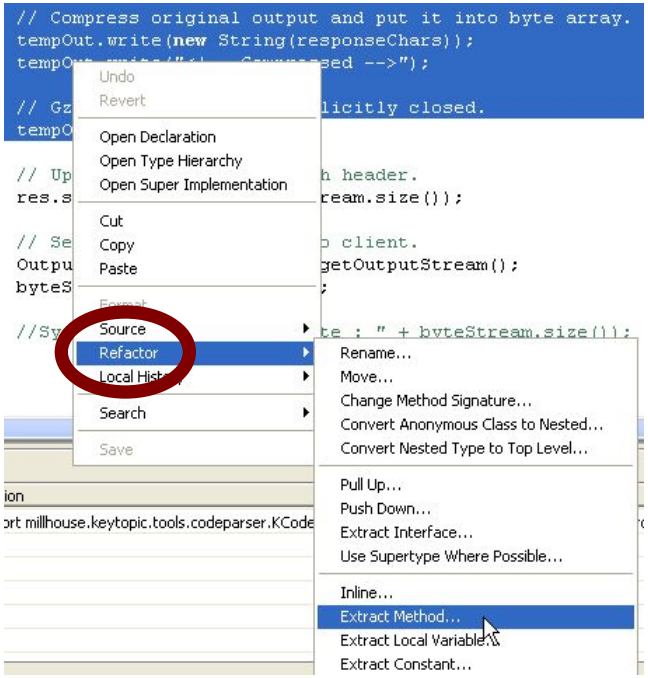
Refactoring: examples

- **Push Down Method:** Behavior on a superclass is relevant only for some of its subclasses. Refactor it by moving it to those subclasses.



IDE support for refactoring

- Eclipse and Visual Studio support:
  - variable / method / class renaming
  - method or constant extraction
  - extraction of redundant code snippets
  - method signature change
  - extraction of an interface from a type
  - method inlining
  - providing warnings about method invocations with inconsistent parameters
  - help with self-documenting code through auto-completion



Refactoring plan, pt 1

- Save / **backup** / checkin the code before you mess with it.
  - If you use a well-managed version control repo, this is done.

- Write **unit tests** that verify the code's external correctness.
  - They should pass on the current poorly designed code.
  - Having unit tests helps make sure any refactor doesn't break existing behavior (regressions).

- Analyze the code to decide the **risk** and benefit of refactoring.
  - If it is too risky, not enough time remains, or the refactor will not produce enough benefit to the project, don't do it.

Refactoring plan, pt 2

(after completing steps on the previous slide)

- **Refactor** the code.
  - Some unit tests may break. Fix the bugs.
  - Perform functional and/or integration testing. Fix any issues.
- **Code review** the changes.
- **Check in** your refactored code.
  - Keep each refactoring **small**; refactor one issue / unit at a time.
    - helps isolate new bugs and regressions
  - Your checkin should contain *only* your refactor.
    - NOT other changes such as adding features, fixing unrelated bugs.
    - Do those in a separate checkin.
      - (Resist temptation to fix small bugs or other tweaks; this is dangerous.)

"I don't have time!"

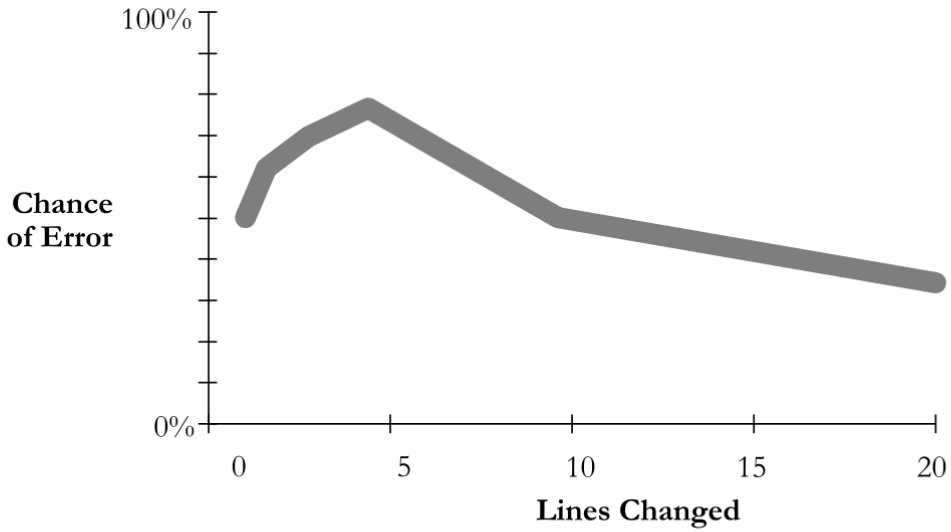
- Refactoring incurs an **up-front cost**.
  - many developers don't want to do it
  - management don't like it; they lose time and gain "nothing" (no new features)

- But...
  - well-written code is more conducive to **rapid development** (some estimates put ROI at 500% or more for well-done code)
  - refactoring is good for **programmer morale**
    - developers prefer working in a "clean house"

Dangers of refactoring

- code that used to be ...
  - well commented, now (maybe) isn't
  - fully tested, now (maybe) isn't
  - fully code reviewed, now (maybe) isn't

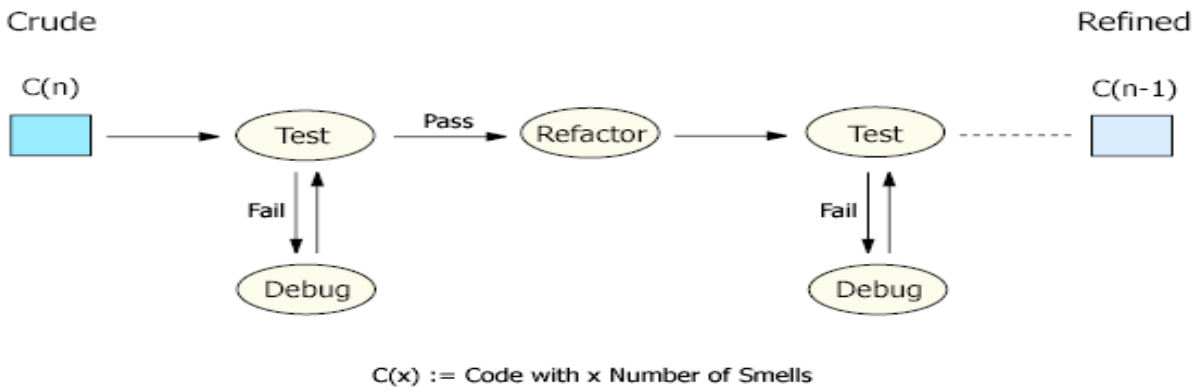
- easy to insert a bug into previously working code (regression!)
  - a small initial change can have a large chance of error



Regressions

What if refactoring introduces new bugs in previously working functionality ("regressions")? How can this be avoided?

- Each refactoring is implemented as a small behavior-preserving transformation.
- Behavior-preservation is achieved through pre- and post-transformation testing.
- Refactoring process: test-refactor-test



Regressions

What if refactoring introduces new bugs in previously working functionality ("regressions")? How can this be avoided?

- Code being refactored should have good **unit test** coverage, and other tests (system, integration) over it, *before* the refactor.
  - If such code is not tested, **add tests** first *before* refactoring.
  - If the refactor makes a unit test not compile, **port it**.
  - If the method being tested goes away, the underlying functionality of that method should still be somewhere. So **move the unit test** to cover that new place in the code.

Company/team culture

- **Organizational barriers** to refactoring:
  - Many small companies and startups don't do it.
    - "We're too small to need it!" ... or, "We can't afford it!"
  - Many larger companies don't adequately reward it.
    - Not as flashy as adding features/apps; ignored at promotion time

- Reality:
  - Refactoring is an investment in quality of the company's product and code base, often their prime assets.
  - Many web startups are using the most cutting-edge technologies, which evolve rapidly. So should the code.
  - If a team member leaves or joins (common in startups), ...
  - Some companies (e.g. **Google**) actively reward refactoring.

Refactoring and teams

- Amount of overhead/communication needed depends on size of refactor.
  - *small refactor:* Just do it, check it in, get it code reviewed.
  - *medium refactor:* Possibly loop in tech lead or another dev.
  - *large refactor:* Meet with team, flush out ideas, do a design doc or design review, get approval before beginning.

- Avoids possible bad scenarios:
  - Two devs refactor same code simultaneously.
  - Refactor breaks another dev's new feature they are adding.
  - Refactor actually is not a very good design; doesn't help.
  - Refactor ignores future use cases, needs of code/app.
  - Tons of merge conflicts and pain for other devs.

Phased refactoring

- Sometimes a refactor is too big to do all at once.
  - Example: An entire large subsystem needs redesigning. We don't think we have time to redo all of it at once.

- **phased refactoring**:
  Adding a **layer of abstraction** on top of legacy code.
  - New well-made System 2 on top of poorly made old System 1.
  - System 1 remains; Direct access to it is *deprecated*.
  - For now, System 2 still forwards some calls down to System 1 to achieve feature parity.
  - Over time, calls to System 1 code are replaced by new System 2 code providing the same functionality with a better design.

Refactoring at Google

- "At Google, refactoring is very important and necessary/inevitable for any code base. If you're writing a new app quickly and adding lots of features, your initial design will not be perfect. Ideally, do *small* refactoring tasks early and often, as soon as there is a sign of a problem."
- "Refactoring is unglamorous because it does not add features. At many companies, people don't refactor because you don't get promoted for it, and their code turns into hacky beasts."
- "Google feels refactoring is so important that there are company-wide initiatives to make sure it is encouraged and rewarded."
- "Common reasons not to do it are incorrect:
  - a) *'Don't have time; features more important'* -- You will pay more cost, time in adding features (because it's painful in current design), fixing bugs (because bad code is easy to add bugs into), ramping up others on code base (because bad code is hard to read), and adding tests (because bad code is hard to test), etc.
  - b) *'We might break something'* -- Sign of a poor design from the beginning, where you didn't have good tests. For same reasons as above, you should fix your testing situation and code.
  - c) *'I want to get promoted and companies don't recognize refactoring work'* -- This is a common problem. Solution varies depending on company. Seek buy-in from your team, gather data about regressions and flaws in the design, and encourage them to buy-in to code quality."
- "An important line of defense against introducing new bugs in a refactor is having solid unit tests (*before* the refactor)."

-- Victoria Kirst,
Software Engineer, Google