Introduction to Software Technology
Software Quality

Klaus Ostermann

Some slides on refactoring adapted from CS246 course at U Waterloo

Software Quality

- How can we maintain or improve the quality of software?

- What *is* software quality, anyway?
  - Correct implementation of requirements specification
  - Design quality, modularity
    - Extensibility, Maintainability, Understandability, Readability, Reusability …
    - Robustness to change
    - Low Coupling, High Cohesion
  - Reliability, Fault Tolerance, Testability, Performance, …

Software Quality

- How can we measure software quality?

> "You can't control what you can't measure"
> Tom DeMarco, 1986

- What about software metrics? What is a software metric?

> Measurement is the empirical, objective assignment of numbers, according to a rule derived from a model or theory, to attributes of objects or events with the intent of describing them.
>
> Kaner & Cem, "Software Engineer Metrics: What do they measure and how do we know?"

Examples of Software Metrics

- Lines of Code (LoC)
- Bugs per line of code
- Comment density
- Cyclomatic complexity
  - measures the number of linearly independent paths through a program's source code
- Halstead complexity measures
  - Derive software complexity from numbers of (distinct) operands and operators
- Program execution time
- Test Coverage
- Number of classes and interfaces
- Abstractness = ratio of abstract classes to total number of classes
- …

Metrics are rarely used

- Few companies establish measurement programs, even fewer succeed with them
- Those that use metrics often do so only to conform to criteria established in certain quality standards such as CMM
  - see N. E. Fenton, "Software Metrics: Successes, Failures & New Directions", 1999
- One *could* interpret this as evidence of the immaturity and unprofessionalism of the field
  - Aren't the engineers so successful because they can measure quality?
  - But this is again the misleading "software as engineering product" analogy that was already refuted in the first lecture of this course

Metrics are **rarely** ~~used~~ **useful**!

- Formally defined metrics are objective, but what do these measurements mean?
  - What can we conclude about quality if the cyclomatic complexity of our code is 12?
  - Answer: Nothing.
- Problem: Often unclear whether the metric correlates to any useful quality factor
  - Similar to the attempt of measuring the intelligence of a person in terms of the weight or circumference of the brain
- If a future potential employer tells you about their extensive software metrics suite, **run**! ☺

Tom DeMarco 23 years later…

"The book's most quoted line is its first sentence: "You can't control what you can't measure." This line contains a real truth, but I've become increasingly uncomfortable with my use of it. Implicit in the quote is that control is an important aspect, maybe the most important, of any software project. **But it isn't.** Many projects have proceeded without much control but managed to produce wonderful products."

Tom DeMarco, "Software Engineering: An Idea Whose Time Has Come and Gone", 2009.

Software Quality

- If metrics don't work, how do we assess the quality of software?
- Answer: By a case-by-case analysis of each individual software project
  - Reason about design quality, extensibility, …
  - Reason by comparing to designs that have proven useful
    - Design patterns etc.
  - Reason by looking for "code smells" and anti-patterns
  - By extensive test suites
  - By using analysis tools: Static analysis, dynamic analysis, formal verification

Code Smells

- Code smell: Any symptom in the code of a program that **possibly** indicates a deeper problem
  - Term popularized by Kent Beck in his “Refactoring” book
- Common code smells
  - Duplicated code
  - Long method, Large class
  - Feature envy, inappropriate intimacy
  - Contrived complexity

Anti-Pattern

- An anti-pattern is a pattern that may be commonly used but is ineffective and/or counterproductive in practice.
- A description of anti-patterns is useful
  - One can recognize the forces that lead to their repetition and learn how others have refactored themselves out of these broken patterns.
- Examples:
  - Action at a distance: Unexpected interaction between otherwise separated parts of a system
  - Sequential coupling: A class that requires its methods to be called in a particular order
  - Circular dependency: Unnecessary direct or indirect mutual dependencies between software modules

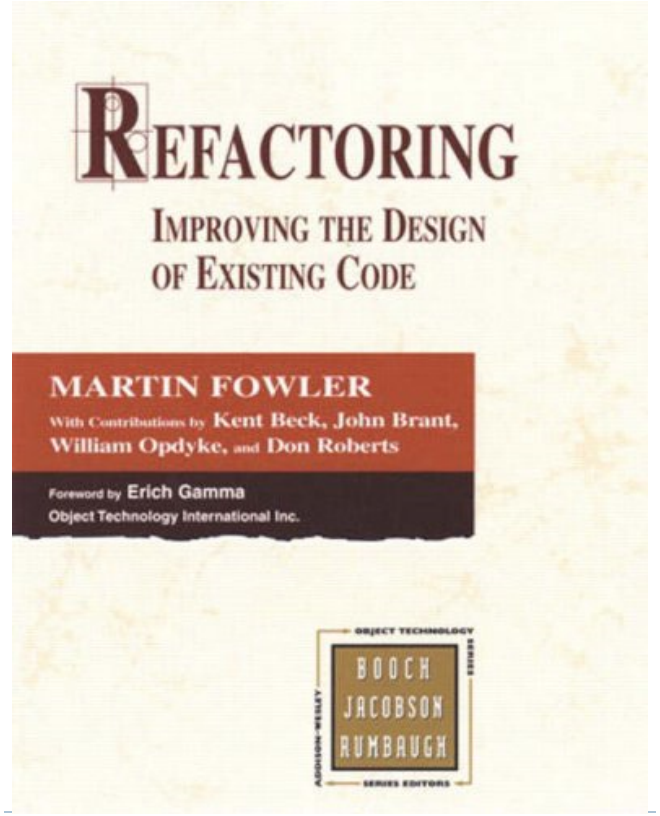
Anti-Pattern

- More Examples:
  - Abstraction Inversion: Re-implement low-level functions using high-level functions
  - Interface bloat: Making an interface so powerful that it is too hard to implement
  - Busy spin or busy waiting: Consuming CPU while waiting for something to happen

- See http://c2.com/cgi/wiki?AntiPatternsCatalog for an extensive overview over common antipatterns

What to do about anti-patterns and code smells?

- Refactorings formalize the idea to systematically remove anti-patterns and code smells
  - Often formalized to a degree that it can be automated in the form of an IDE tool
- Standard reference: →
- Refactorings can often be understood to improve the modularity of the code
- Refactorings do not change the behavior of code, e.g., add a feature
- Let's look at some smells and associated refactorings in more detail!



Bad smells and associated refactorings

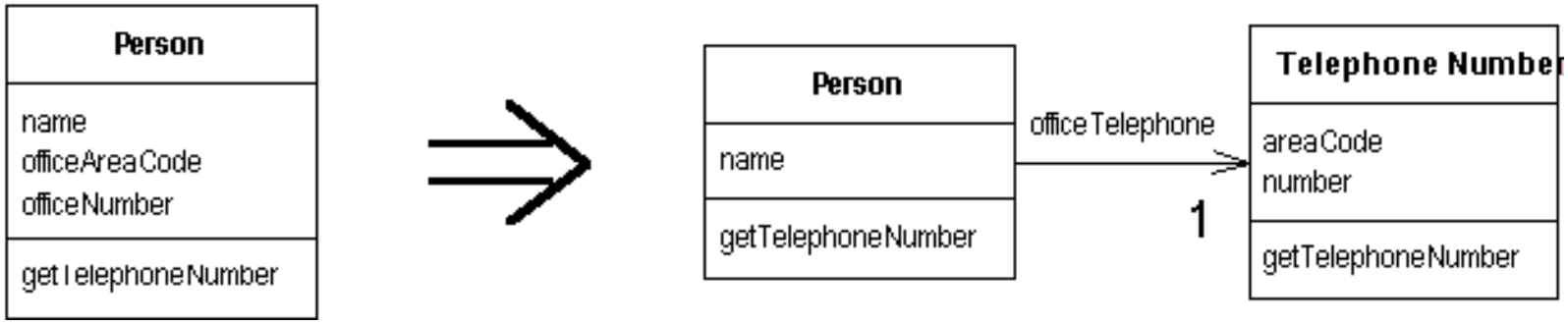
- *Duplicated code* – "The #1 bad smell"
- We have already discussed how to abstract over different forms of duplicated code in the lecture on reuse
- Same expression in two methods in the same class?
  - Make it a `private` auxiliary routine and parameterize it
    (Extract method refactoring)
- Same code in two related classes?
  - Push commonalities into closest mutual ancestor and parameterize
  - Use *template method* DP for variation in subtasks
    (Form template method refactoring)

Bad smells and associated refactorings

- *Duplicated code*
  - Same code in two *unrelated* classes?
    - Ought they be related?
      - Introduce abstract parent *(Extract class, Pull up method)*
    - Does the code really belong to just one class?
      - Make the other class into a client *(Extract method)*
    - Can you separate out the commonalities into a subpart or other function object?
      - Make the method into a *subobject* of both classes.
      - *Strategy* DP allows for polymorphic variation of methods-as-objects

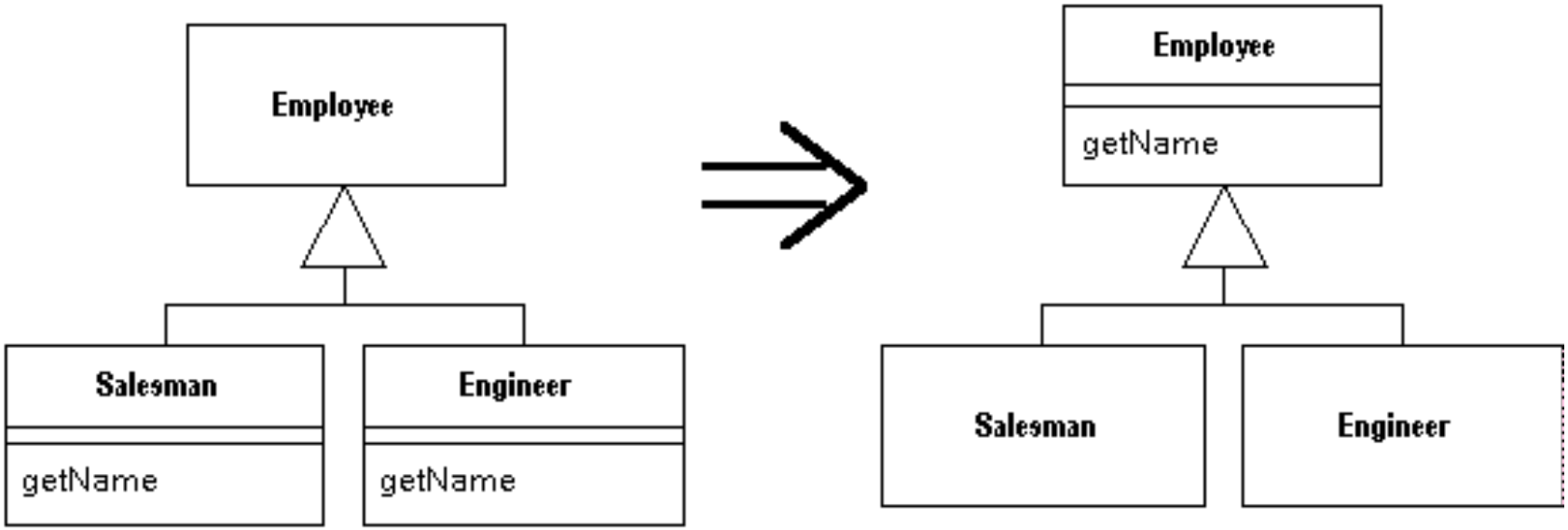
      (Replace method with method object) = apply strategy pattern

Extract Class Refactoring



You have one class doing work that should be done by two.
Create a new class and move the relevant fields and methods from the old class into the new class.

Pull-Up Method Refactoring



You have methods with identical results on subclasses.
Move them to the superclass

Extract Method Refactoring

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

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

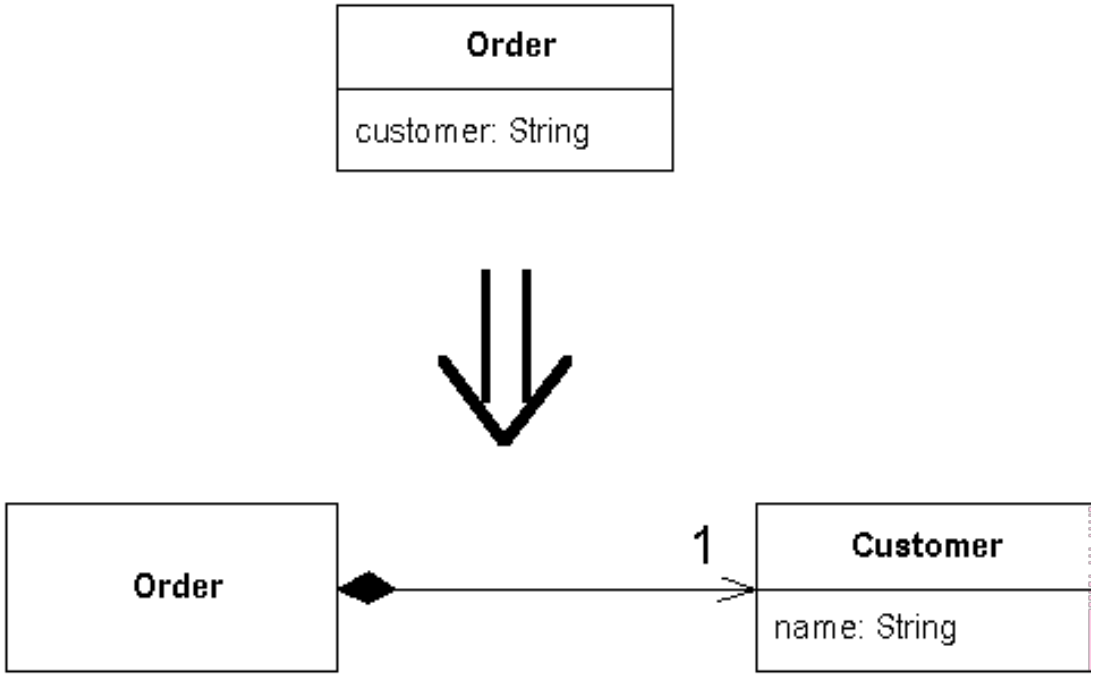
You have a code fragment that can be grouped together.

Turn the fragment into a method whose name explains the purpose of the method.

Bad smells in code

- ***Long method***
  - Often a sign of:
    - Trying to do too many things
    - Poorly thought out abstractions and boundaries
  - Best to think carefully about the major tasks and how they inter-relate. Be aggressive!
    - Break up into smaller `private` methods within the class
      (Extract method)
    - Delegate subtasks to subobjects that "know best" (*i.e.,* template method DP)
      (Extract class/method, Replace data value with object)

Replace Data Value with Object Refactoring



You have a data item that needs additional data or behavior.
Turn the data item into an object.

Bad smells in code

- ***Long method***
  - Fowler's heuristic:
    - *When you see a comment, make a method.*
    - Often, a comment indicates:
      - The next major step
      - Something non-obvious whose details detract from the clarity of the routine as a whole.
    - In either case, this is a good spot to "break it up".

Bad smells in code

- **Large class**
  - *i.e.,* too many different subparts and methods
  - Two step solution:
    1. Gather up the little pieces into aggregate subparts.
       (Extract class, replace data value with object)
    2. Delegate methods to the new subparts.
       (Extract method)
  - Likely, you'll notice some unnecessary subparts that have been hiding in the forest!
  - Resist the urge to micromanage the subparts!

Bad smells in code

- **Large class**
  - Counter example:
    - Library classes often have large, fat interfaces (many methods, many parameters, lots of overloading)
      - If the many methods exist *for the purpose of flexibility*, that's OK in a library class.

Bad smells in code

- ***Long parameter list***
  - Long parameter lists make methods difficult for clients to understand
  - This is often a symptom of
    - Trying to do too much
    - … too far from home
    - … with too many disparate subparts

"if your procedure has more than about half a dozen *parameters*, you *probably forgot a few*." – Alan Perlis

Bad smells in code

- ***Long parameter list***
  - In the old days, structured programming taught the use of parameterization as a cure for global variables.
    - With modules/OOP, objects have mini-islands of state that can be reasonably treated as "global" to the methods (yet are still hidden from the rest of the program).
    - *i.e.,* You don't need to pass a subpart of yourself as a parameter to one of your own methods.

Bad smells in code

- ***Long parameter list***
  - Solution:
    - Trying to do too much?
      - Break up into sub-tasks

        (Extract method)
    - … too far from home?
      - Localize passing of parameters; don't just pass down several layers of calls

        (Preserve whole object, introduce parameter object)
    - … with too many disparate subparts?
      - Gather up parameters into aggregate subparts
      - Your method interfaces will be much easier to understand!

        (Preserve whole object, introduce parameter object)

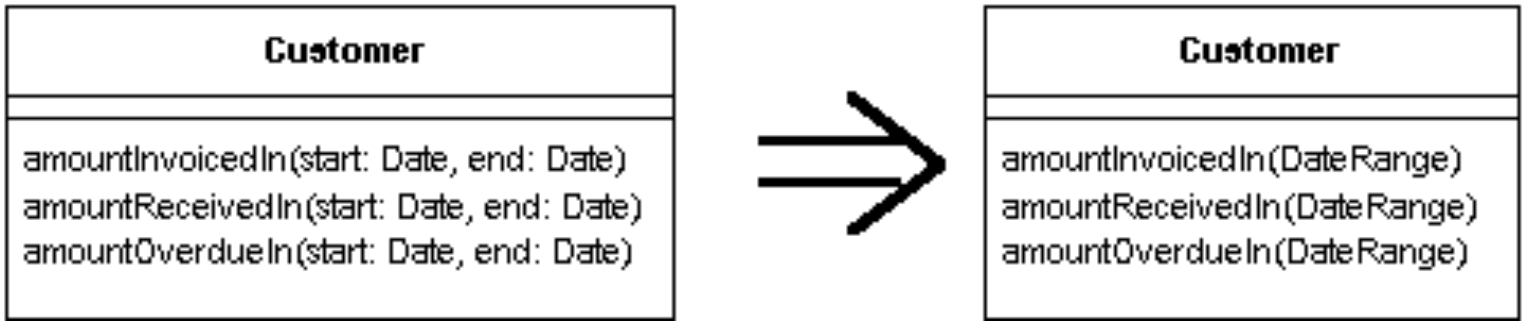
Preserve Whole Object Refactoring

```
int low = daysTempRange().getLow();
int high = daysTempRange().getHigh();
withinPlan = plan.withinRange(low, high);
```

```
withinPlan = plan.withinRange(daysTempRange());
```

You are getting several values from an object and passing these values as parameters in a method call.
Send the whole object instead.

Introduce Parameter Object Refactoring



You have a group of parameters that naturally go together.
Replace them with an object.

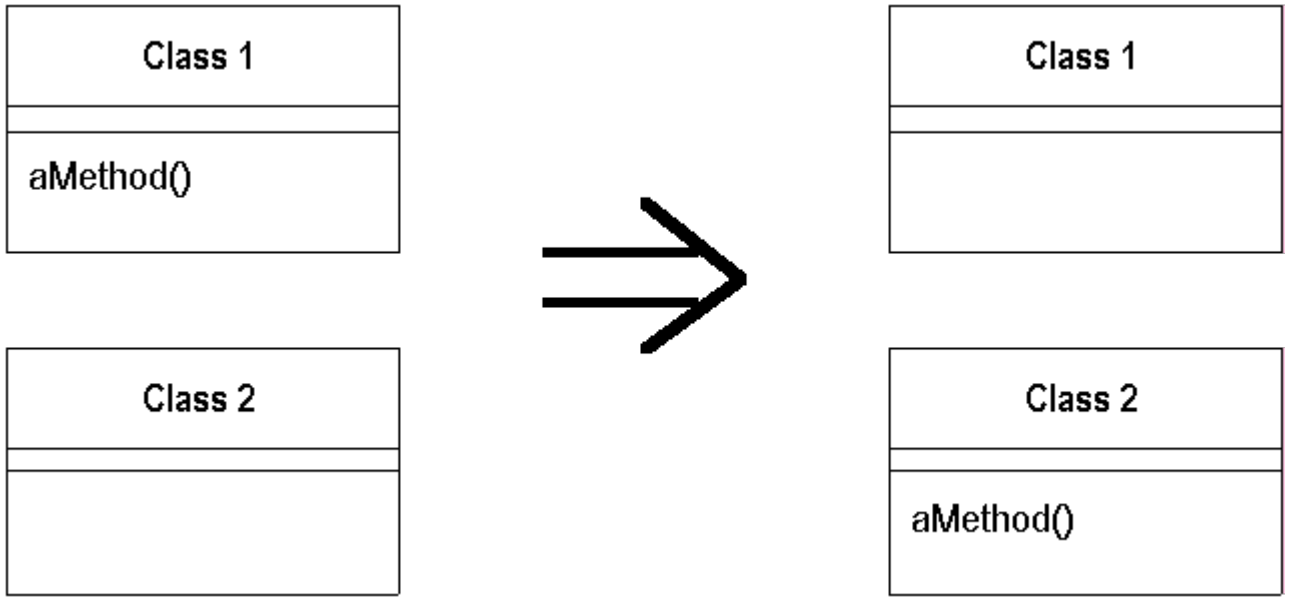
Bad smells in code

- **Divergent change**
  - Occurs when one class is commonly changed in different ways for different reasons
  - Likely, this class is trying to do too much and contains too many unrelated subparts
  - Over time, some classes develop a "God complex"
    - They acquires details/ownership of subparts that rightly belong elsewhere
  - This is a sign of *poor cohesion*
    - Unrelated elements in the same container
  - Solution:
    - Break it up, reshuffle, reconsider relationships and responsibilities *(Extract class)*

Bad smells in code

- **Shotgun surgery**
  - … the opposite of divergent change
    - Each time you want to make a single, seemingly coherent change, you have to change lots of classes in little ways
  - Also a classic sign of poor cohesion
    - Related elements are *not* in the same container!
  - Solution:
    - Look to do some gathering, either in a new or existing class. *(Move method/field)*

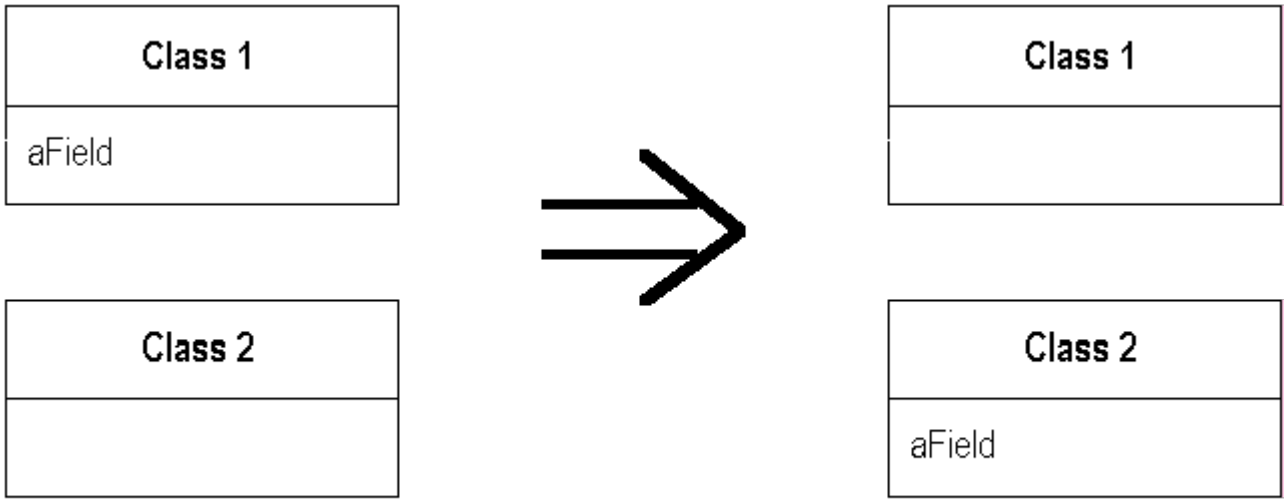
Move Method Refactoring



A method is, or will be, using or used by more features of another class than the class on which it is defined.

Create a new method with a similar body in the class it uses most.
Either turn the old method into a simple delegation, or remove it altogether

Move Field Refactoring



A field is, or will be, used by another class more than the class on which it is defined.
Create a new field in the target class, and change all its users.

Bad smells in code

- **Feature envy**
  - A method seems more interested in another class than the one it's defined in

    e.g., a method `A.m()` calls lots of get/set methods of class `B`
  - Solution:
    - Move `m()` (or part of it) into `B`!
      (Move method/field, extract method)
  - Exceptions:
    - *Visitor/iterator/strategy* DP where the whole point is to decouple the data from the algorithm
      - Feature envy is more of an issue when both `A` and `B` have interesting data

Bad smells in code

- **Data clumps**
  - You see a set of variables that seem to "hang out" together

    e.g., passed as parameters, changed/accessed at the same time
  - Usually, this means that there's a coherent subobject just waiting to be recognized and encapsulated

```
void Scene::setTitle (string titleText,
        int titleX, int titleY,
        Colour titleColour){…}

void Scene::getTitle (string& titleText,
        int& titleX, int& titleY,
        Colour& titleColour){…}
```

Bad smells in code

- **Data clumps**
  - In the example, a `Title` class is waiting to be born
  - If a client knows how to change a title's `x`, `y`, `text`, and `colour`, then it knows enough to be able to "roll its own" `Title` objects.
    - However, this does mean that the client now has to talk to another class.
  - This will greatly shorten and simplify your parameter lists (which aids understanding) and makes your class conceptually simpler too.
  - Moving the data may create **feature envy** initially
    - May have to iterate on the design until it feels right.

  (Preserve whole object, extract class, introduce parameter object)

Bad smells in code

- **Primitive obsession**
  - All subparts of an object are instances of primitive types (`int`, `string`, `bool`, `double`, *etc.*)

    e.g., dates, currency, SIN, tel.#, ISBN, special string values
  - Often, these small objects have interesting and non-trivial constraints that can be modelled

    e.g., fixed number of digits/chars, check digits, special values
  - Solution:
    - Create some "small classes" that can encapsulate coherent subsets of the primitive data and validate and enforce the constraints.

    (Replace data value with object, extract class, introduce parameter object)

Bad smells in code

- **Switch statements**
  - We saw this before; here's Fowler's example:

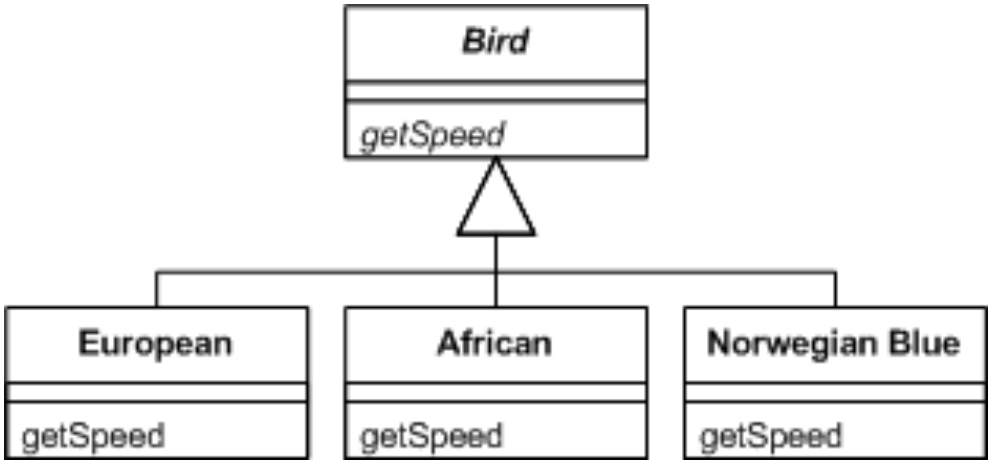
```
Double getSpeed () {
    switch (_type) {
        case EUROPEAN:
            return getBaseSpeed();
        case AFRICAN:
            return getBaseSpeed() -
                getLoadFactor() * _numCoconuts;
        case NORWEGIAN_BLUE:
            return (_isNailed) ? 0
                : getBaseSpeed(_voltage);
    }
}
```

Bad smells in code

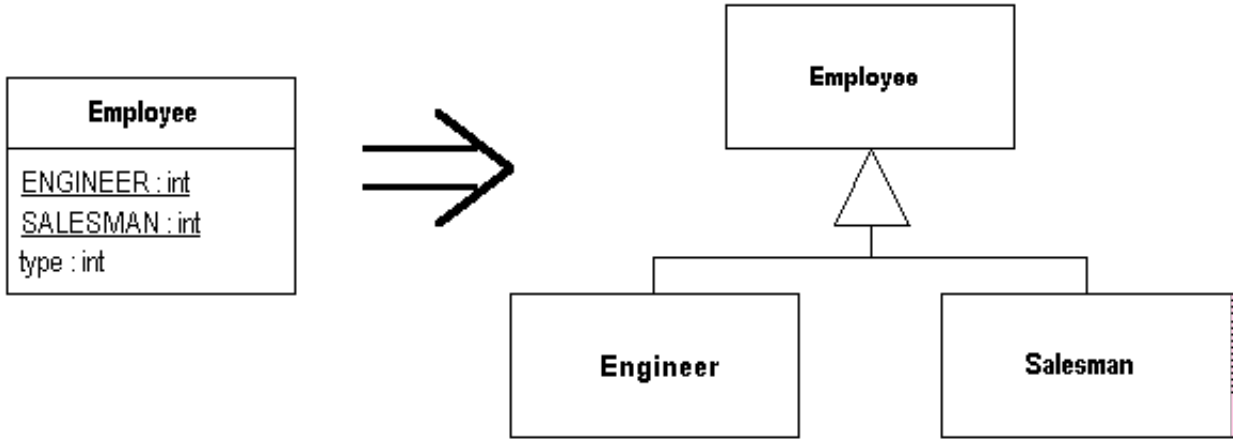
- **Switch statements**
  - This is an example of a lack of understanding polymorphism and a lack of encapsulation.
  - Solution:
    - Redesign as a polymorphic method of `PythonBird`

  (Replace conditional with polymorphism, replace type code with subclasses)

Replace conditional with polymorphism, replace type code with subclasses



Replace conditional with polymorphism



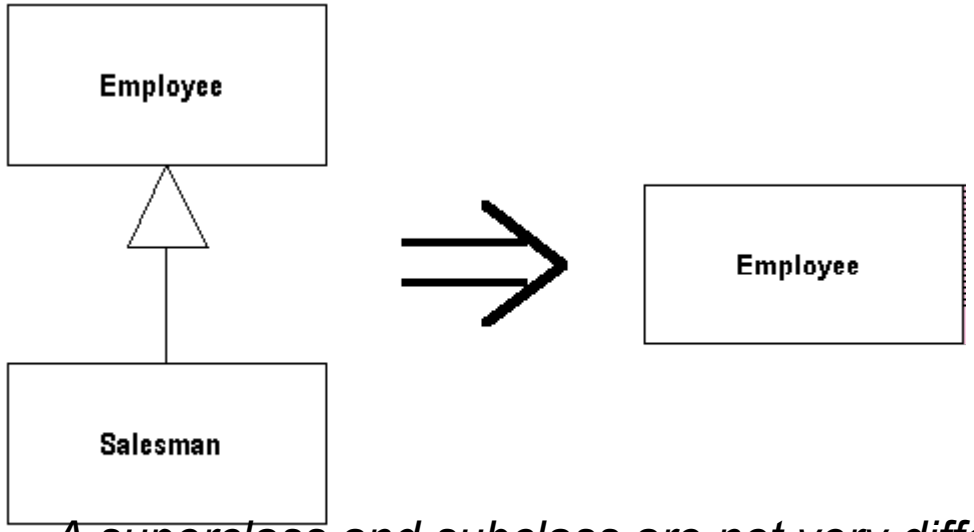
replace type code with subclasses

Bad smells in code

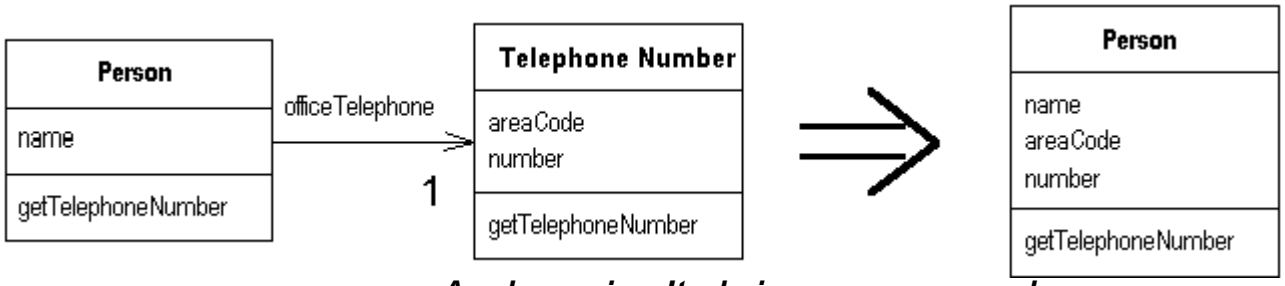
- ***Lazy class***
  - Classes that doesn't do much that's different from other classes.
  - If there are several sibling classes that don't exhibit polymorphic behavioural differences , then consider just collapsing them back into the parent and add some parameters
  - Often, ***lazy classes*** are legacies of ambitious design or a refactoring that gutted the class of interesting behaviour

    (Collapse hierarchy, inline class)

Collapse hierarchy, inline class



A superclass and subclass are not very different.
Merge them together.



A class isn't doing very much.
Move all its features into another class and delete it.

Bad smells in code

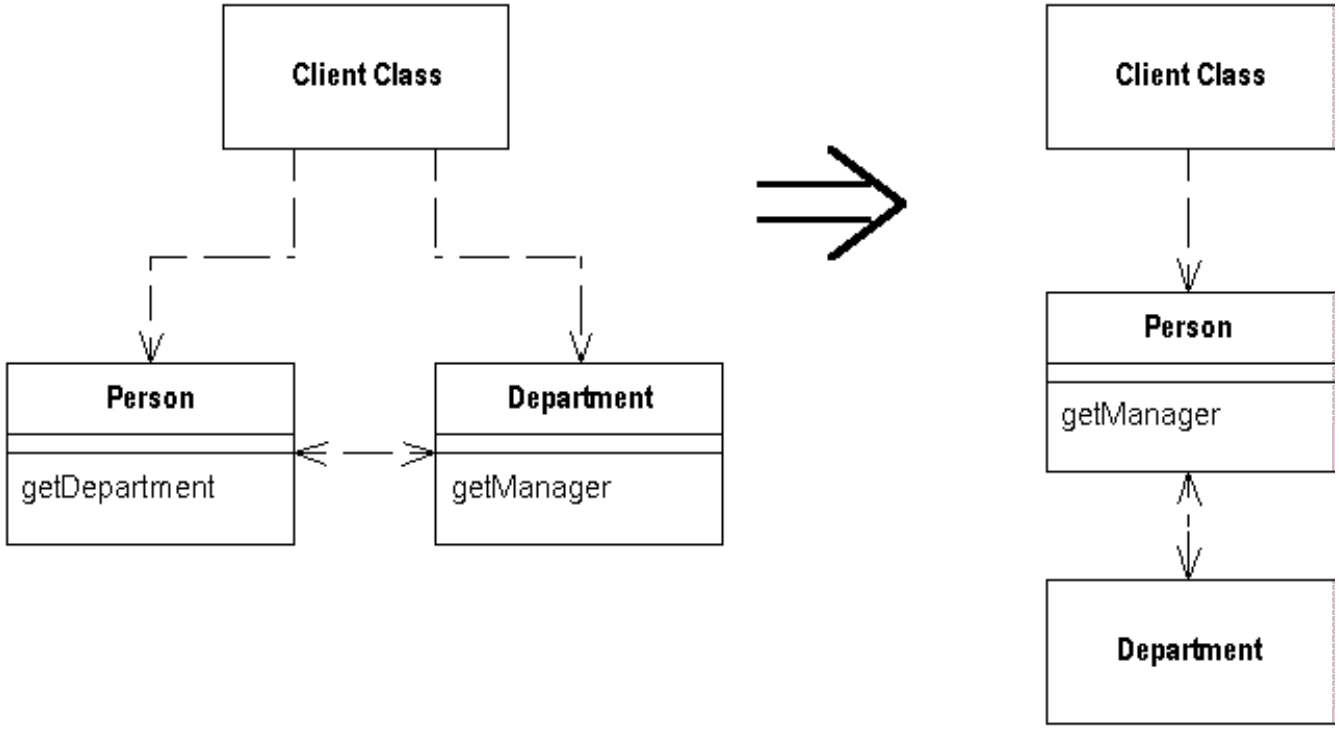
- **Speculative generality**
  - *"We might need this one day …"*
    - Fair enough, but did you really need it after all?
    - Extra classes and features add to complexity.
  - "Extreme Programming" philosophy:
    - "As simple as possible but no simpler."
    - "Rule of three".
  - Keep in mind that refactoring is an ongoing process.
    - If you really do need it later, you can add it back in.

  (Collapse hierarchy, inline class, remove parameter)

Bad smells in code

- ***Message chains***
  - A client asks one object for another object, which the client then asks for yet another object, which the client then asks for yet another another object,
  - Navigating this way means the client is coupled to the structure of the navigation.
  - Any change to the intermediate relationships causes the client to have to change (cf. **Law of Demeter**)
  - Solution: *Hide delegate*

Hide delegate refactoring



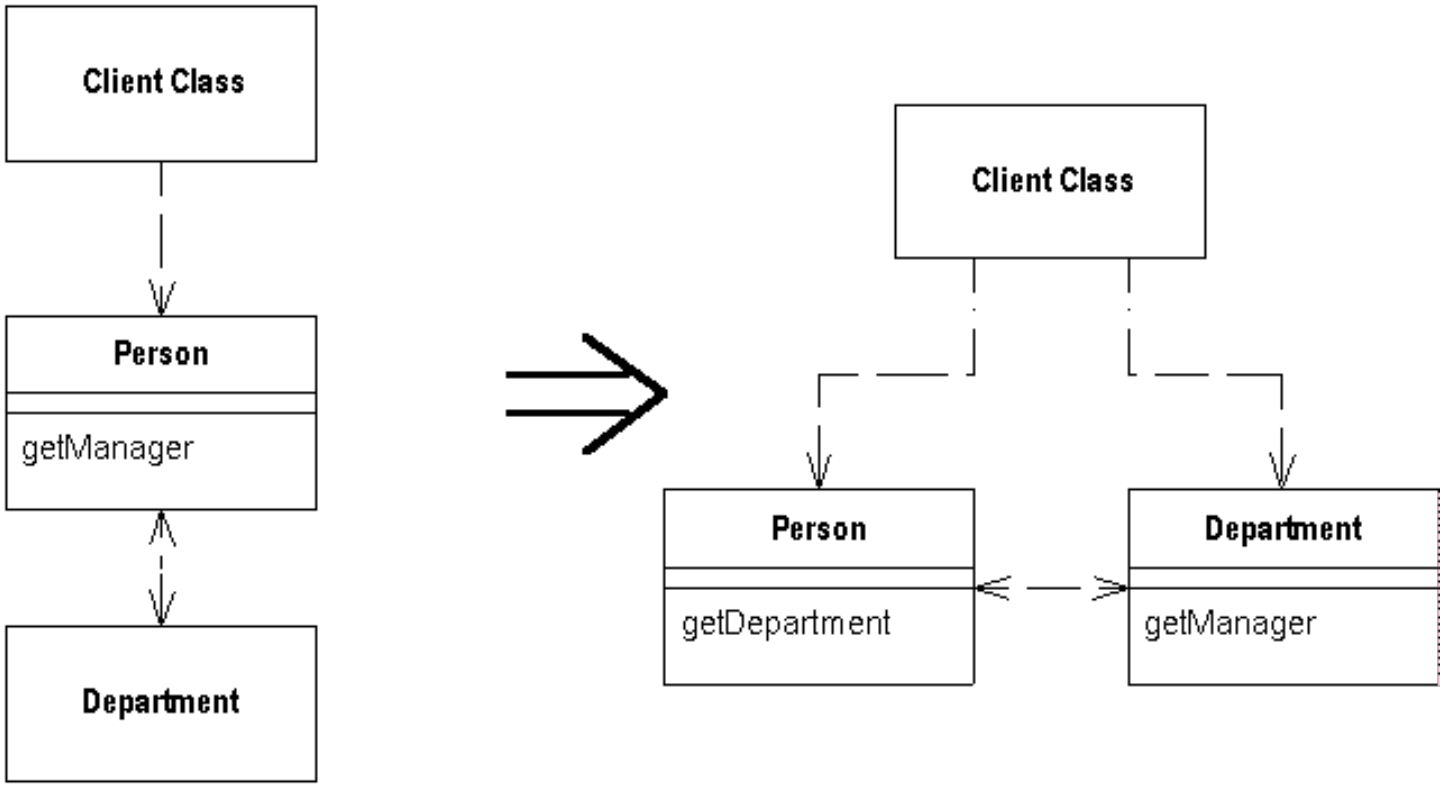
A client is calling a delegate class of an object.
Create methods on the server to hide the delegate.

Bad smells in code

- **Middle man**
  - *"All hard problems in software engineering can be solved by an extra level of indirection."*
    - OODPs pretty well all boil down to this, albeit in quite clever and elegant ways.
  - If you notice that many of a class's methods just turn around and beg services of delegate subobjects, the basic abstraction is probably poorly thought out.
  - An object should be more than the sum of its parts in terms of behaviours!

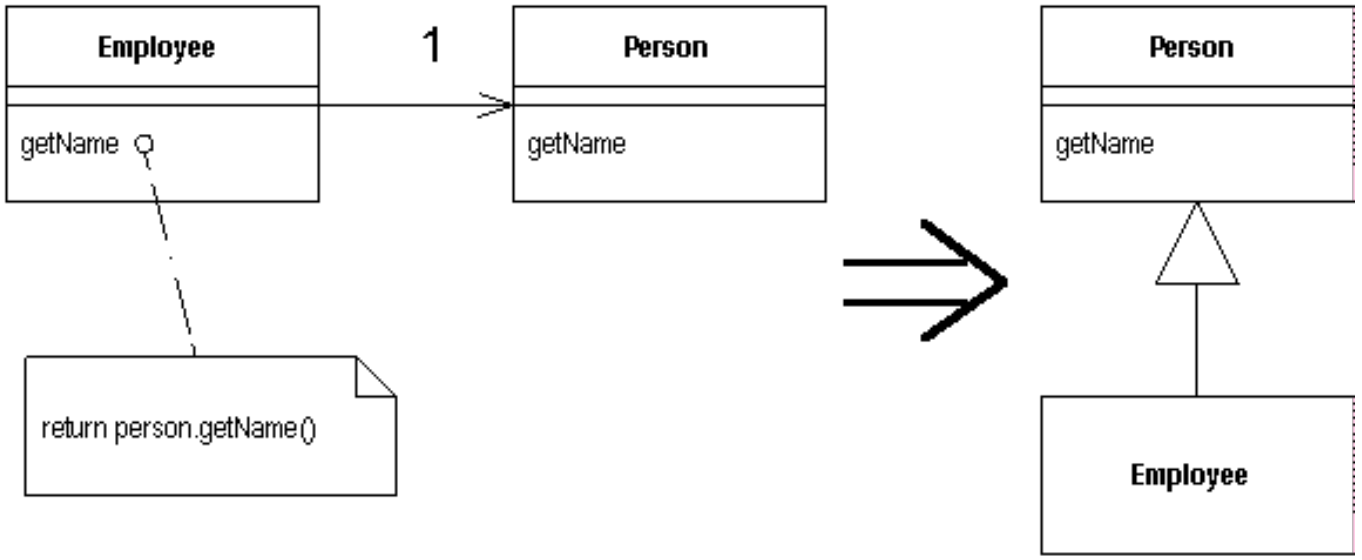
  (Remove middle man, replace delegation with inheritance)

Remove Middle Man Refactoring



A class is doing too much simple delegation.
Get the client to call the delegate directly

Replace Delegation with Inheritance



You're using delegation and are often writing many simple delegations for the entire interface.

Make the delegating class a subclass of the delegate

Bad smells in code

- **Inappropriate intimacy**
  - Sharing of secrets between classes, esp. outside of the holy bounds of inheritance

    e.g., `public` variables, indiscriminate definitions of get/set methods, C++ friendship, `protected` data in classes
  - Leads to data coupling, intimate knowledge of internal structures and implementation decisions.
    - Makes clients brittle, hard to evolve, easy to break.
  - Solution:
    - *Appropriate* use of get/set methods
    - Rethink basic abstraction.
    - Merge classes if you discover "true love"

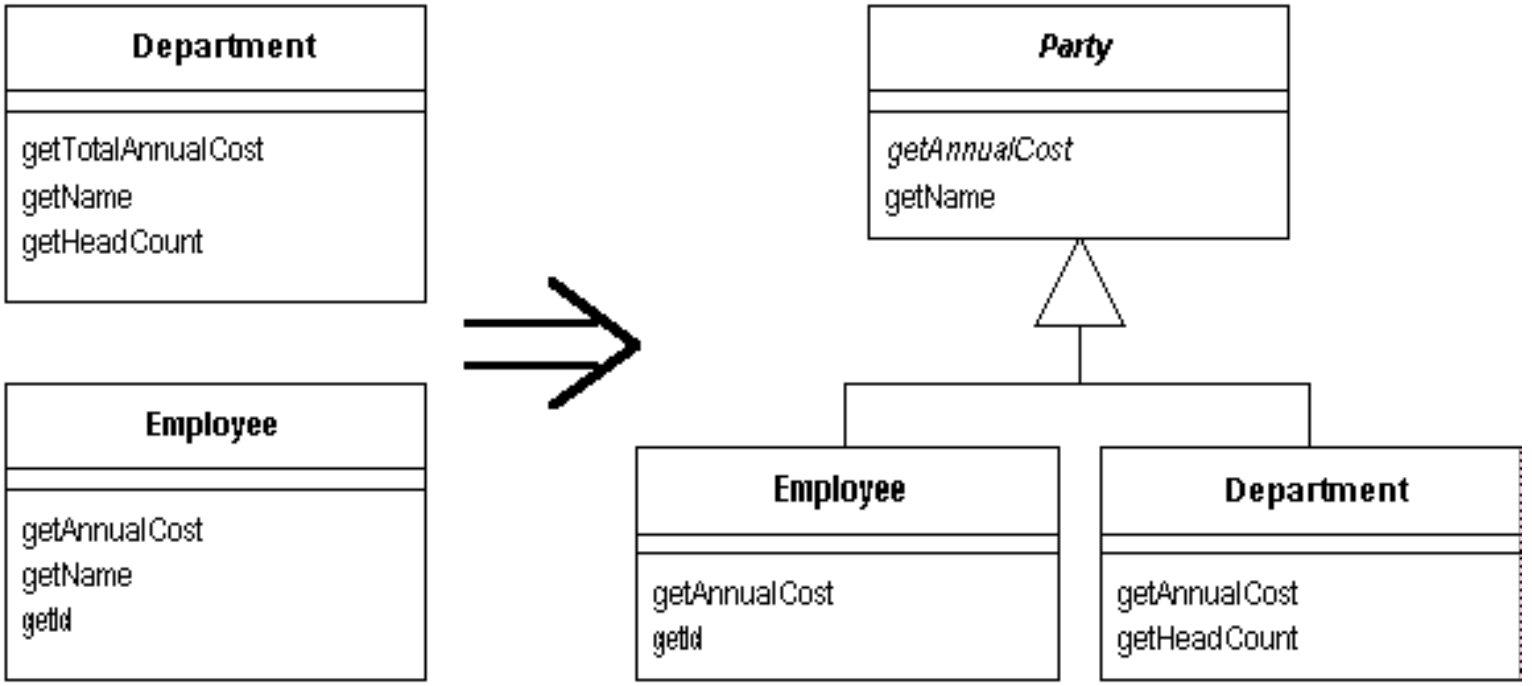
    (Move/extract method/field, change bidirectional association to unidirectional, hide delegate)

Bad smells in code

- **Alternative classes with different interfaces**
  - Classes/methods seem to implement the same or similar abstraction yet are otherwise unrelated.
  - Solution:
    - Move the classes "closer" together.
      - Find a common interface, perhaps an ABC.
      - Find a common subpart and remove it.

    (Extract [super]class, move method/field, rename method)

Extract Superclass Refactoring



You have two classes with similar features.

Create a superclass and move the common features to the superclass.

Bad smells in code

- **Data class**
  - Class consists of (simple) data fields and simple accessor/mutator methods only.
  - Solution:
    - Have a look at usage patterns in the clients
    - Try to abstract some commonalities of usage into methods of the data class and move some functionality over
  - *"Data classes are like children. They are OK as a starting point, but to participate as a grownup object, they need to take on some responsibility."*

    (Extract/move method)

Bad smells in code

- **Comments**
  - XP philosophy discourages comments, in general:
    - Instead, make methods short and use long identifiers
  - In the context of refactoring, Fowler claims that long comments are often a sign of opaque, complicated, inscrutable code.
    - They aren't against comments so much as in favour of self-evident coding practices.
    - Rather than explaining opaque code, restructure it!

    (Extract method/class, [many others applicable] …)
  - Comments are best used to document rationale

    i.e., explain *why* you picked one approach over another.

Summary

- Fowler *et al.*'s *Refactoring* is a well-written book that summarizes a lot of "best practices" of OOD/OOP with nice examples.
  - Many books on OOD heuristics are vague and lack concrete advice.
  - Most of the advice in this book is aimed at low-level OO programming.
    - *i.e.,* loops, variables, method calls, and class definitions.
  - Next obvious step up in abstraction/scale is to OODPs.
    - *i.e.,* collaborating classes
- This is an excellent book for the intermediate-level OO programmer.
  - Experienced OO programmers will have discovered a lot of the techniques already on their own.
- Many sources about refactoring on the web:
  - http://refactoring.com/

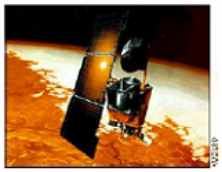
Testing

Some slides by T. Ball and J. Aldrich

Why test?

Mars Climate Orbiter

- Purpose: to relay signals from the Mars Polar Lander once it reached the surface of the planet
- Disaster: smashed into the planet instead of reaching a safe orbit
- Why: Software bug - failure to convert English measures to metric values
- $165M

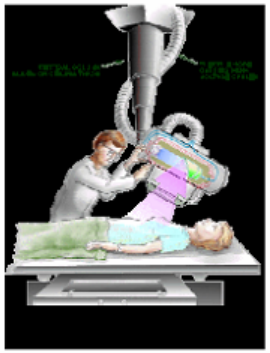


Shooting Down of Airbus 320

- 1988
- US Vicennes shot down Airbus 320
- Mistook airbus 320 for a F-14
- 290 people dead
- Why: Software bug - cryptic and misleading output displayed by the tracking software

THERAC-25 Radiation Therapy

- THERAC-25, a computer-controlled radiation-therapy machine
- 1986: two cancer patients at the East Texas Cancer Center in Tyler received fatal radiation overdoses
- Why: Software bug - mishandled race condition (i.e., miscoordination between concurrent tasks)



Testing: Challenges

- Testing is costly
- Test effectiveness and software quality hard to measure
- Incomplete, informal and changing specifications
- Downstream cost of bugs is enormous
- Lack of spec and implementation testing tools
- Integration testing across product groups
- Patching nightmare
- Versions exploding

Example: Testing MS Word

- inputs
  - keyboard
  - mouse/pen
  - .doc, .htm, .xml, …
- outputs (WYSIWYG)
  - Printers
  - displays
  - doc, .htm, .xml, …
- variables
  - fonts
  - templates
  - languages
  - dictionaries
  - styles
- Interoperability
  - Access
  - Excel
  - COM
  - VB
  - sharepoint
- Other features
  - 34 toolbars
  - 100s of commands
  - ? dialogs
- Constraints
  - huge user base

From Microsoft Office EULA…

11. EXCLUSION OF INCIDENTAL, CONSEQUENTIAL AND CERTAIN OTHER DAMAGES. TO THE MAXIMUM EXTENT PERMITTED BY APPLICABLE LAW, **IN NO EVENT SHALL MICROSOFT OR ITS SUPPLIERS BE LIABLE FOR ANY SPECIAL, INCIDENTAL, INDIRECT, OR CONSEQUENTIAL DAMAGES WHATSOEVER** (INCLUDING, BUT NOT LIMITED TO, DAMAGES FOR LOSS OF PROFITS OR CONFIDENTIAL OR OTHER INFORMATION, FOR BUSINESS INTERRUPTION, FOR PERSONAL INJURY, FOR LOSS OF PRIVACY, FOR FAILURE TO MEET ANY DUTY INCLUDING OF GOOD FAITH OR OF REASONABLE CARE, FOR NEGLIGENCE, AND FOR ANY OTHER PECUNIARY OR OTHER LOSS WHATSOEVER) **ARISING OUT OF OR IN ANY WAY RELATED TO THE USE OF OR INABILITY TO USE THE SOFTWARE PRODUCT,** THE PROVISION OF OR FAILURE TO PROVIDE SUPPORT SERVICES, OR OTHERWISE UNDER OR IN CONNECTION WITH ANY PROVISION OF THIS EULA, EVEN IN THE EVENT OF THE FAULT, TORT (INCLUDING NEGLIGENCE), STRICT LIABILITY, BREACH OF CONTRACT OR BREACH OF WARRANTY OF MICROSOFT OR ANY SUPPLIER, AND EVEN IF MICROSOFT OR ANY SUPPLIER HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

From GPL

- **11.** BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. **THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU**. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.
- **12. IN NO EVENT** UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING **WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM** AS PERMITTED ABOVE, **BE LIABLE TO YOU FOR DAMAGES**, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES **ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM** (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

The goals of testing

- Not-quite-right answers
  - Make sure it doesn't crash
  - Regression testing –no new bugs
  - Make sure you meet the spec
  - Make sure you don't have harmful side effects
- Actual goals
  - Reveal faults
  - Establish confidence
  - Clarify or represent the specification

THE limitation of testing

> Testing can only show the presence of errors, not their absence
>
> - E.W. Dijkstra

… to be continued in the next lecture!