Topic 3

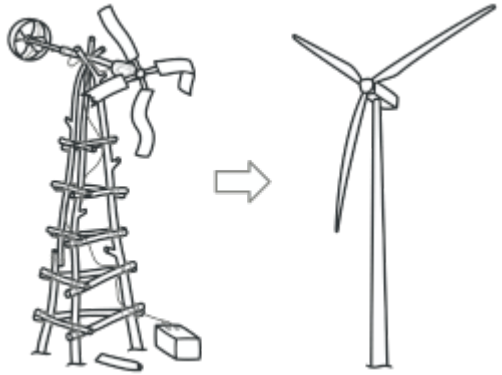
Any fool can write code that a computer can understand. Good programmers write code that humans can understand. [M. Fowler]

Refactoring

References:
Refactoring: Improving the Design of Existing Code by Martin Fowler; Kent Beck (1st Edition Addison Wesley, 1999, 2nd Edition Addison Wesley, 2018)

What is Refactoring?

- Refactoring is the process of changing a software system in such a way that it does not alter external behaviour of the code yet improves its internal structure.
- It is a disciplined way to clean up the code, improving the structural “-ilities” (understandability etc.) while minimizing the chance of new bugs.
- Based on the understanding that good software does not just mean “does it work”.
- Good development involves designing and coding – a good design implemented correctly in code.
  - e.g. as per SOLID and GRASP which we have covered so far
- However changes made over time can cause loss of structure or distort the original design – causing the code to “smell” (Martin Fowler)
- Refactoring does the opposite – reworks a bad design into a good one.



What does Refactoring Involve?

- Refactoring is made in a number of simple steps (e.g.):
  - Add new attributes/methods/classes/interfaces
  - Extract or merge methods/classes/interfaces
  - Move or rename attributes/methods/classes/interfaces
- Refactoring ensures that design is not restricted/constrained by the initial stages
  - occurs continuously during development.

When Should I Refactor?

- Allocate two weeks for refactoring every couple of months?
  - Refactoring should be something that is done all the time in small steps

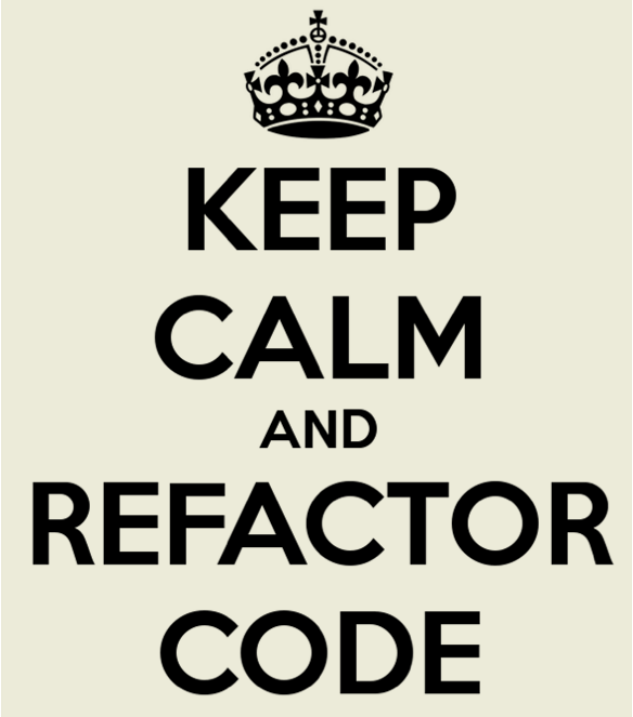
- Refactor when you add new functionality
  - Refactoring existing code can make it easier to add a new feature

- Refactor when you need to fix a bug (especially when debugging has become difficult)
  - Refactoring when fixing bugs makes the code more understandable/traceable
  - There is a school of though that says you should only refactor working code!

- Refactor when you do a code review
  - Many organizations do a regular code review
  - They help to transfer knowledge to less experienced staff
  - The code may look clear to the author but not others
  - Reviews allow suggestions from many people
  - Code review is taken to its limit with extreme programming practice of pair programming – continuous code review takes place

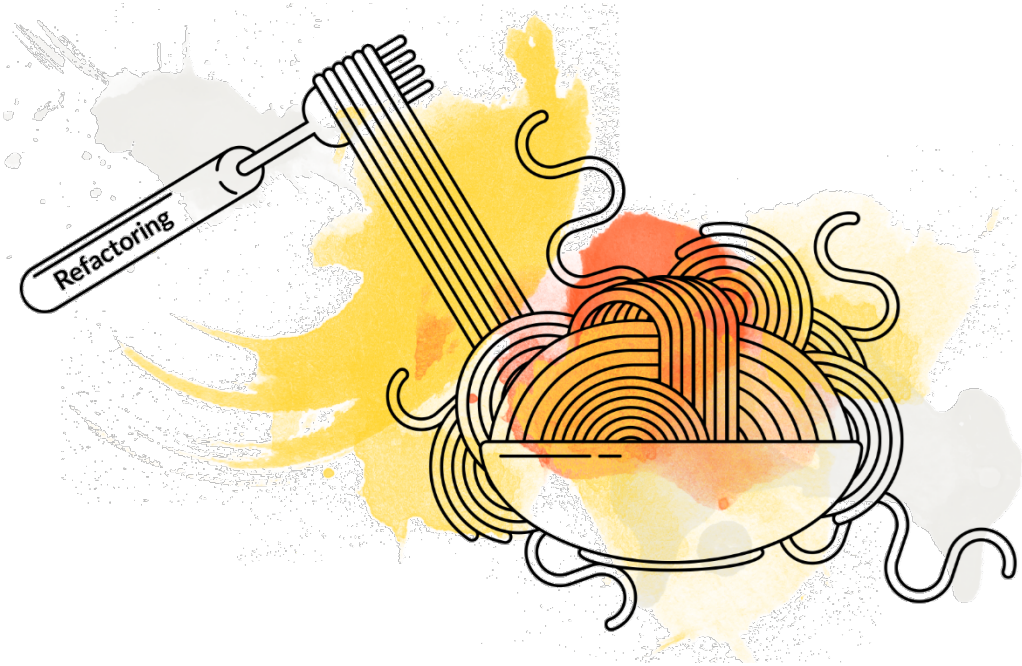
Challenges with Refactoring

- If a business application/information system is tightly coupled to the database schema it may be difficult to change
  - Minimize the impact by placing an additional layer between business objects and the database model (**Indirection** in GRASP).
- Changing Interfaces affects the existing client code – to which you may not have access
  - If you are changing the published interface you must retain the old and new interfaces, at least for a time period
  - new interface can *extend* old interface
  - **OR** the old interface calls though to the new interface (i.e., the old method calls the new one) so that there is only a single implementation
- Major design changes (due to discovered mistakes) may be difficult to refactor
  - When considering design alternatives one may consider the amount of effort needed in refactoring, if a design change is necessary (later).
- Requires a good (ideally automated) testing routine since refactoring can introduce bugs (regressions)



When Shouldn't you Refactor?

- When existing code is such a mess it is easier to rewrite.
  - Remember it is best if code works correctly before you commence refactoring
  - Otherwise you are debugging which is a different activity
- Avoid refactoring when you are close to a project deadline
  - may be harder than expected
  - could cause missed deadline
  - the benefits may take some time to realise



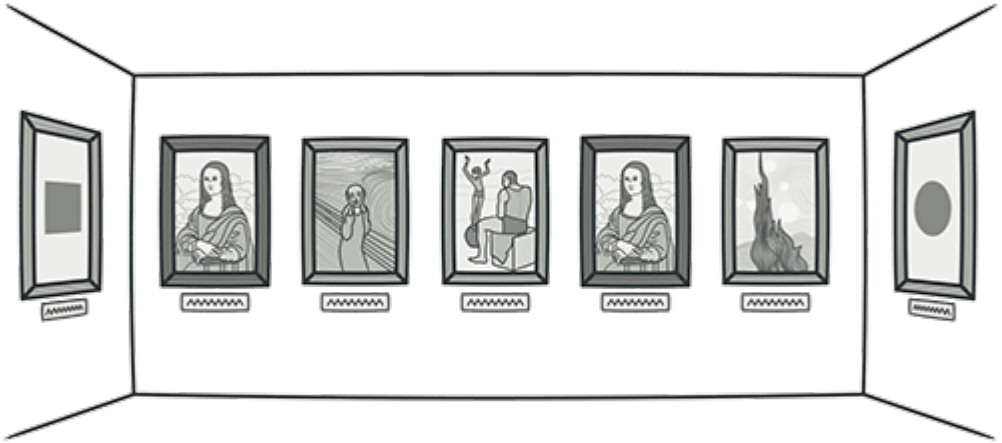
When to Refactor? Bad Smells

- Martin Fowler and Kent Beck coined the term *Bad Smells* or “code smells” for code that requires refactoring
- In general there are no hard and fast rules for:
  - Maximum lines of code in methods
  - Maximum number of instance variables in a class
- Human intuition is the key; however the bad smells can give an indication
- List of Bad Smells are presented next.

Code Duplication

https://refactoring.guru/smells/duplicate-code

- This is usually the most obvious/odious “smell”
- Less code written means less code to maintain
- How to address?
  - *Extract Method/Class*
  - Inheritance or Delegation/Composition
  - Template method (which allows steps to be deferred)

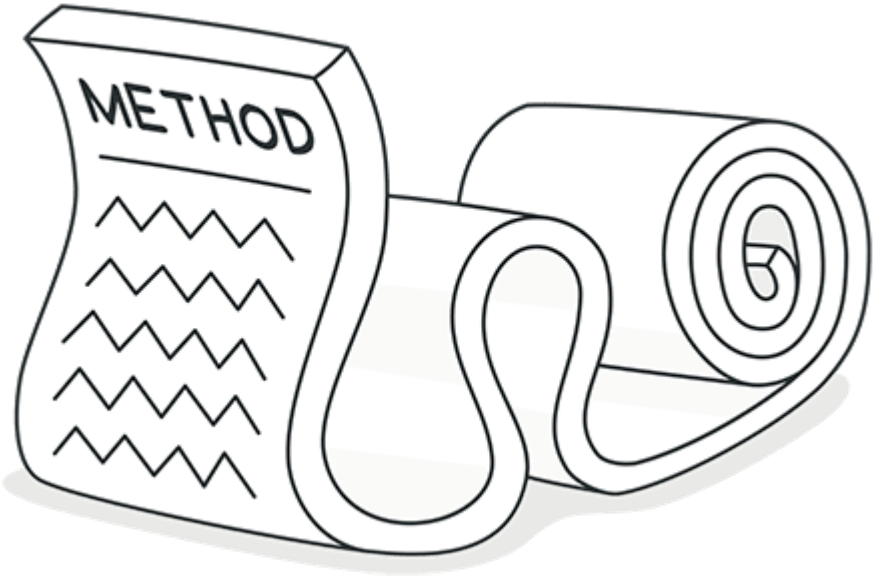


Long Methods

https://refactoring.guru/smells/long-method

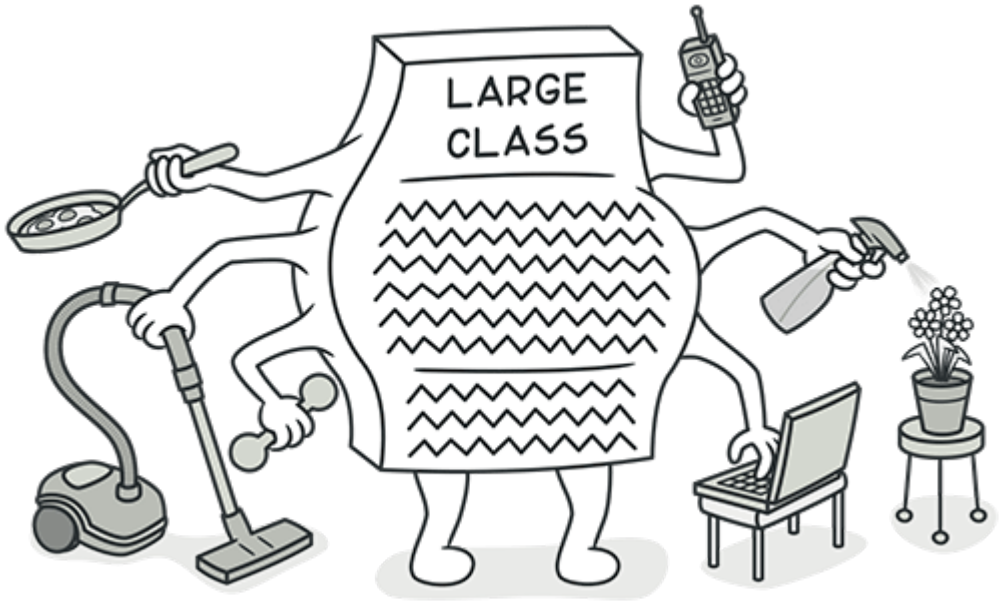
- Can result from duplicated code
- Many temporary or intermediate values
- Many branches/conditionals and loops
- Many comments explaining complex code

- Possible refactoring methods (covered later)
  - Extract Method
  - Replace Method with Method Object
  - Decompose Conditional



Large or Non-Cohesive Classes

https://refactoring.guru/smells/large-class

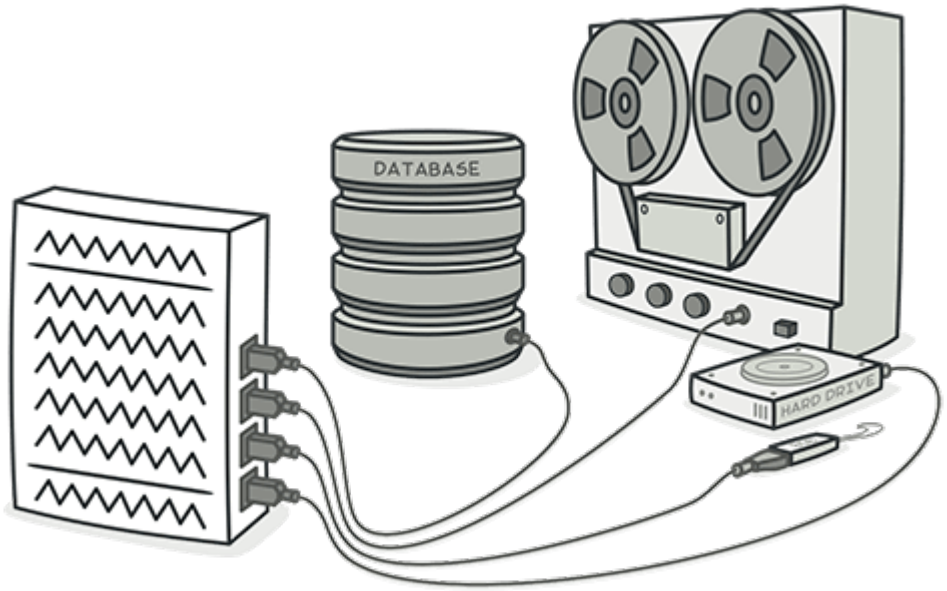


- Can result from duplicated code
- Some other symptoms:
  - Too many instance variables
  - Too many lines of code
  - Not using all the instance variables all the time
  - Different parts to be changed for different reasons (database change, GUI change, business logic)
- Possible refactoring methods (presented later)
  - Move methods
  - Extract Class
  - Extract Subclass/Superclass

Data Classes

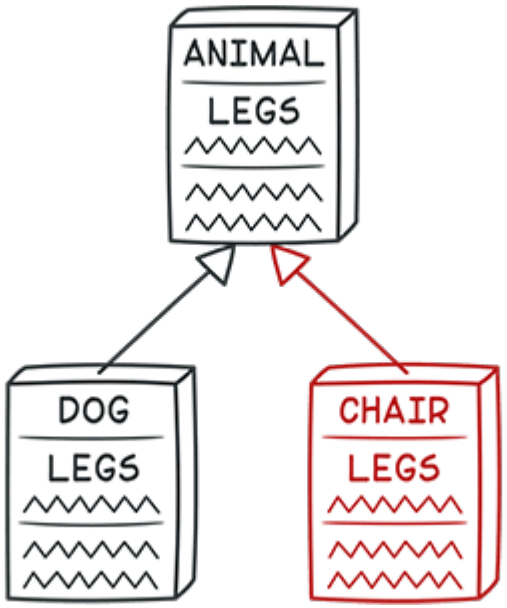
https://refactoring.guru/smells/data-class

- Avoid public fields in such classes (Encapsulation/Information Hiding)
- Classes with fields, getter and setters only – simple data holders
  - something has to be done with the data (even if only displaying)
  - the data should probably go with that behavior
  - Use Move method to move appropriate behavior



Class with Wrong Hierarchy
(M. Fowler named it Refused Bequest)

https://refactoring.guru/smells/refused-bequest



- Subclasses which do not require superclass methods or instance variables
- The smell is worse if subclasses are refusing public interfaces (via NO-Ops or exceptions) rather than private implementations
- If only implementations involved use *Move Field* or *Move Method* to push down the superclass fields and methods to siblings.
- If interfaces are involved we need to use the refactoring technique *Replace Inheritance with Delegation*

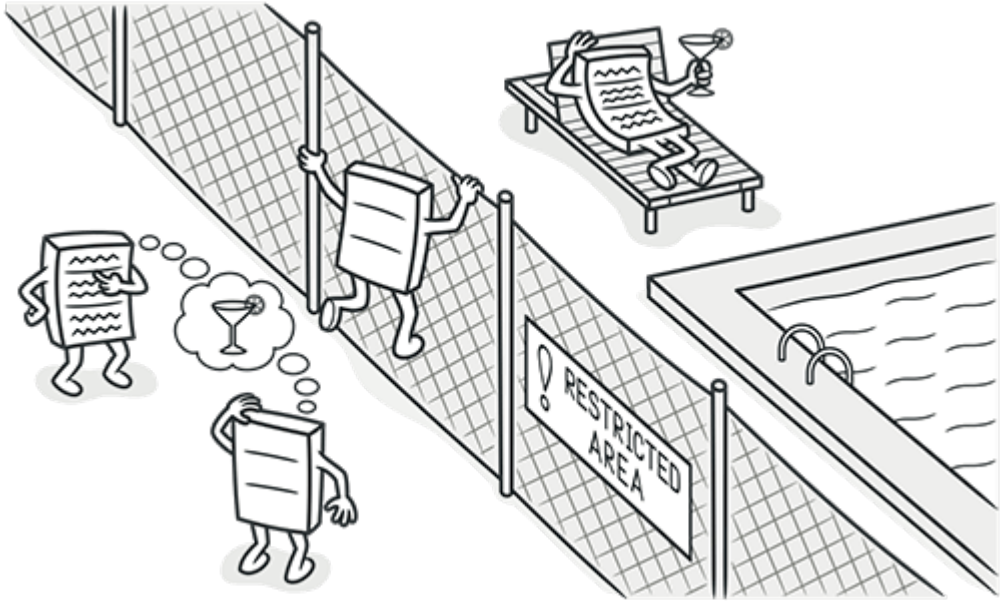
Feature Envy

https://refactoring.guru/smells/feature-envy

A method is more interested in variables and methods of other classes (than its own methods)

SOLUTION: Move method/functionality to “envied” class

```
public class Customer
{
    public Customer( ….. Address currentAddress)
      …
      public string MailingAddress()
      {
       StringBuilder sb = new StringBuilder ();
       sb.Append(currentAddress.AddressLine1);
       sb.Append("\n");
       sb.Append(currentAddress.City + ", " +
           currentAddress.State);
       sb.Append("\n");
       sb.Append(currentAddress.PostalCode);
       return sb.ToString();
     }
```



Various:

- Base Classes depending on Their Derivatives
  - In general, base classes should not directly refer to their derivatives
  - Patterns such as “template method” (covered later in semester) avoid this explicit dependency
- Poorly defined Modules needing wide and deep interfaces (too much information)
  - Increases coupling
- Misplaced Responsibilities (see GRASP)
  - e.g. Where should you define the mathematical constant PI?
- Inappropriate Intimacy between classes
  - Classes accessing each others private fields (parts)
  - C++ has the friend construct
  - In OO classes should be properly encapsulated
  - Use Move-methods or Move-fields to reduce dependency

Various

- Vertical Separation: Defining variables and methods many lines away from their first use
  - Always use tightest scope possible (e.g. local or block scope {})
  - “What happens in Vegas stays in Vegas!”
- Inconsistency – doing things differently
  - To avoid do things in the same order and use the same variable names when used for the same purpose
- Magic (raw) numbers in code
  - Use well named constants e.g. LINES_PER_PAGE
- Switch statements
  - Same switch statement scattered in many places
  - Use polymorphism instead

Various

- Long parameter list
  - Necessary in procedural programming
  - Long parameter list makes logic difficult to follow
  - In OO a method can access all it needs using other known objects
  - Can replace with single Class/Object parameter
  - Parameters can reduce coupling so may be appropriate in some cases
- Inappropriate static declarations
  - static methods that take objects
  - static int max(int x, int y) // OK
  - static double pay(Employee e, double rate)

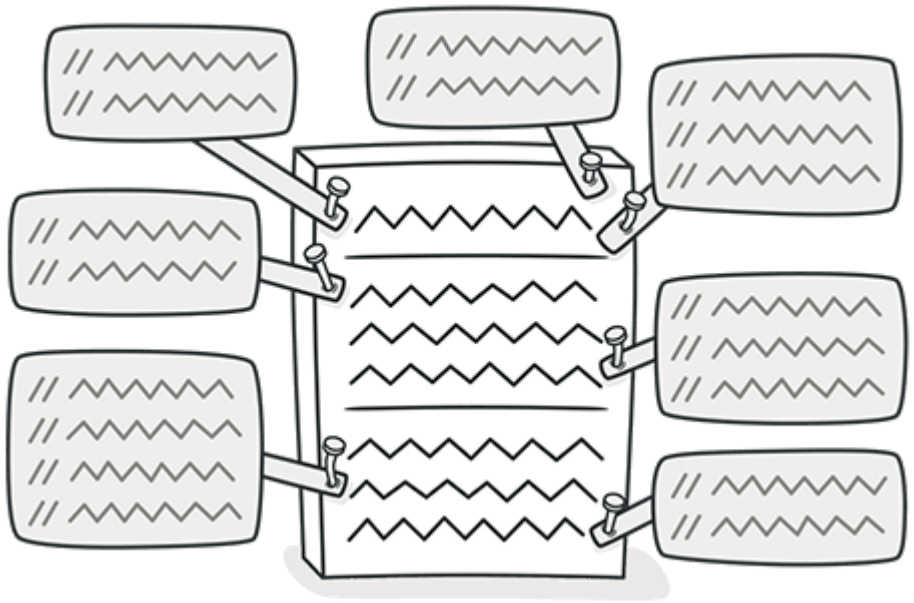
    // Inappropriate prevents dynamic binding

Various

- Shotgun Surgery
  - every time you need to make a change it occurs in several places
  - suggests lack of cohesion
  - create new class to encapsulate concept
- Divergent Change
  - one class is changed for many different reasons
  - opposite of shotgun surgery
  - again suggests poor cohesion
  - create new classes to encapsulate different concepts

Comments: Redundancy

https://refactoring.guru/smells/comments



- Commenting something that adequately describes itself.

  i++; // increment *i*

- A Javadoc that says nothing more than the function signature:

```
/**
* @param sellRequest
* @return
* @throws ManagedComponentException
*/
public SellResponse beginSellItem(SellRequest sellRequest) throws
ManagedComponentException
```

- Comments should say things that the code cannot say for itself
- Comments as "**deodorant**"
  - used to explain incomprehensible code .. fix the code instead!
  - good naming can help a lot
  - **Readable code is always preferable to a comment!**

Comments: Inappropriate Information

- Holding comments better held in other record-keeping systems such as source code control systems, user manual etc.
- Comments should be reserved for technical notes about the code and design.

Good code is its own best documentation. As you're about to add a comment, ask yourself, "How can I improve the code so that this comment isn't needed?" ~Steve McConnell

Comments: Commented-Out Code and Dead Functions

- Can be confusing
  - Increases complexity
  - Is it meaningful and is this code even finished/tested?
  - Is it to be used or fixed in the future?
  - It may depend on other code which no longer in use (not applicable to dead functions)
- How to handle it?
  - Use source code management/versioning system
  - Delete it (can be retrieved from previous version if necessary)

Refactoring Techniques

- These techniques can be grouped into (1-3 covered in this lecture)
  - 1. Composing Methods
  - 2. Moving Features between objects
  - 3. Dealing with Generalization
  - … and more in Fowler Book!
  - Organizing Data
  - Simplifying Conditional Expressions
  - Making Method Calls Simpler

1. (De)Composing Methods

Deals with problems such as long methods and temporary/intermediate variables which make the methods difficult to follow.

- Extract Method
  - Replace temp with query (method)
- Introduce Explaining Variable
- Split Temporary Variable
- Remove assignments to params (less relevant in Java)
- Substitute Algorithm

Extract Method

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

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

Extract Method Motivation

- It improves code clarity and reduces code duplication and facilitates reuse
- Makes reading and understanding higher level methods easier as they read like comments – provided the methods are named properly.
- Attention must be paid to any local variables used in the extracted code
  - May increases coupling between methods although parameters are easy to understand/trace especially if well named
  - Replace temp variable with a query method (**replace temp with query**) that is called when needed
  - Alternatively can turn method into class (GRASP Pure Fabrication) or “Replace Method with Method Object” using Fowler terminology
  - Locals become attributes of new class
  - May need to pass additional params and return intermediate results

Introduce Explaining Variable

- Break a complicated expression into parts that explain the meaning
- Consider extracting a separate method instead unless the dependencies are too complex

```
if ( cw >= CWMIN && exam >= EXMIN  &&
        overall >= OVERALLMIN)
{
   …
}
```

```
final boolean PassCwHurdle = cw >= CWMIN;
final boolean PassExHurdle = exam >= EXMIN;
final boolean PassOverallHurdle = overall >= OVERALLMIN;
if (PassCwHurdle && PassExHurdle && PassOverallHurdle)
{
   …
}
```

Split Temporary Variable

If temporary variable used more than once and is not a loop or aggregate variable make a separate temporary variable for each use.
They can become *explaining* variables

```
double temp = 2 * Math.PI * _radius;
System.out.println (temp);
temp = Math.PI  * _radius * _radius;
System.out.println (temp);
```

```
double circum = 2 * Math.PI * _radius;
System.out.println (circum);
double area = Math.PI  * _radius * _radius;
System.out.println (area);
```

Substitute Algorithm

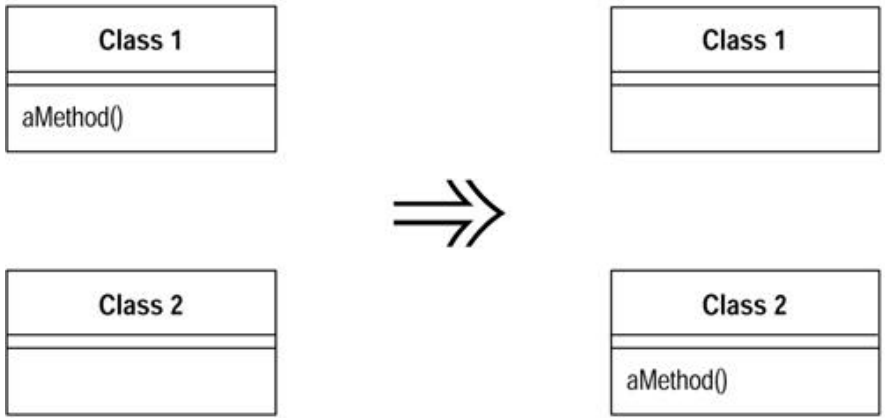
- This refactoring refers to directly improving the code inside a single method
- Used where a cumbersome algorithm/method can be tidied up
  - eliminate unneeded variables
  - restructure clumsy conditional and/or looping structures
  - replace repetition with looping
  - replace multiple variables with a Collection etc.
  - done **in addition to** the previous composing techniques

2. Moving Features Between Objects

- A fundamental decision in OO design is the relationships between, and placement of responsibility in, objects
  - we covered this with GRASP
- Refactoring can be in response to an early design that proves problematic or an initial good design that has degraded over time
- Design structure can be improved as follows
  - Move Method
  - Move Field
  - Extract class/interface
  - Inline class
  - Hide Delegate

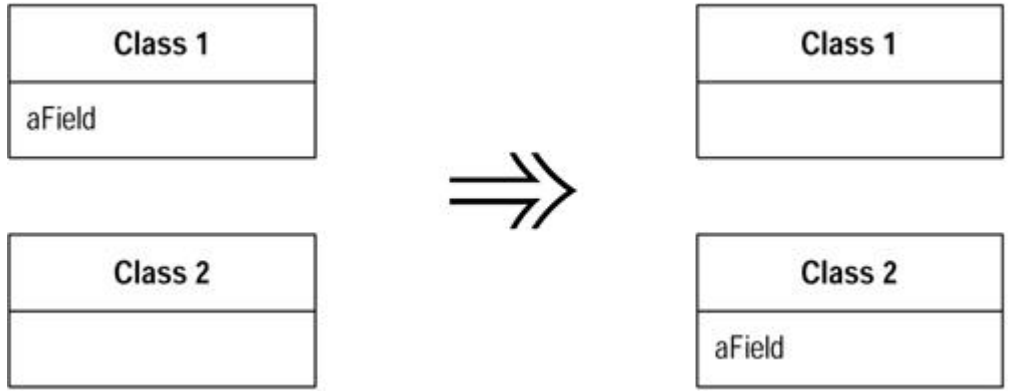
Move Method

- Move a method when it will be used by or using more features from another class (old method can be forwarded or removed altogether)
  - e.g. as per GRASP *information expert* and *creator*
- In response to *Feature Envy* or *Inappropriate Intimacy*
- Reduces coupling by reducing the amount of collaboration
- Increases cohesion by placing it closer to where it used
- The code must be adjusted to take into account references and exceptions
- Can move groups of related methods together
- Need to consider any interface or inheritance dependencies
- If many references are involved delegation/forwarding may be easier.



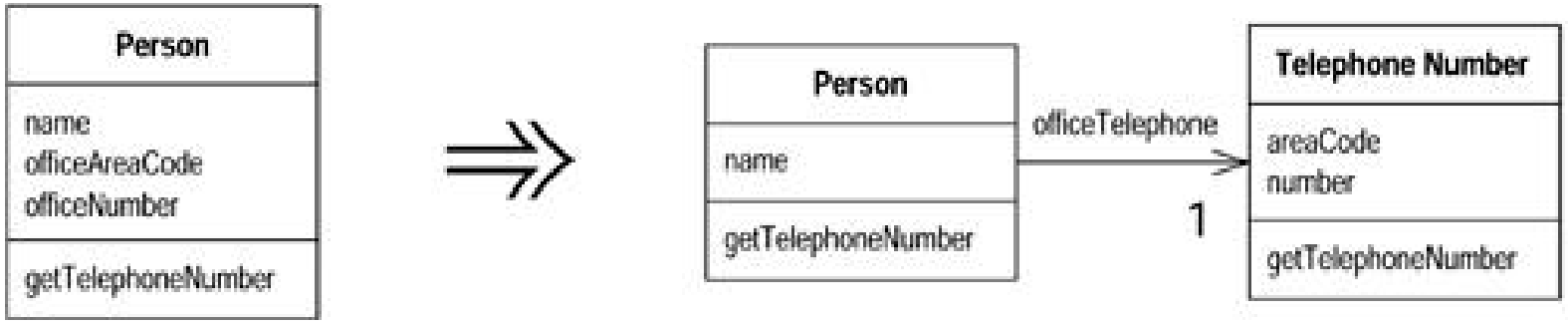
Move Field/Attribute

- Similar to moving a method
- Move a field/attribute when it will be used by more features from another class
- Can look at which class is using the getters and setters the most
- If field is public then encapsulate with getters/setters
- Related to *GRASP Information Expert*



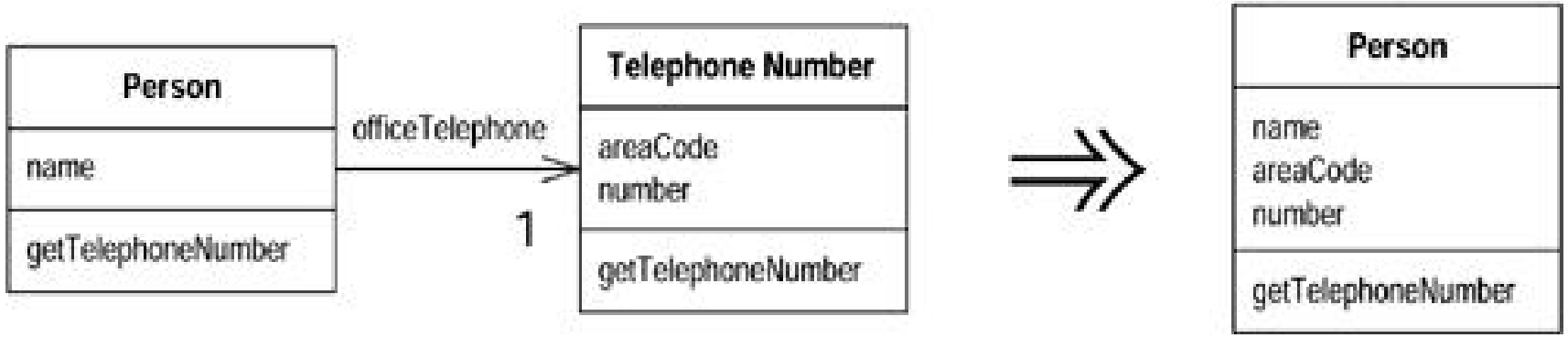
Extract/Split Class

- Classes often grow over time handling more responsibility --- becoming too complicated
- Create a new class and move related methods and fields when you have a non-cohesive class doing work that should be done by two (or more).
- See *Pure Fabrication* in GRASP



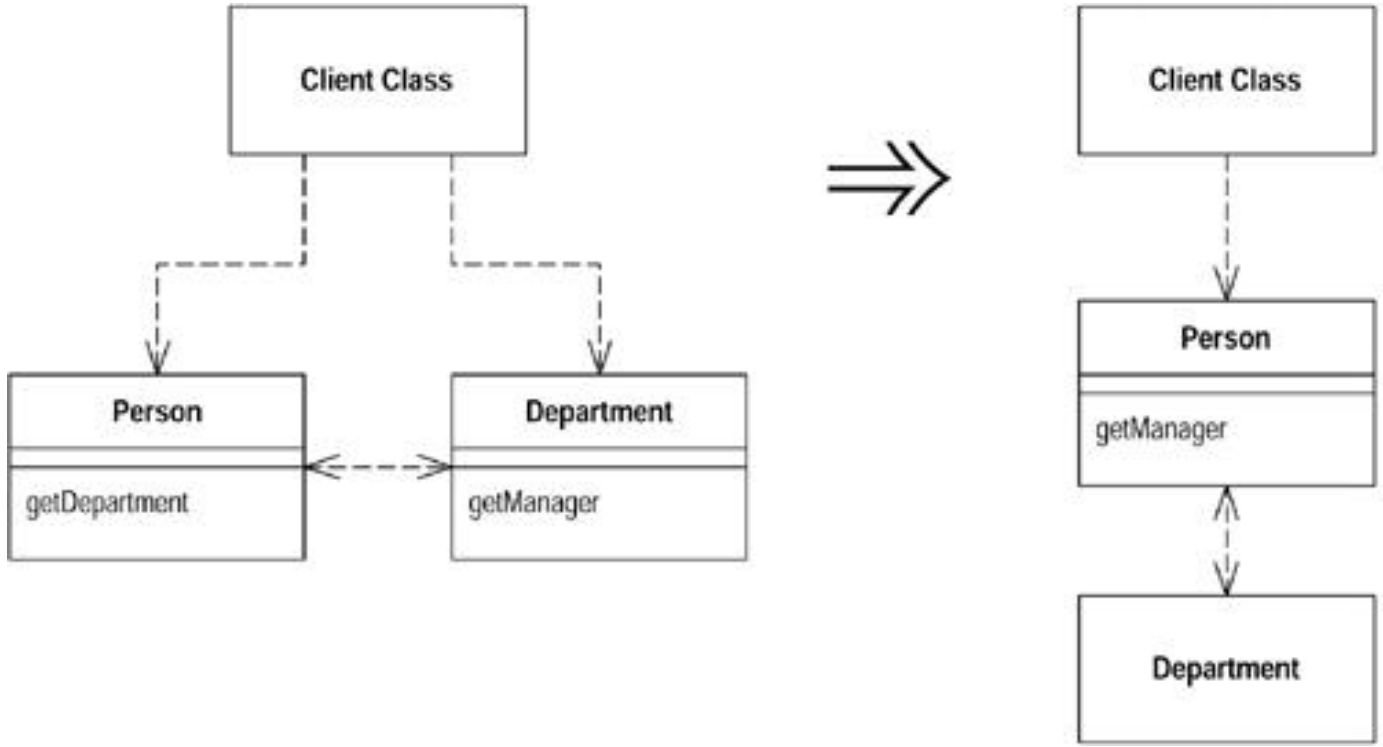
Inline Class

- Reverse of extract class
- Use when a class need not exist anymore often as a result of refactoring.
- e.g. we have moved methods and attributes into different classes and not much is left, might as well finish it off for good!



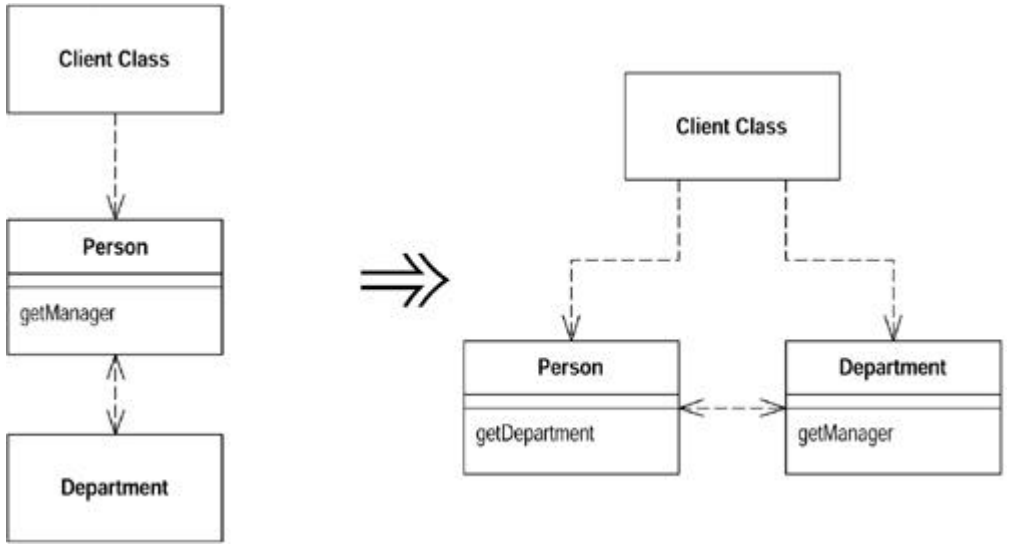
Hide Delegate

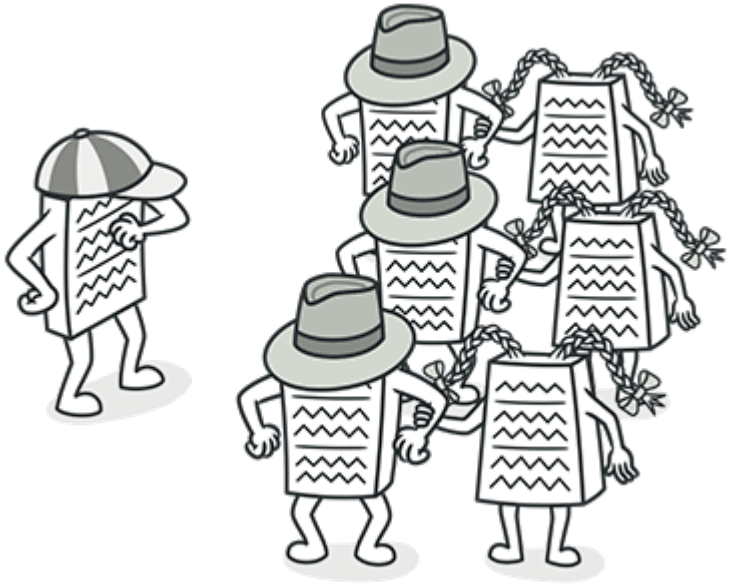
- To avoid a client calling the provider's delegate class create additional methods on the provider
- Reduces coupling (see diagram) and change impact
- See *Coupling* examples in GRASP
- Do still need to consider cohesion (of the Person class in this example) .. see "Remove Middle" next)



Remove Middle (Man)

- Opposite of Hide Delegate
- The problem with Hide Delegate: Every method in delegate needs a corresponding method in the provider … may be clumsy for large interfaces.





https://refactoring.guru/smells/middle-man

Introduce Foreign Method (Helper Method)

- Useful when you are not allowed to change the provider class (such as `java.util.Date`) to add additional operations (e.g. get next day)
- Such an operation can be hard coded in the client but if it is used repeatedly a single method can be added to the client.
- Or can add the new method to a new utility class (e.g. `DateUtils`) for better cohesion (see *Local Extension* next)

```
// hard coded (gets repetitive!)
Date newStart = new Date (previousEnd.getYear(), previousEnd.getMonth(),
                        previousEnd.getDate() + 1);
```

```
Date newStart = nextDay(previousEnd);
// Foreign/Helper method
private static Date nextDay(Date arg) {  return new Date (arg.getYear(),
                arg.getMonth(), arg.getDate() +  1);
}
```

Introduce Local Extension

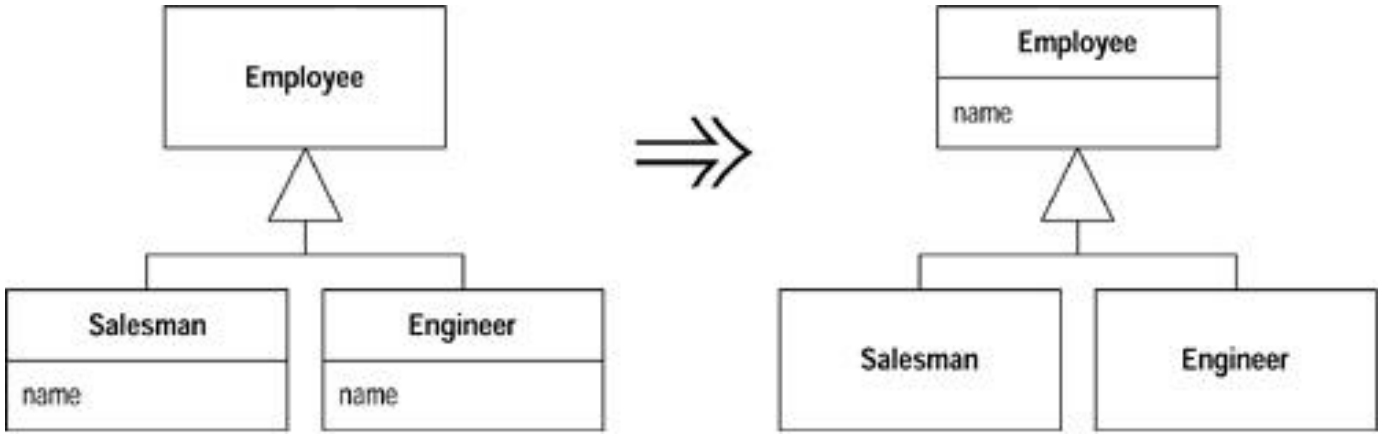
- Alternative to Foreign Method i.e. you need additional methods in a provider class which you cannot alter
- Create a subclass or wrapper class (which forwards to the original).
- This class has the advantage of reduced coupling from the client
- all the data and methods are packaged together
- however we do introduce extra coupling to the original class

3. Dealing with Generalization

- Generalization requires moving methods and fields within the hierarchy.
- Generalization may also require creating new classes during refactoring.
- When inheritance is not appropriate code can be refactored to use delegation.
- This section covers:
  - Pull up/Push down field/method
  - Extract Subclass/Superclass/Interface
  - Replace Inheritance

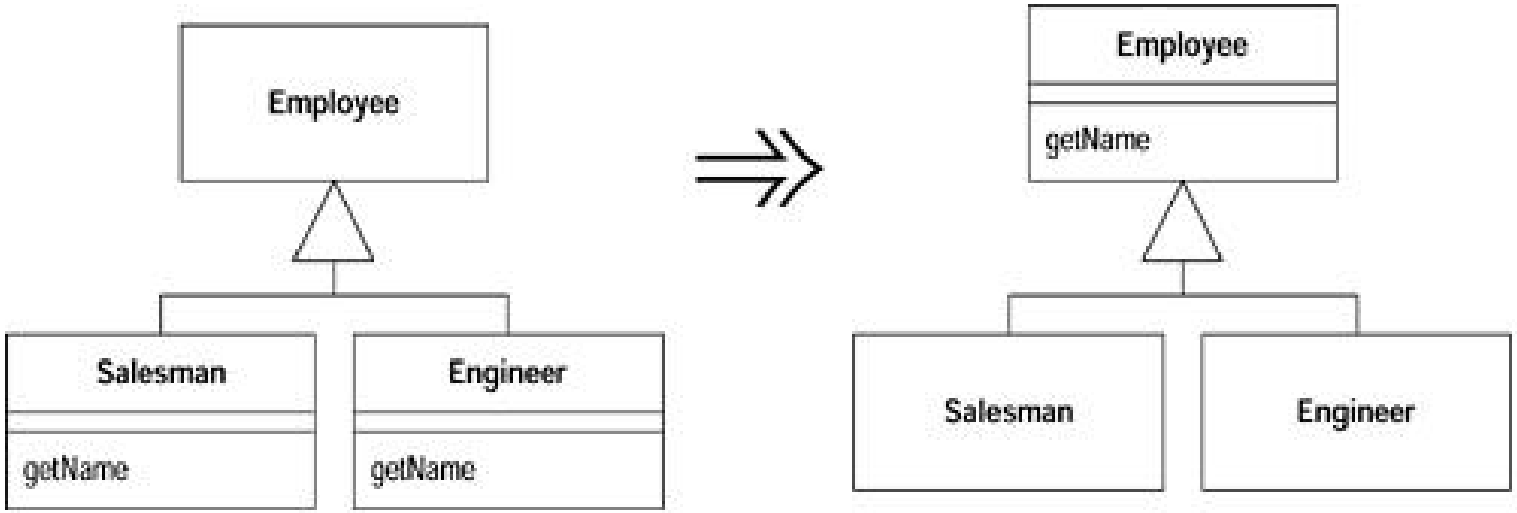
Pull Up Field

- If two subclasses have the same field move it to the superclass



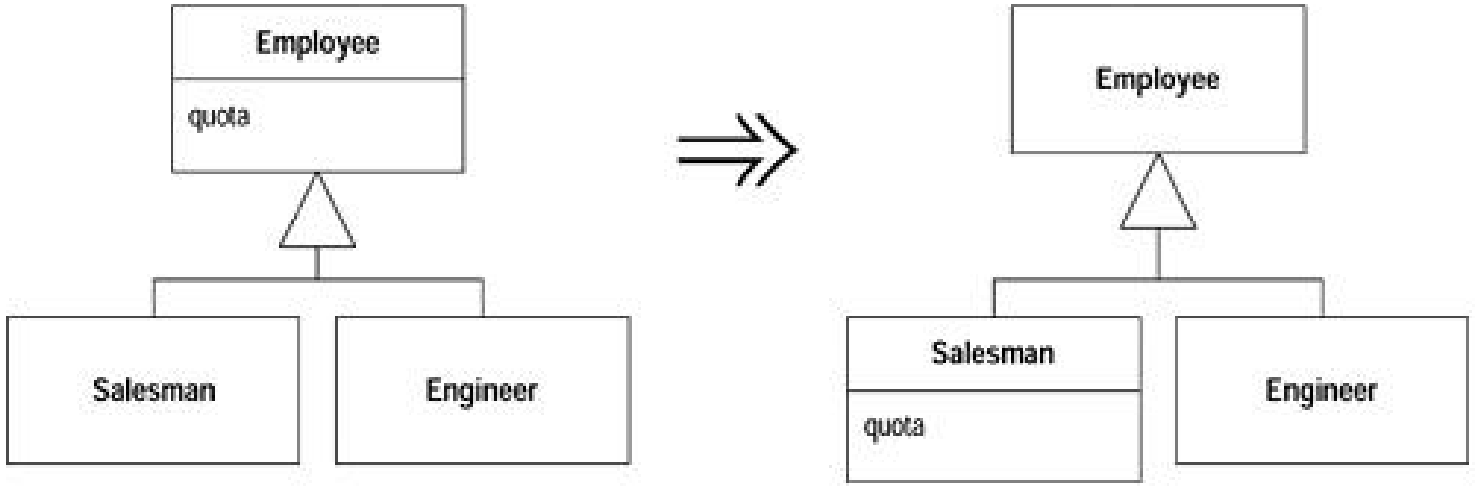
Pull Up Method

- When you have methods with identical code move them to the superclass
- Duplicate methods are breeding ground for bugs/hard work!
- If there are only small differences then use template methods (covered later with GoF patterns) to factor out the difference
- Can apply to constructors as well



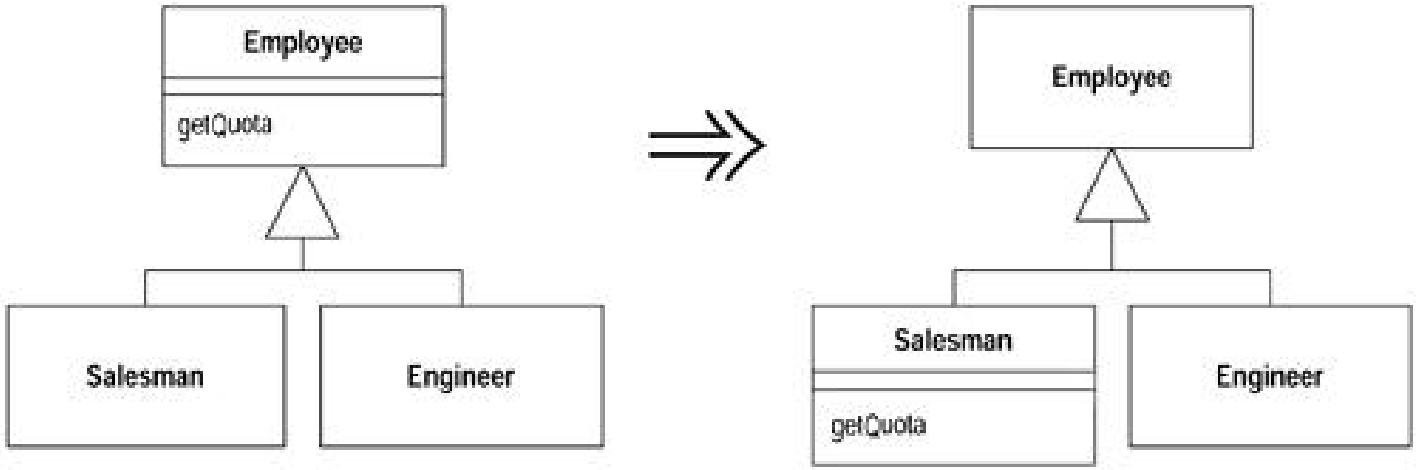
Push Down Field

- If a field is used only by some of the subclasses move it to those.



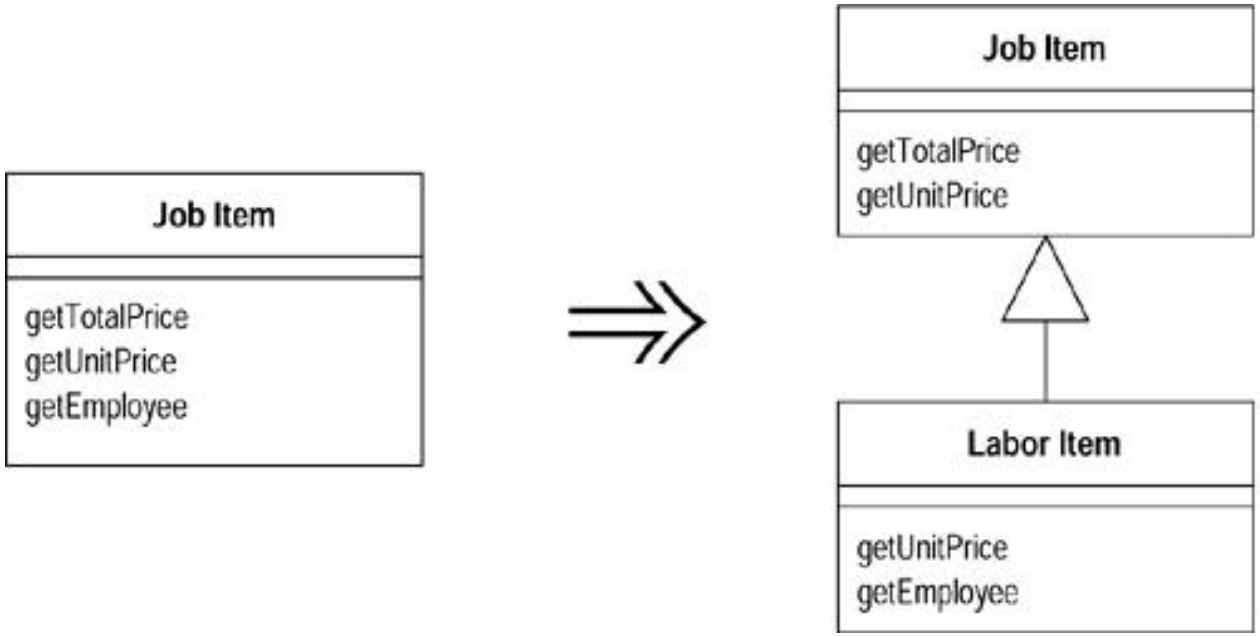
Push Down Method

- If the behavior of a superclass is relevant only to some of its subclasses, move it to those.
- Can apply to constructors as well



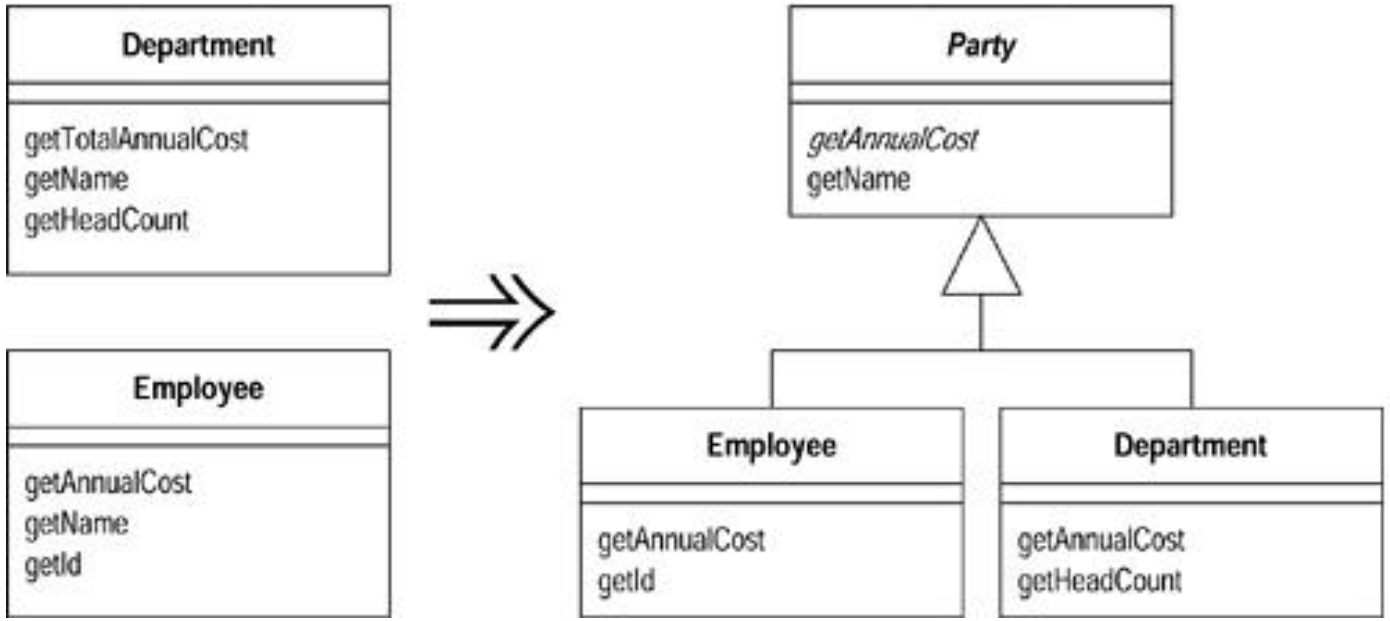
Extract Subclass

- If a class has features that are used only in some instances create a subclass for the subset of features.



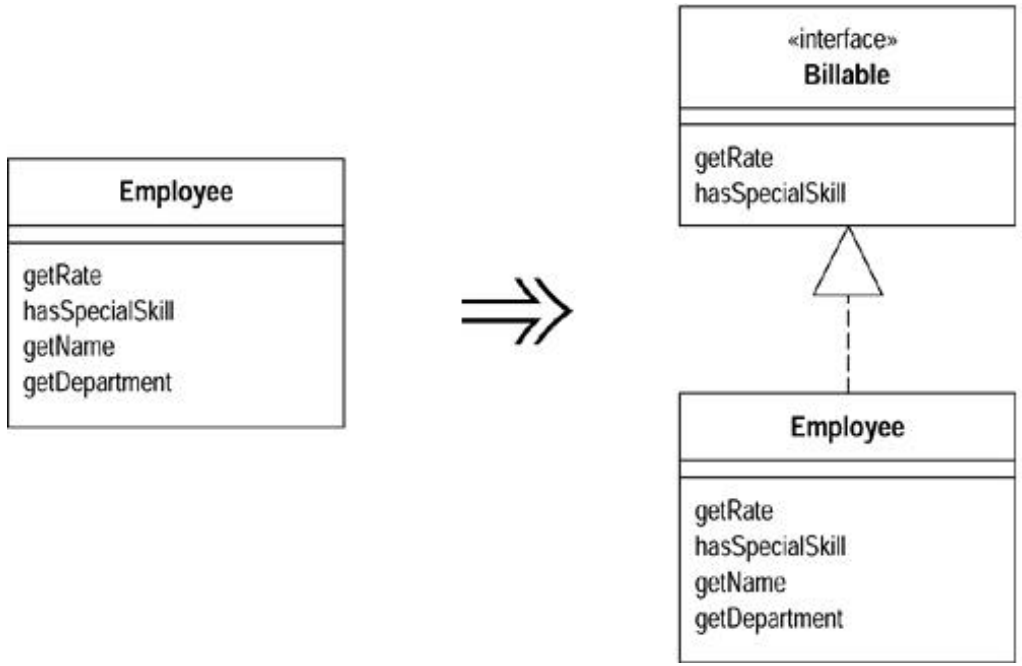
Extract Superclass

- When you have two classes with similar features move the common features to the superclass.



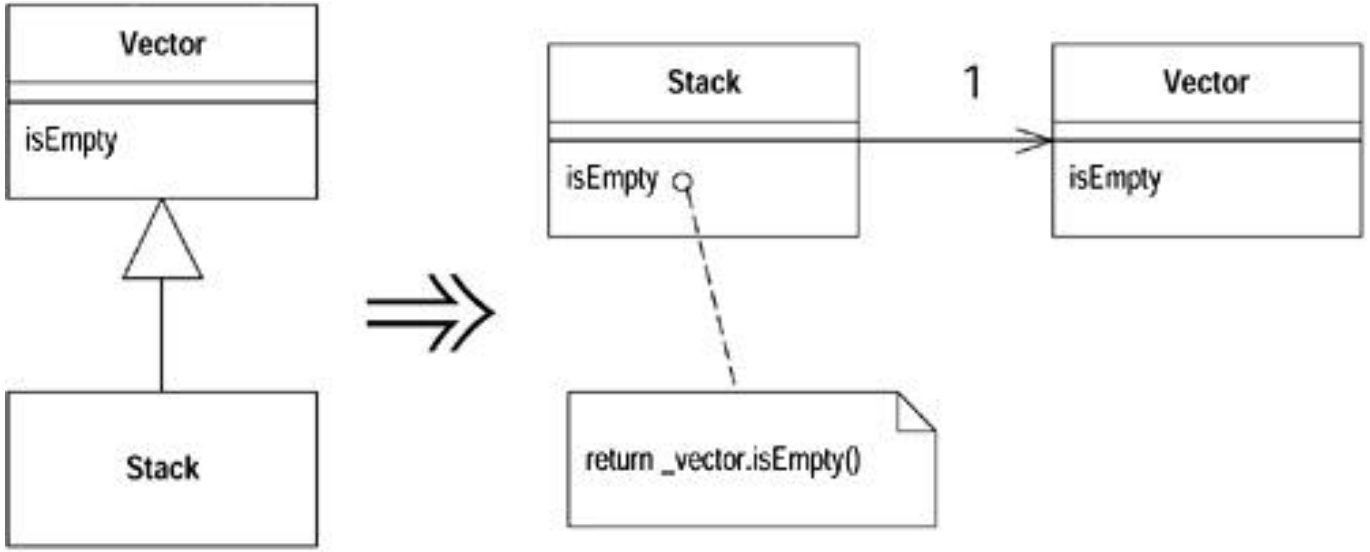
Extract Interface

- When two or more classes have part of their interface in common extract the subset into an interface
- Interfaces are also good whenever classes have distinct roles in different situations
- The only way to support multiple inheritance for responsibilities in Java
- The only way to support polymorphism for non-hierarchical commonality in Java



Replace Inheritance with Delegation

- When a subclass uses only part of a superclass interface create a field to refer to the superclass through which its methods can be called (and remove the subclassing)
- Forward only the required methods so unused methods cannot be accidentally called
- Addresses previously discussed *Refused Bequest* “code smell”
- The cost is extra delegating methods which can be cumbersome and mildly increases complexity and compromises readability (although performance impact is negligible)
- Can go the other way and **replace delegation with inheritance** if you end up forwarding the whole interface!



Summary

- Refactoring is the process of changing a software system in such a way that it does not alter external behaviour of the code yet improves its internal structure
  - occurs continuously during development
  - in small increments or major restructuring
- Based on the understanding that good software does not just mean “does it work”.
- Changes made over time can cause loss of structure or distort the original design – causing the code to “smell”
  - Keep a look out for bad trends and fix them early!