

CS 563:
Software Maintenance and Evolution

Software Refactoring

Oregon State University, Spring 2024

Today's Plan

- In-class exercise: project stand-up meetings (30 min)
 - Please sit together with your project team
 - I will be visiting each group for 5 min to see/know:
 1. Rough-sketch of your finalized approach
 2. Finalized dataset and evaluation metrics
 3. Implementation plan details
 4. GitHub repo for your project with me added as a contributor
- Learn about Software Refactoring

Problem

- 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.

Problem

- 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.

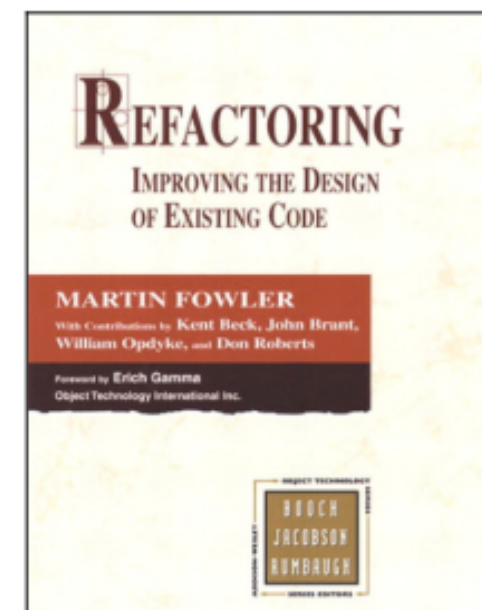
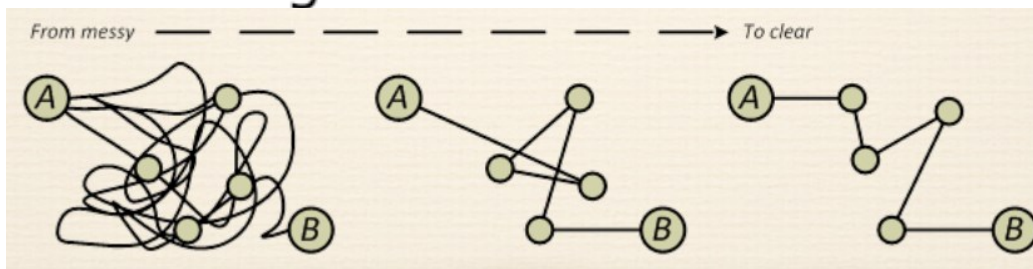
- Why?
 - Systems evolve to meet new needs and add new features
 - If the code's structure does not also evolve, it will "rot"



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

Software 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

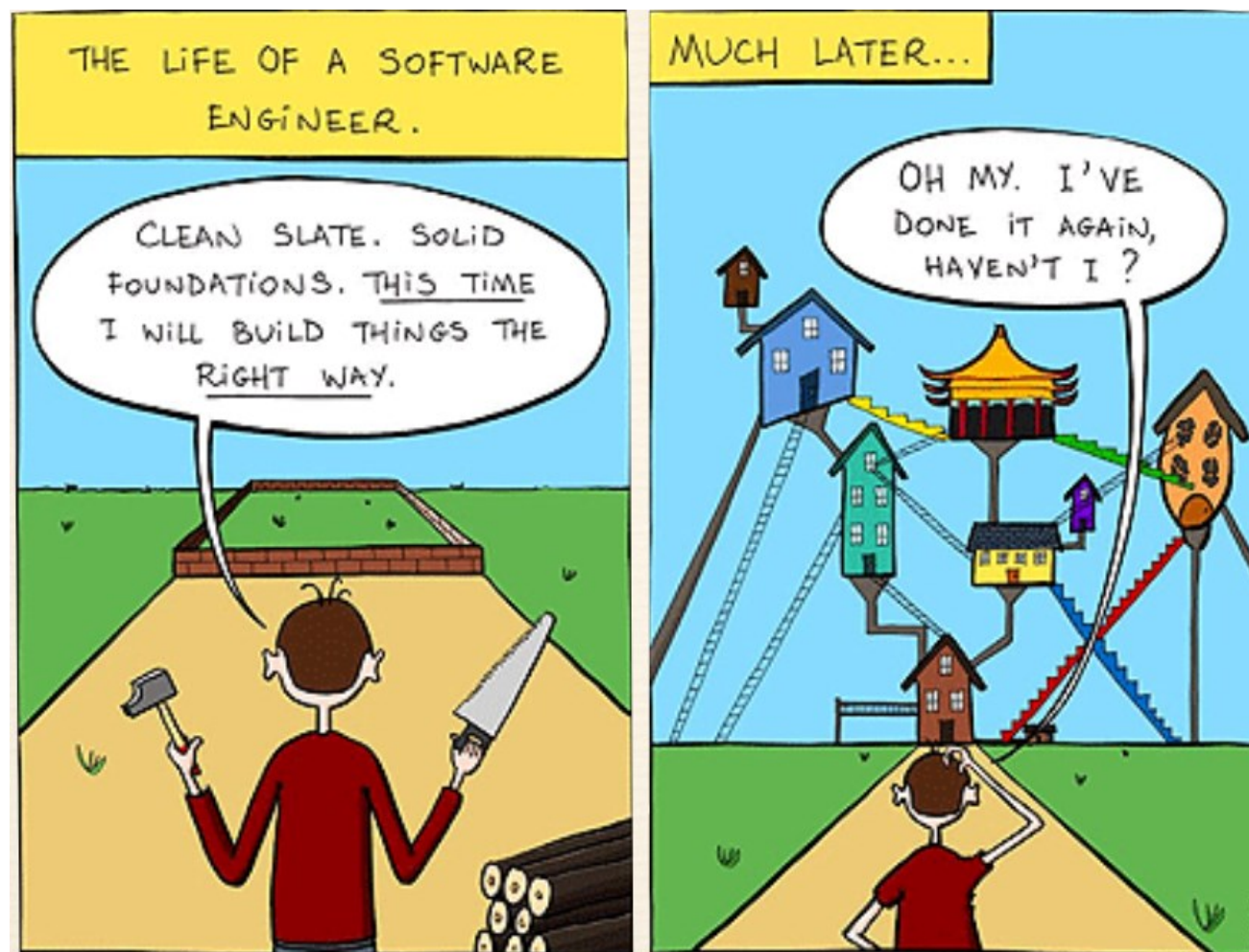


Why Refactor?

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

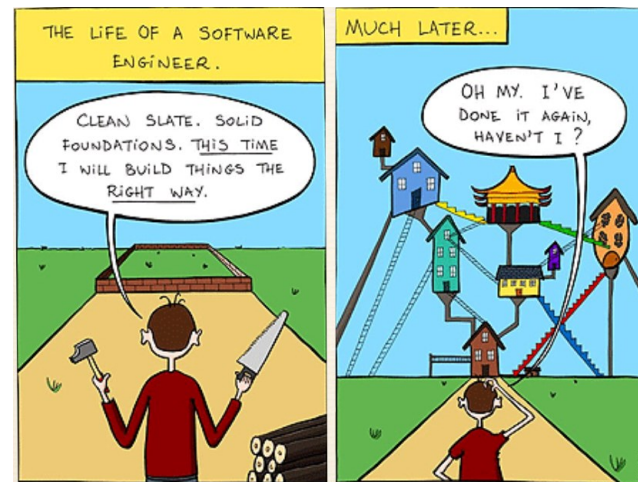
Why Refactor?

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



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
- Rule of three
 - 1st time:** implement from scratch
 - 2nd time:** implement something similar by code duplication
 - 3rd time:** do not implement similar things again, but refactor



When to Refactor?

- Refactor when adding new features or functions
 - Especially if feature is difficult to integrate with the existing code
- Refactor during bug fixing
 - If a bug is very hard to trace, refactor first to make the code more understandable, so that you can understand better where the bug is located
- Refactor during code reviews

When to Refactor?

- Refactor when adding new features or functions
 - Especially if feature is difficult to integrate with the existing code
- Refactor during bug fixing
 - If a bug is very hard to trace, refactor first to make the code more understandable, so that you can understand better where the bug is located
- Refactor during code reviews

What do you tell the manager?

- When (s)he's technically aware, (s)he'll understand why refactoring is important.
- When (s)he's interested in quality, (s)he'll understand that refactoring will improve software quality.
- When (s)he's only interested in the schedule, don't tell that you're doing refactoring, just do it anyway.
 - In the end refactoring will make you more productive.



When shouldn't you refactor?

- When the existing code is such a mess that although you could refactor it, it would be easier to rewrite everything from scratch instead.
- When you are too close to a deadline.
 - The productivity gain would appear after the deadline and thus be too late.
 - However, when you are **not** close to a deadline you should **never** put off refactoring because you don't have the time.
 - Not having enough time usually is a sign that refactoring is needed.

Signs you should refactor

- code is **uplicated**
- 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

Code “smells”

- **Duplicated Code**

- bad because if you modify one instance of duplicated code but not the others, you (may) have introduced a bug!

- **Long Method**

- long methods are more difficult to understand
 - performance concerns with respect to lots of short methods are largely obsolete

Code “smells”

- **Large Class**
 - Large classes try to do too much, which reduces cohesion
- **Long Parameter List**
 - hard to understand, can become inconsistent if the same parameter chain is being passed from method to method
- **Divergent Change**
 - symptom: one type of change requires changing one subset of methods; another type of change requires changing another subset
 - Related to cohesion

Code “smells”

- **Shotgun Surgery**

- a change requires lots of little changes in a lot of different classes

- **Feature Envy**

- A method requires lots of information from some other class
 - move it closer!

- **Data Clumps**

- attributes that clump together (are used together) but are not part of the same class

Code “smells”

- **Primitive Obsession**

- characterized by a reluctance to use classes instead of primitive data types

- **Switch Statements**

- Switch statements are often duplicated in code; they can typically be replaced by use of polymorphism (let OO do your selection for you!)

- **Parallel Inheritance Hierarchies**

- Similar to Shotgun Surgery; each time I add a subclass to one hierarchy, I need to do it for all related hierarchies
 - Note: some design patterns encourage the creation of parallel inheritance hierarchies (so they are not always bad!)

Code “smells”

- **Lazy Class**

- A class that no longer “pays its way”
 - e.g. may be a class that was downsized by a previous refactoring, or represented planned functionality that did not pan out

- **Speculative Generality**

- “Oh I think we need the ability to do this kind of thing someday”

- **Temporary Field**

- An attribute of an object is only set/used in certain circumstances;
 - but an object should need all of its attributes

Code “smells”

- **Message Chains**

- a client asks an object for another object and then asks that object for another object etc. Bad because client depends on the structure of the navigation

- **Middle Man**

- If a class is delegating more than half of its responsibilities to another class, do you really need it? Involves trade-offs, some design patterns encourage this (e.g. Decorator)

- **Inappropriate Intimacy**

- Pairs of classes that know too much about each other's implementation details (loss of encapsulation; change one class, the other has to change)

Code “smells”

- **Data Class (information holder)**
 - These are classes that have fields, getting and setting methods for the fields, and nothing else; they are data holders, but objects should be about data AND behavior
- **Refused Bequest**
 - A subclass ignores most of the functionality provided by its superclass
 - Subclass may not pass the “IS-A” test
- **Comments (!)**
 - Comments are sometimes used to hide bad code
 - “...comments often are used as a deodorant” (!)

Categories of Refactoring (according to [Fowler2000])

Small refactorings

(de)composing methods

moving features between objects

organising data

simplifying conditional expressions

dealing with generalisation

simplifying method calls

Big refactorings

Tease apart inheritance

Extract hierarchy

Convert procedural design to objects

Separate domain from presentation

Small Refactorings

1. (de)composing methods [9 refactorings]
2. moving features between objects [8 refactorings]
3. organizing data [16 refactorings]
4. simplifying conditional expressions [8 refactorings]
5. dealing with generalisation [12 refactorings]
6. simplifying method calls [15 refactorings]

Small Refactorings

1. (de)composing methods [9 refactorings]

1. Extract Method
2. Inline Method
3. Inline Temp
4. Replace Temp With Query
5. Introduce Explaining Variable
6. Split Temporary Variable
7. Remove Assignments to Parameter
8. Replace Method With Method Object
9. Substitute Algorithm

(de)composing methods:

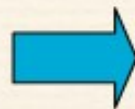
1. Extract Method

What? When you have a fragment of code that can be grouped together, turn it into a method with a name that explains the purpose of the method

Why? improves clarity, removes redundancy

Example:

```
public void accept(Packet p) {  
    if ((p.getAddressee() == this) &&  
        (this.isASCII(p.getContents())))  
        this.print(p);  
    else  
        super.accept(p); }  
    
```



```
public void accept(Packet p) {  
    if this.isDestFor(p) this.print(p);  
    else super.accept(p); }  
public boolean isDestFor(Packet p) {  
    return  
        ((p.getAddressee() == this) &&  
         (this.isASCII(p.getContents()))); }  
    
```

Beware of local variables!

(de)composing methods:

2. Inline Method


(Opposite of Extract Method)

What? When a method's body is just as clear as its name, put the method's body into the body of its caller and remove the method

Why? To remove too much indirection and delegation

Example:

```
int getRating() {  
    return moreThanFiveLateDeliveries();  
}  
  
moreThanFiveLateDeliveries() {  
    return _numberOfLateDeliveries > 5;  
}
```



```
int getRating() {  
    return (_numberOfLateDeliveries > 5);  
}
```


(de)composing methods:

5. Introduce Explaining Variable

What? When you have a complex expression, put the result of the (parts of the) expression in a temporary variable with a name that explains the purpose

Why? Breaking down complex expressions for clarity

Example:

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



```
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){  
  //ACTION  
}
```

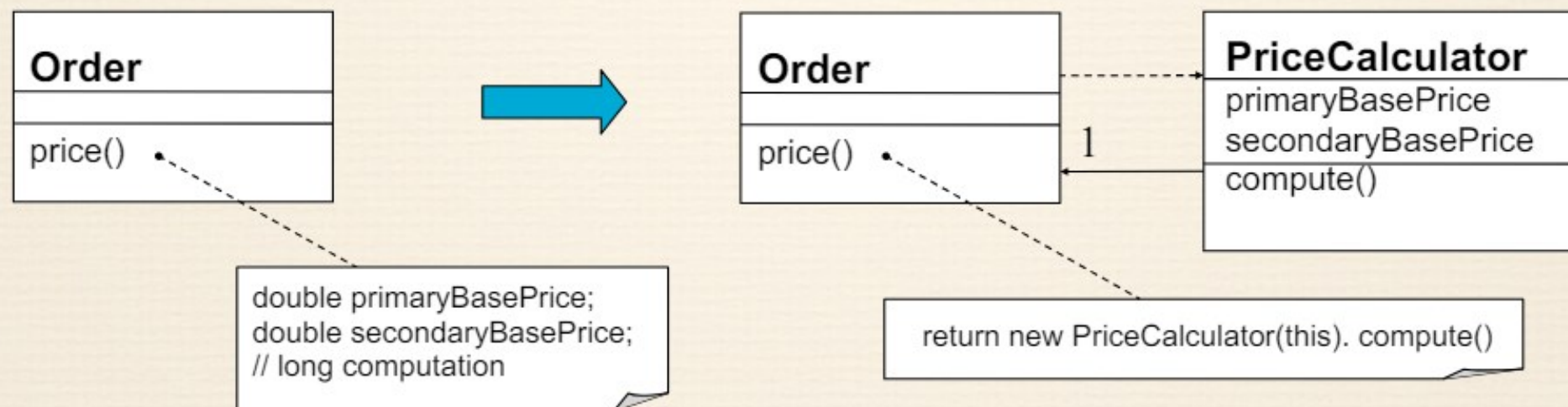
(de)composing methods:

8. Replace Method With Method Object

What? When you have local variables but cannot use **extract method**, turn the method into its own object, with the local variables as its fields

Why? Extracting pieces out of large methods makes things more comprehensible

Example:



Small Refactorings

1. (de)composing methods [9 refactorings]
- 2. moving features between objects [8 refactorings]**
 1. Move Method
 2. Move Field
 3. Extract Class
 4. Inline Class
 5. Hide Delegate
 6. Remove Middle Man
 7. Introduce Foreign Method
 8. Introduce Local Extension

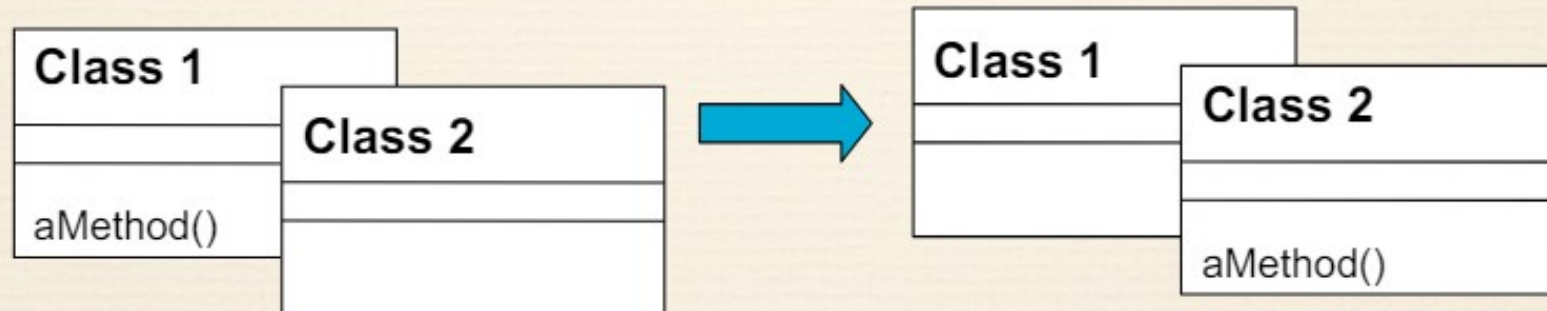
Moving Features Between Objects:

1,2. Move Method / Field

What? When a method (resp. field) is used by or uses more features of another class than its own, create a similar method (resp. field) in the other class; remove or delegate original method (resp. field) and redirect all references to it.

Why? Essence of refactoring

Example:



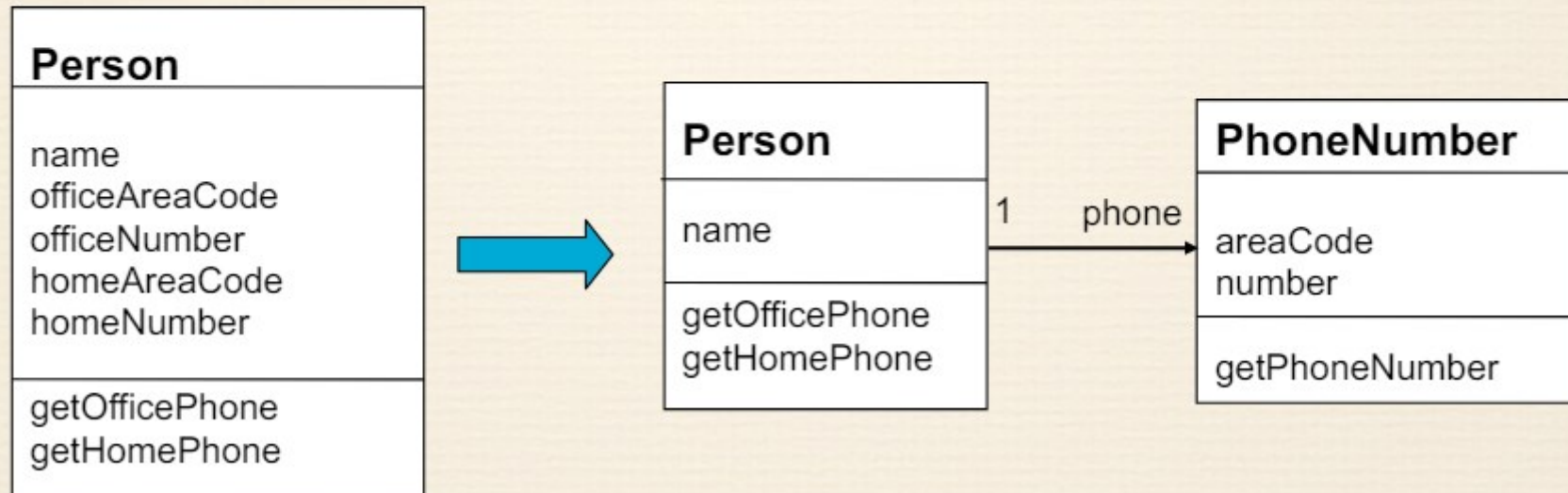
Moving Features Between Objects:

3. Extract Class

What? When you have a class doing work that should be done by two, create a new class and move the relevant fields and methods to the new class

Why? Large classes are hard to understand

Example:



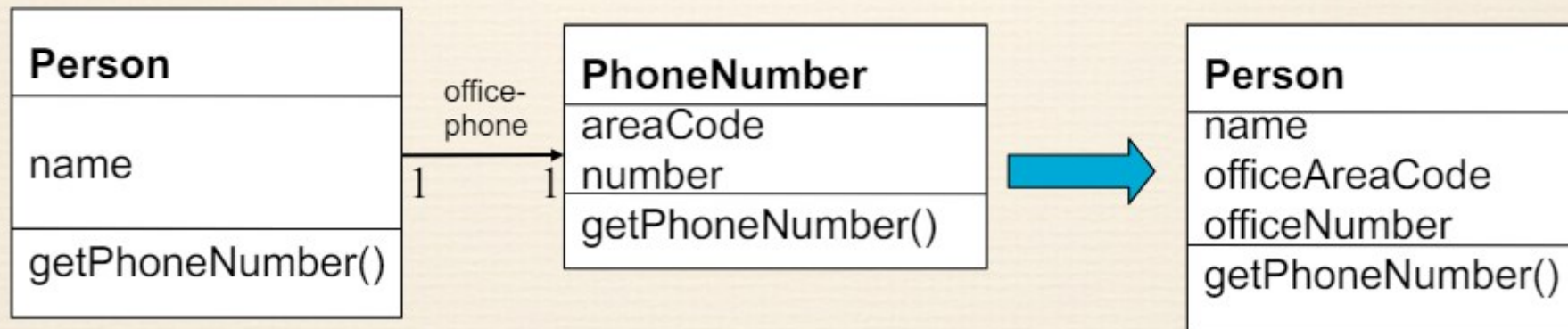
Moving Features Between Objects:

4. Inline Class

What? When you have a class that does not do very much, move all its features into another class and delete it

Why? To remove useless classes (as a result of other refactorings)

Example:



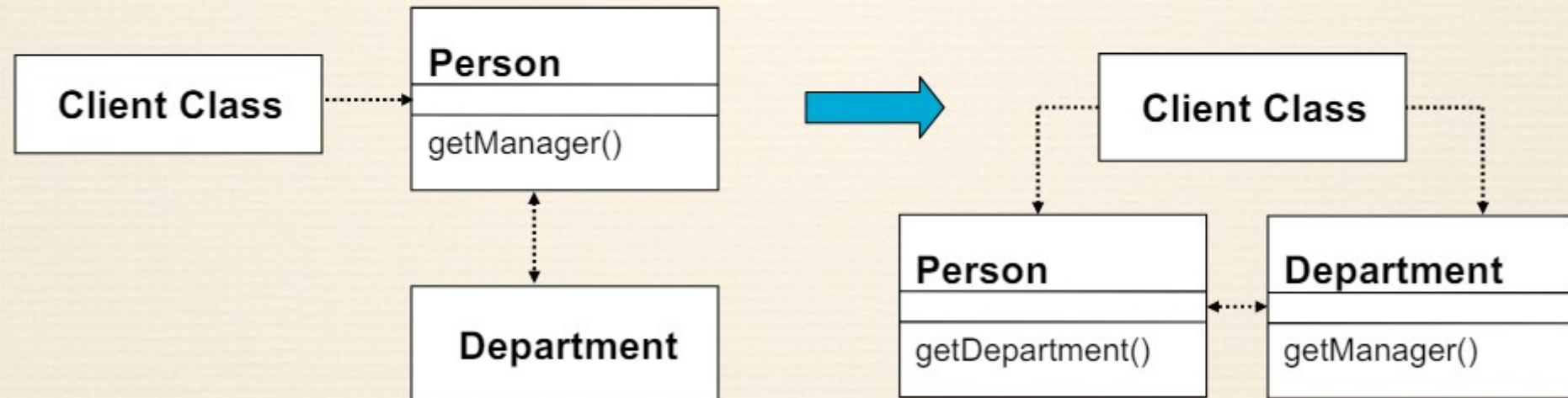
Moving Features Between Objects:

6. Remove Middle Man

What? When a class is doing too much simple delegation, get the client to call the delegate directly

Why? To remove too much indirection (as a result of other refactorings)

Example:



Small Refactorings

1. (de)composing methods [9 refactorings]
2. moving features between objects [8 refactorings]
- 3. organizing data [16 refactorings]**
 1. Encapsulate field
 2. Replace data value with object
 3. Change value to reference
 4. Change reference to value
 5. Replace array with object
 6. Duplicate observed data
 7. Change unidirectional association to bidirectional
 8. Change bidirectional association to unidirectional
 9. Replace magic number with symbolic constant
 10. Encapsulate collection
 11. Replace record with data class
 12. Replace subclass with fields
 13. Replace type code with class / subclass / state / strategy (14—16)

Organizing Data

1. Encapsulate Field

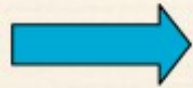
What? There is a public field. Make it private and provide accessors.

Why? Encapsulating state increases modularity, and facilitates code reuse and maintenance.

When the state of an object is represented as a collection of private variables, the internal representation can be changed without modifying the external interface

Example:

```
public String name;
```



```
private String name;  
public String getName() {  
    return this.name; }  
public void setName(String s) {  
    this.name = s; }
```

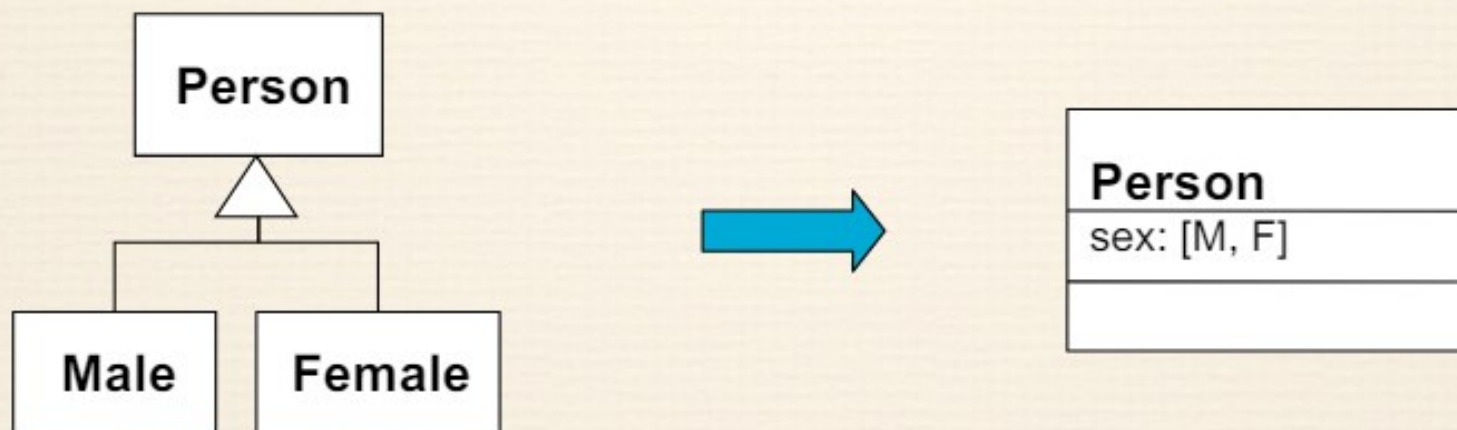

Organizing Data

12. Replace Subclass with Fields

What? Subclasses vary only in methods that return constant data

Solution: Change methods to superclass fields and eliminate subclasses

Example:



Organizing Data

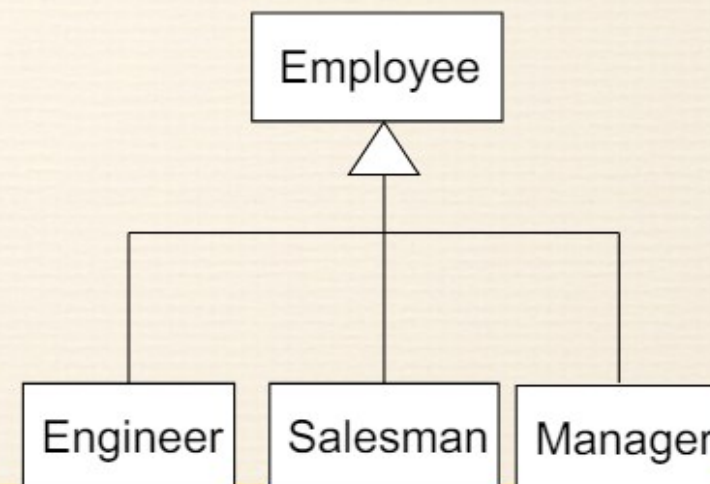
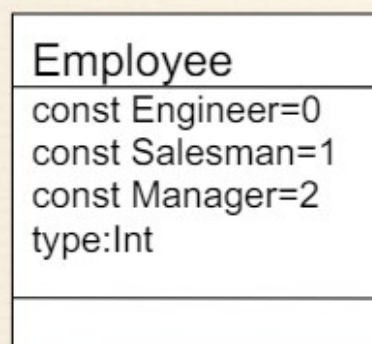
13. Replace Type Code with Subclass

PROBLEM

YOU HAVE A CODED TYPE FIELD OF WHICH THE VALUES DIRECTLY AFFECT TRIGGER DIFFERENT BEHAVIOUR IN CONDITIONALS.

What? An immutable type code affects the behaviour of a class

Example:



SOLUTION

CREATE SUBCLASSES FOR EACH VALUE OF THE CODED TYPE.
 EXTRACT RELEVANT BEHAVIORS FROM THE ORIGINAL CLASS TO THESE SUBCLASSES.
 REPLACE THE CONTROL FLOW CODE WITH POLYMORPHISM.

Categories of Refactoring (according to [Fowler2000])

Small refactorings

(de)composing methods [9]

moving features between objects [8]

organising data [16]

simplifying conditional expressions [8]

dealing with generalisation [12]

simplifying method calls [15]

Big refactorings

Tease apart inheritance

Extract hierarchy

Convert procedural design to objects

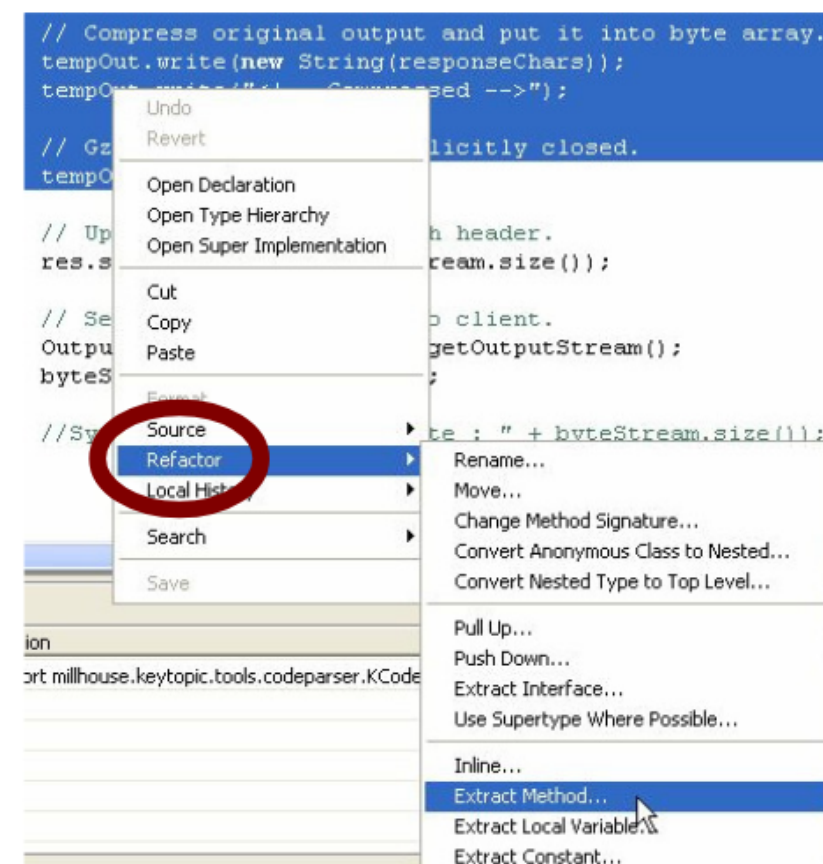
Separate domain from presentation

Big Refactorings

- Require a large amount of time (> 1 month)
- Require a degree of agreement among the development team
- No instant satisfaction, no visible progress

Refactoring Tools

- Available for all major programming languages and OO programming languages in particular)
- **Java** : IntelliJ IDEA, Eclipse, NetBeans, JDeveloper, ...
- **JavaScript** : WebStorm, Eclipse, ...
- **C++** : VisualStudio, Eclipse, ...
- **ObjectiveC** and **SWIFT** : XCode
- **.NET** : VisualStudio
- **Almost all**: Copilot



Refactoring Plan (part 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 (part 2)

(after completing steps on the previous slide)

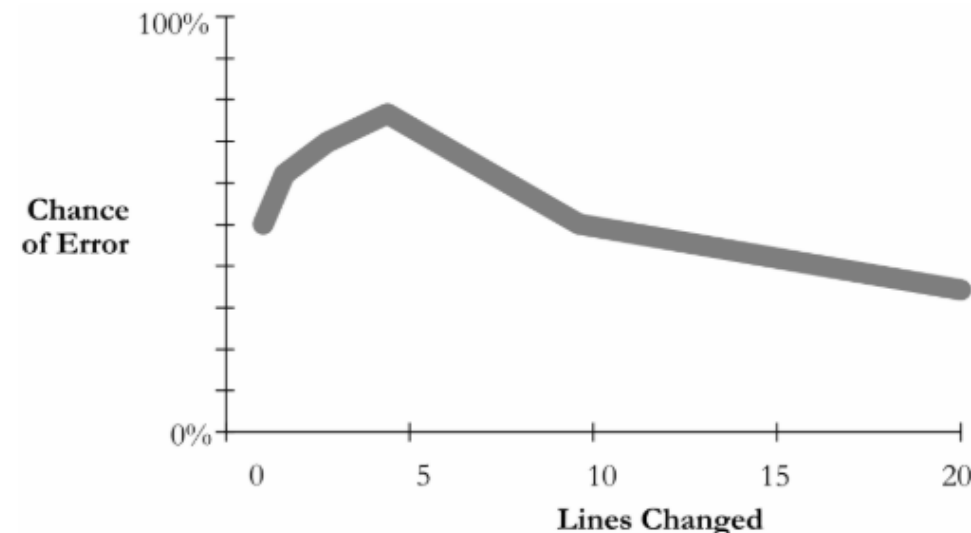
- **Refactor** the code.
 - Some unit tests may break. Fix the bugs.
 - Perform functional and/or integration testing. Fix any issues.
- **Code review** the changes.
- **Check in** your refactored code.
 - Keep each refactoring **small**; refactor one issue / unit at a time.
 - helps isolate new bugs 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.

“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?

- 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."
- "Your most important line of defense against introducing new bugs in a refactor is having solid unit tests (before the refactor)."

-- Victoria Kirst,
Software Engineer, Google