CLEAN CODE, CODE SMELLS, REFACTORING

AND RELATED PRINCIPLES

Barbora Bühnová
buhnova@fi.muni.cz

LAB OF SOFTWARE ARCHITECTURES
AND INFORMATION SYSTEMS

FACULTY OF INFORMATICS
MASARYK UNIVERSITY, BRNO

lasaris

Outline of the lecture

- Motivation
- The role of naming
  - Classes, methods, variables
- The role of code structure
  - S.O.L.I.D. principles
  - DRY and GRASP principles
- Bad code smells
- Refactoring
  - When, how, where

Why are we here?

DO YOU KNOW HOW EXPENSIVE IT IS TO HAVE DEVELOPERS AROUND?

lasaris

How is that related to code quality?

Your codebase is just **like the database of a website**.

- **Read:Write** ratio is like **10:1**

… AND BRAINS FAIL AS CACHES

Goal is cheap Reads

- People will read your code **again** and **again** and **again** and …
- How long does it really take to **understand** your module?

lasaris

Why is our code hard to understand?

- Management?
- Unclear requirements?
- Customers?
- Schedules?
- Requirement changes?

NO. BECAUSE WE WROTE IT LIKE THAT.

SO HOW TO GET IT RIGHT?

lasaris

Two rules of understandable code

1. **Name things right**

- Reveal intent, self-documented code. What about comments?

2. **Balance code structure**

- No God classes, long methods and other bad code smells.

IT IS EASY TO WRITE CODE THAT A MACHINE UNDERSTANDS. WRITING CODE THAT ANOTHER HUMAN CAN UNDERSTAND IS A LOT HARDER.

lasaris

Let's name things!

CLASSES, METHODS, VARIABLES, …

lasaris

Classes

- Name it after its purpose – **Single responsibility principle!**

THE CLASS NAME IS THE SINGLE MOST IMPORTANT DEFINITION OF WHAT FITS INTO THAT CLASS.

- Using generic names throws that away!
  `ApplicationManager`,
  `FrameworkDataInformationProvider`,
  `UtilityDataProcessor`

lasaris

Proper class names lead to smaller classes

- Should we let our `Email` class figure out attachment mime types?
- By always asking if stuff fits we can 'discover' new classes in our applications
- `EmailAttachment`, `ImageEmailAttachment`?

lasaris

And we care because?

- Big classes rob you of all OO benefits
- You depend on way to much other stuff
- Usually tightly coupled
- You can extend from those classes and chances are you can't swap out either
- You don't get nice interfaces as well

lasaris

Methods

- Does it return a boolean?
- Call it hasX or isX

- **Hidden booleans**

```
status = user.getStatus();
if(status == user.STATUS_BANNED) {
}

if(user.isBanned()) {
}
```


lasaris

Getters and Setters

- **Getter** **retrieve** internal state
- **Setters** **modify** internal state
- Both should not modify anything else
- Don’t make your setters into liars

Important

- Make sure that setters have **no return value**
- Use setters **only if** you want to **allow this**

lasaris

No boolean parameters

- If you don't have a very good reason!

```
user.setAdminStatus(false);
user.setAdminStatus(true);
```

- vs.

```
user.revokeAdminRights();
user.grantAdminRights();
```


lasaris

Classes are nouns, methods start with verbs!

```
X.createStuff(); X.deleteStuff()
X.dispatchCall(); X.subscribeUser();
```

- But never

```
user.admin(); or user.bag();
list.subscription();
```


lasaris

Agree on verbs for actions

- Can you tell me the difference between

```
directory.delete(entry);
```

and

```
directory.remove(entry);
router.dispatch/delegate(call);
list.add/append(user);
```

- Different actions need distinguishable names!
  Even if that makes function names longer.

lasaris

Always favor long function names? NO!

- A **short** an precise **public API** is important
- Precise names help a lot to create readable client code.

- But you want **long privates**
- Internally used functions can be named as verbosely as needed to communicate their intent and behavior.

lasaris

Readable classes

Lots of small descriptive methods, best arranged as:

1. First you build it
2. Then you call it
3. Then it does work
4. Most work happens in private methods

```
class Log {
  public Log(SplFileInfo fileInfo) {}
  public void log() {}
  private bool isLogfileWriteable() {}
  private void createLogfileIfNecaccary() {}
  private void writeToLogfile() {}
  private void flushCurrentLogfileBuffer() {}
}
```


lasaris

Last thing to name are Variables

Rules of thumb:

- Descriptive function parameters
- **Big scope:** long name
- **Short scope:** short name

The last thing? And what about **comments**?

- Self-documenting code

lasaris

Enough of naming?

LET'S MOVE ON TO CODE STRUCTURE!

lasaris

S.O.L.I.D. principles

- **S**ingle Responsibility Principle
- **O**pen / Closed Principle
- **L**iskov Substitution Principle
- **I**nterface Seggregation Principle
- **D**ependency Inversion

SRP: The Single Responsibility Principle

THERE SHOULD NEVER BE MORE THAN ONE REASON FOR A CLASS TO CHANGE.

Robert C. "Uncle Bob" Martin

- Why a **reason for a class to change**? Why not responsibility? Because this is all interconnected!
  - The more **responsibilities**, the more **dependencies**.
  - The more **dependencies**, the higher risk of **change propagation**.
  - The bigger **change propagation**, the higher **risk of error**.
- Following SRP leads to **lower coupling and higher cohesion**.
- Many small classes with distinct responsibilities result in a **more flexible design**.

lasaris

SRP: The Single Responsibility Principle

- ```
  public abstract class BankAccount
  {
    double Balance { get; }
    void Deposit(double amount);
    void Withdraw(double amount);
    void Transfer(double amount, IBankAccount toAccount);
    void AddInterest(double amount);
  }
  ```

  Refactor to:

  ```
  public abstract class BankAccount
  {
    double Balance { get; }
    void Deposit(double amount);
    void Withdraw(double amount);
    void Transfer(double amount, IBankAccount toAccount);
  }

  public class CheckingAccount : BankAccount
  {
  }

  public class SavingsAccount : BankAccount
  {
    public void AddInterest(double amount);
  }
  ```


lasaris
  ```

SRP Summary

- Many small classes with distinct responsibilities result in a **more flexible design**.
- Following SRP leads to **higher cohesion** and **lower coupling**.

- **Cohesion…**
  - How strongly-related and focused are the various responsibilities of a module.
  - Classes with low cohesion have a *split personality*.

- **Coupling…**
  - The degree to which each program module relies on each one of the other modules.
  - Coupling is directly related to decomposition and you need to keep it in mind when you decompose.

OCP: The Open/Closed Principle

Open to Extension

New behavior can be added in the future

Closed to Modification

Changes to source or binary code are not required

- Bertrand Meyer originated the OCP term in his 1988 book, *Object Oriented Software Construction*.

In OOP, **abstractions** include:

- Interfaces
- Abstract classes

lasaris

Why is that a problem?

- We want to avoid introducing changes that *cascade* through many modules in our application
- **Writing new classes is less likely to introduce problems!**
  - Nothing depends on new classes (yet).
  - New classes have no legacy coupling to make them hard to design or test.
- **Remember *TANSTAAFL***
  - *There Ain't No Such Thing As A Free Lunch*
  - OCP adds complexity to design!
  - **Don't apply OCP at first**
    If the module changes once, accept it.
    If it changes a second time, refactor to achieve OCP

lasaris

OCP Summary

- Conformance to OCP yields flexibility, reusability, and maintainability.
- Know which changes to guard against, and resist premature abstraction.

LSP: The Liskov Substitution Principle

THE LISKOV SUBSTITUTION PRINCIPLE STATES THAT SUBTYPES MUST BE SUBSTITUTES FOR THEIR BASE TYPES.

Agile Principles, Patterns, and Practices in C#

- Named for Brabara Liskov, who first described the principle in 1988.

Substitutability:

- Child classes must not:
  - Remove base class behavior
  - Violate base class invariants
- And in general must not require calling code to know they are different from their base type.
- To follow LSP, derived classes must not violate any constraints defined (or assumed by clients) on the base classes.

lasaris

The Problem

- Non-substitutable code breaks polymorphism
- Client code expects child classes to work in place of their base classes
- "Fixing" substitutability problems by adding if-then or switch statements quickly becomes a maintenance nightmare (and violates OCP)

LSP violation:

```
foreach (var emp in Employees)
{
  if( emp is Manager )
  {
    printer.PrintManager( emp as Manager );
  }
  else
  {
    printer.PrintEmployee( emp );
  }
}
```

LSP Summary

- Conformance to LSP allows for proper use of polymorphism and produces more maintainable code.
- Remember IS-SUBSTITUTABLE-FOR instead of IS-A.

Consider Refactoring to a new Base Class

- Given two classes that share a lot of behavior but are not substitutable.
- Create a third class that both can derive from.
- Ensure substitutability is retained between each class and the new base.

lasaris

ISP: The Interface Segregation Principle

THE ISP STATES THAT CLIENTS SHOULD NOT BE FORCED TO DEPEND ON METHODS THEY DO NOT USE.

Agile Principles, Patterns, and Practices in C#

- That is, prefer small, cohesive interfaces to "fat" interfaces.
- What is an interface?

  Interface keyword/type

  ```
  public interface IDoSomething { … }
  ```

  Public interface of a class

  ```
  public class SomeClass { … }
  ```


lasaris

The Problem

- Client references a class but only uses small portion of it
- Interface Segregation violations result in classes that depend on things they do not need, increasing coupling and reducing flexibility and maintainability
- Unimplemented interface methods

When do we fix ISP?

- **Once there is pain**
  - If there is no pain, there's no problem to address.
- **If you find yourself depending on a "fat" interface you own**
  - Create a smaller interface with just what you need
  - Have the fat interface implement your new interface
  - Reference the new interface with your code
- **If you find "fat" interfaces are problematic but you do not own them**
  - Create a smaller interface with just what you need
  - Implement this interface using an Adapter that implements the full interface

ISP Summary

- **Don't force client code to depend on things it doesn't need.**
- Keep interfaces small, cohesive, and focused
- Whenever possible, let the client define the interface
- Whenever possible, package the interface with the client
  - Alternately, package in a third assembly client and implementation both depend upon
  - Last resort: Package interfaces with their implementation

DIP: The Dependency Inversion Principle

HIGH-LEVEL MODULES SHOULD NOT DEPEND ON LOW-LEVEL MODULES. BOTH SHOULD DEPEND ON ABSTRACTIONS.

lasaris

Traditional Programming and Dependencies

- **High Level modules call Low Level modules**
- User Interface depends on
  - Business Logic depends on
  - Infrastructure
  - Utility
  - Data Access
- Static methods are used for convenience or as Façade layers
- Class instantiation / Call stack logic is scattered through all modules
  - Violation of Single Responsibility Principle

lasaris

The Problem

- Dependencies Flow Toward Infrastructure
- Core / Business / Domain Classes Depend on Implementation Details
- Result
  - Tight coupling
  - No way to change implementation details without recompile (OCP violation)
  - Difficult to test

lasaris

Classes should declare what they need

- Class constructor should require any dependencies the class needs.
- Classes whose constructors make this clear have explicit dependencies. Classes that do not, have implicit, hidden dependencies.

```
public class HelloWorldExplicit
{
  private readonly DateTime timeOfGreeting;

  public HelloWorldExplicit(DateTime timeOfGreeting)
  {
    timeOfGreeting = timeOfGreeting;
  }

  public string Hello(string name)
  {
    if (timeOfGreeting.Hour < 12) return "Good morning, " + name;
    if (timeOfGreeting.Hour < 18) return "Good afternoon, " + name;
    return "Good evening, " + name;
  }
}
```

DIP: Summary

- Depend on abstractions.
- Don’t force high-level modules to depend on low-level modules through direct instantiation or static method calls.
- Declare class dependencies explicitly in their constructors.

Dependency injection

- Inject dependencies via constructor, property, or parameter injection.

DRY: Don't Repeat Yourself

"EVERY PIECE OF KNOWLEDGE MUST HAVE A SINGLE, UNAMBIGUOUS REPRESENTATION IN THE SYSTEM."

The Pragmatic Programmer

Variations include:

- Once and Only Once
- Duplication Is Evil (DIE)

lasaris

No duplication

- What can be duplicated?
  - Code Blocks
  - Methods
  - Classes
  - Functions
  - Components
  - Exceptions

- **Minimalistic code**

- What minimalism mean? What do we want to minimize? What can be minimized?

lasaris

Analysis

- Magic Strings/Values
- Duplicate logic in multiple locations
- Repeated if-then logic
- Conditionals instead of polymorphism
- Repeated Execution Patterns
- Lots of duplicate, probably copy-pasted, code

DRY: Summary

- Repetition breeds errors and waste
- Refactor code to remove repetition

Repetition in process

- **Testing**
  - Performing testing by hand is tedious and wasteful
- **Builds**
  - Performing builds by hand is tedious and wasteful
- **Deployments**
  - Performing deployments by hand is tedious and wasteful

GRASP

- General Responsibility Assignment Software Patterns
- A set of design patterns or aspects emphasizing good coding practices.
- Might be useful, but only after you have good understanding of **SOLID** principles.

There is quite a lot to it. Google it if you are interested!

Ready for more?

BUT WHAT IF WE ALREADY HAVE THE CODE?
HOW CAN WE FIND OUT IT SMELLS AND HOW CAN REFACTORING HELP US WITH THAT?

lasaris

Bad Code Smells

- Different abstraction levels (not top down - mixed, skipping levels, mixing levels in one method)
- Low cohesion (God classes, long methods, script/program wrapped as a class)
- Circular dependencies (between classes - mother of all tight couplings)
- Duplicated code
- Long parameter list
- … and many many more

What is refactoring?

- Refactoring is the process of changing a software system in such a way that it **does not alter the external behavior** of the code yet **improves its internal structure**
- Refactoring (noun): a change made to the internal structure of software to make it easier to understand and **cheaper to modify** without changing its observable behavior.
- Refactor (verb): to **restructure software** by applying a series of refactorings without changing its observable behavior.

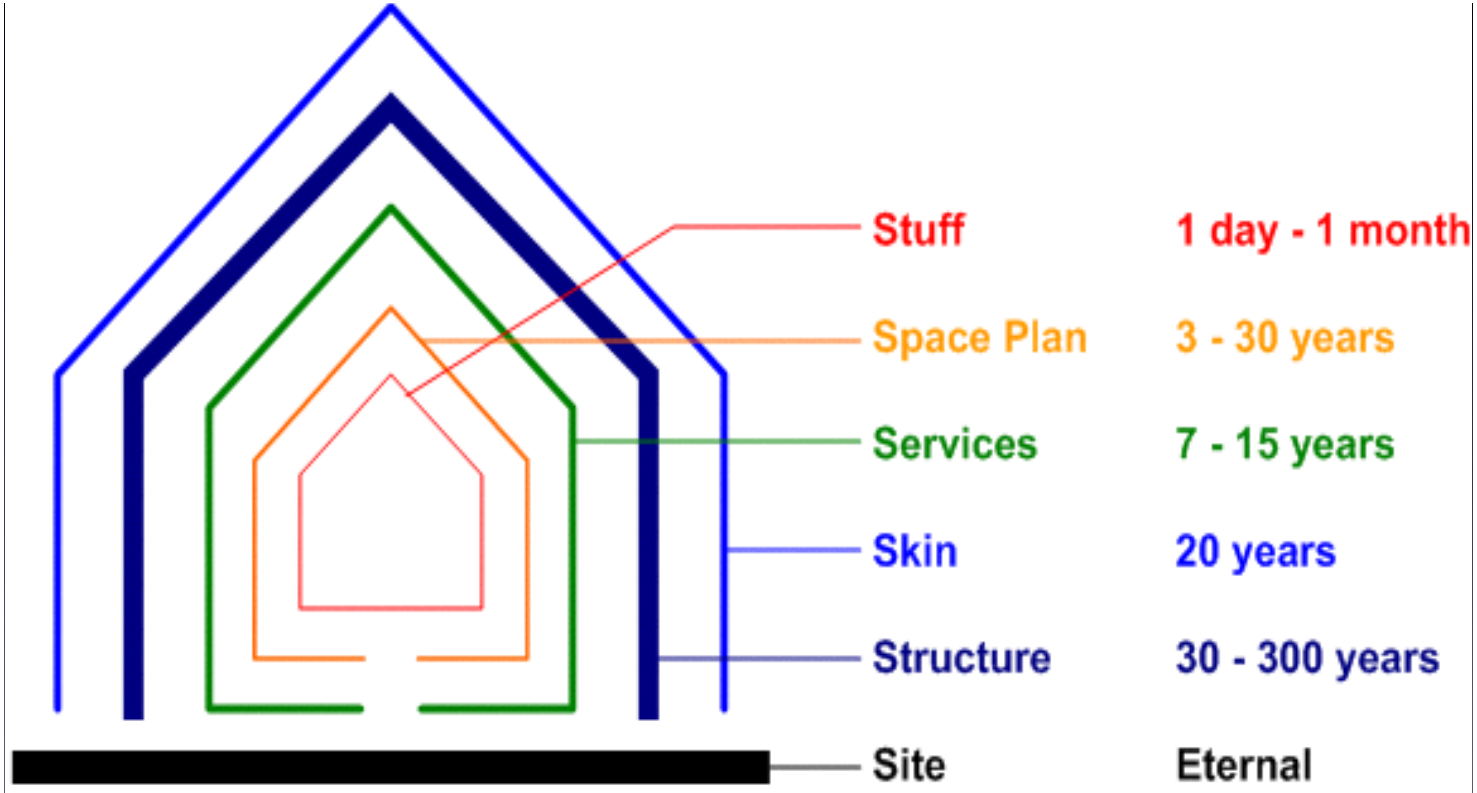
When to refactor?

When you have the refactoring hat on your head!

- As part of the **routine** (e.g. TDD).
- After you find **weak code** (boy scout rule), or need to **fix a bug**.
- Before and/or after you introduce **code of a new feature** (or a new technology like IoC container).
- Long term **planned** refactoring

Only when it leads to faster delivery and better maintenance.
Clean code is a means to this end.

Where to refactor?



Stewart Brand's 6 S's from *How Buildings Learn*

How to refactor?

- Use IDE all the time (even when renaming!).
  Run tests before and after!
- Boundary tests (testing published interfaces) should stay green.
- Know most common refactorings (extract ..., rename, move, introduce) - learn to use them as part of your IDE mastery.

And how Martin Fowler does it?

- M.F: “Whenever I do refactoring, the first step is always the same. I need to build a solid set of tests for that section of code. The tests are essential because even though I follow refactorings structured to avoid most of the opportunities for introducing bugs, I'm still human and still make mistakes. Thus I need solid tests.”

Tip:

Before you start refactoring, check that you have a solid suite of tests. These tests must be self-checking

Refactoring methods

Extract Method

- You have a code fragment that can be grouped together. Turn the fragment in to a method whose name explains the purpose of the method.
- Extract Method is one of the most common refactorings being done. One looks at a method that is too long or look at code that needs a comment to understand its purpose. One then turns that fragment of code into its own method.

Extract method

Before refactoring

```
void printOwing(double amount)
{
  printBanner();

  //print details
  WriteLine("name:" + name);
  WriteLine("amount" + amount);
}
```

After Refactoring

```
void printOwing(double amount)
{
  printBanner();
  printDetails(amount);
}

void printDetails(double amount)
{
  WriteLine("name:" + name);
  WriteLine("amount" + amount);
}
```

Inline method

Before Refactoring

```
int getRating()
{
  return (moreThanFiveLateDeliveries()) ?
  2 : 1;
}

boolean moreThanFiveLateDeliveries()
{
  return numberOfLateDeliveries > 5;
}
```

After Refactoring

```
int getRating()
{
  return (numberOfLateDeliveries > 5) ?
  2 : 1;
}
```

Maximum method length?

- Maximum method length?
  6 Lines ought to be enough for everybody?

Smaller is harder to write

- Writing ONLY small functions is a SKILL
- It is **easy** to write big functions!
- But the first change makes it all worth!

lasaris

Split temporary variable

- You have a temporary variable assigned to more than once, but is not a loop variable nor a collecting temporary variable. Make a separate temporary variable for each assignment.

Before Refactoring

```
double temp = 2 * (height + width);
WriteLine(temp);
temp = height * width;
WriteLine(temp);
```

After Refactoring

```
double perimeter = 2 * (height + width);
WriteLine(perimeter);
double area = height * width;
WriteLine(area);
```

Remove assignments to parameters

- The code assigns to a parameter.
  Use a temporary variable instead.

Before Refactoring

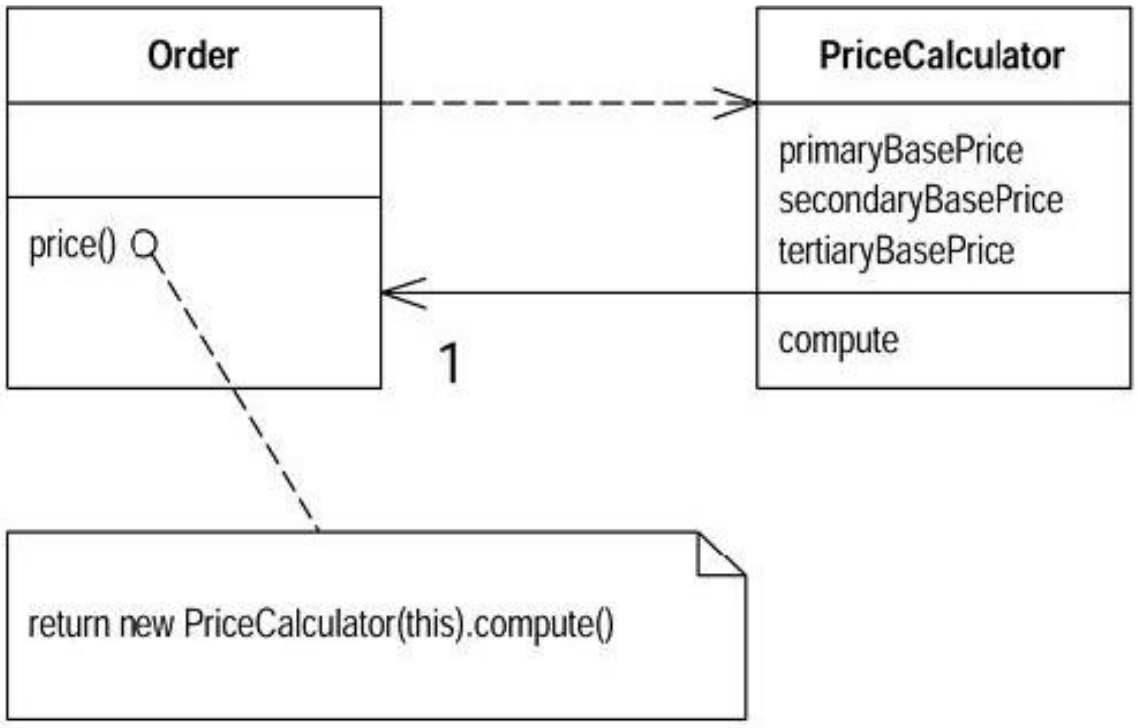
```
int discount (int inputVal, int quantity, int yearToDate) {
  if (inputVal > 50) inputVal -= 2;
  ...
}
```

After refactoring

```
int discount (int inputVal, int quantity, int yearToDate) {
  int result = inputVal;
  if (inputVal > 50) result -= 2;
  ...
}
```

Replace method with method object

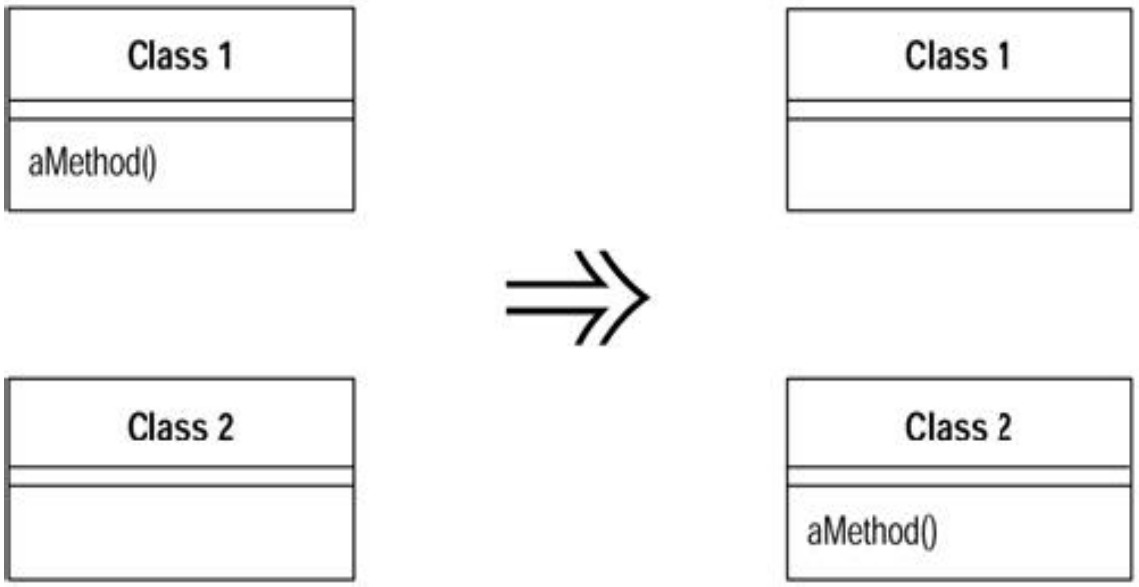
- You have a long method that uses local variables in such a way that you cannot apply Extract Method. Turn the method into its own object so that all the local variables become fields on that object. You can then decompose the method into other methods on the same object.



Move method

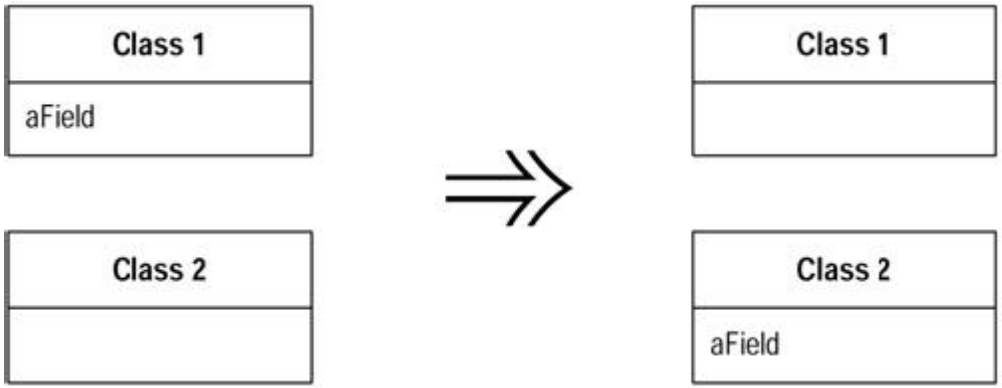
- A method is, or will be, using or used by more features of another class than the class on which it is defined.

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.



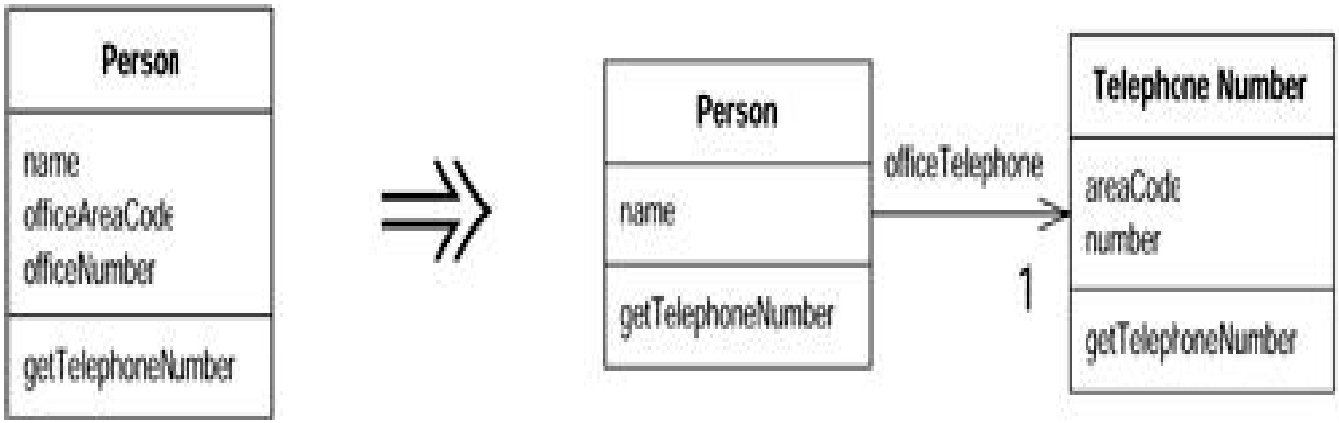
Move field

- 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.*



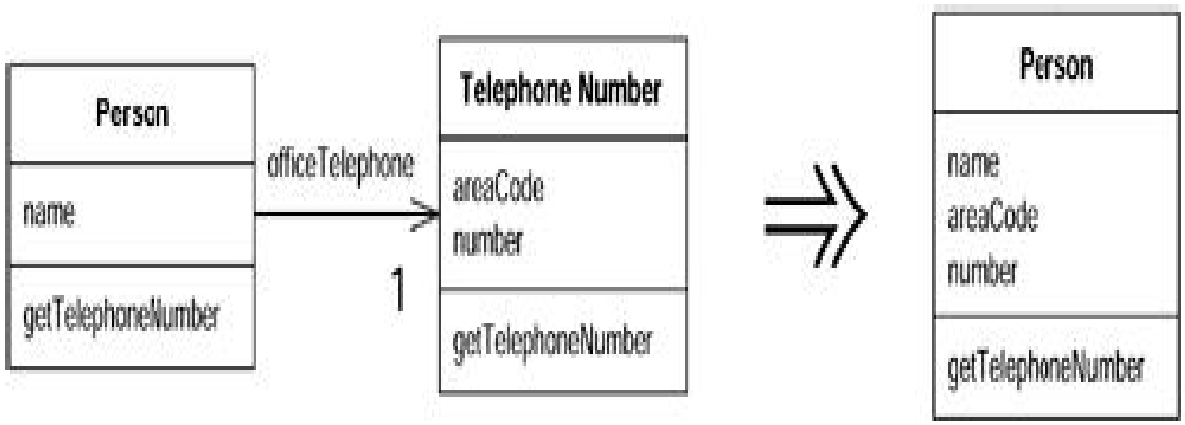
Extract class

- 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.



Inline class

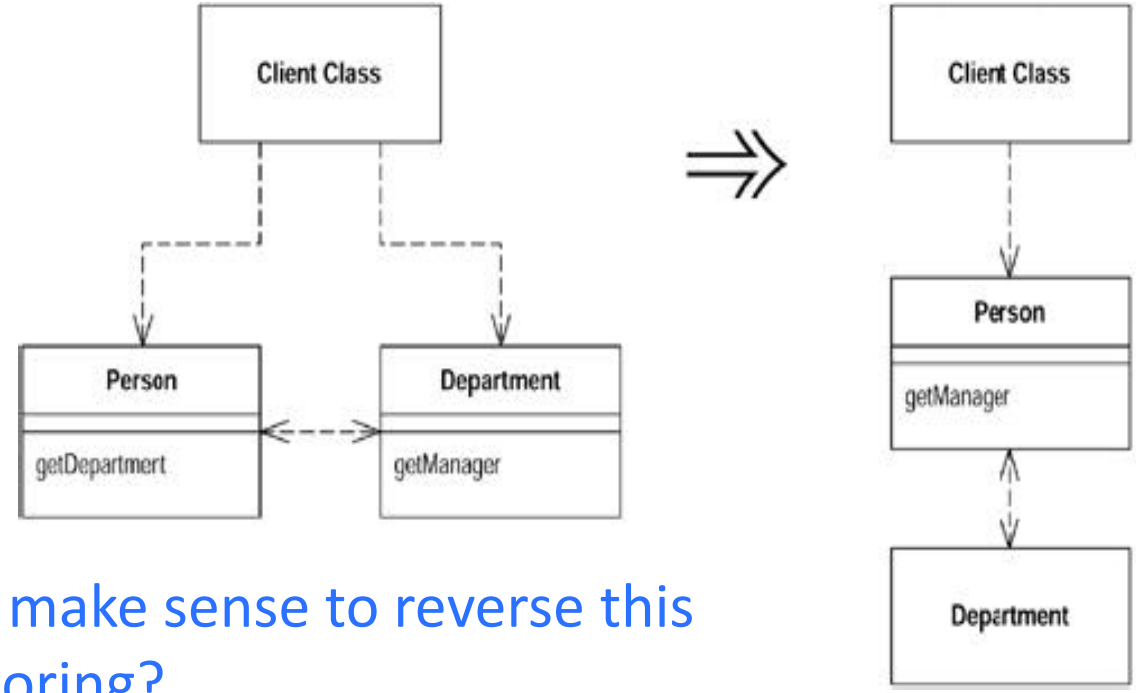
- A class isn't doing very much.Move all its features into another class and delete it.
- *Inline Class* is the reverse of Extract Class. I use *Inline Class* if a class is no longer pulling its weight and shouldn't be around any more. Often this is the result of refactoring that moves other responsibilities out of the class so there is little left.



Hide delegate

- A client is calling a delegate class of an object.

Create methods on the server to hide the delegate.

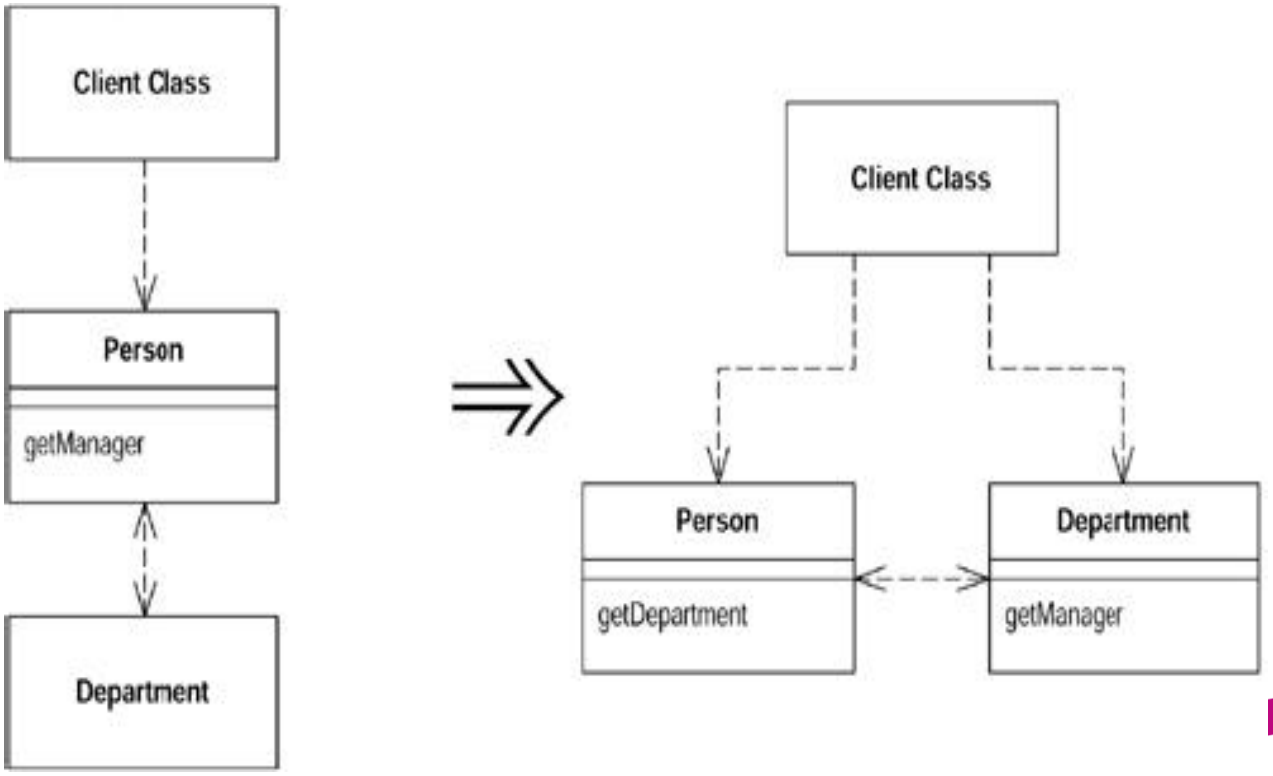


Does it make sense to reverse this refactoring?

lasaris

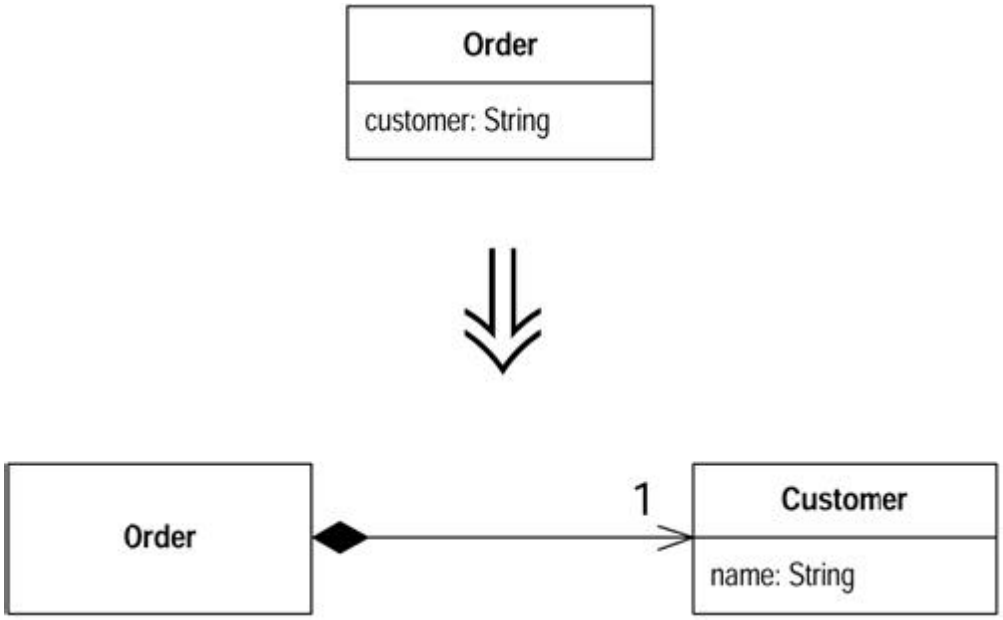
Remove the middle man

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



Replace data with value object

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



Replace array with object

- You have an array in which **certain elements mean different things**. *Replace the array with an object that has a field for each element.*

```
String[] row = new String[3];
row [0] = "Liverpool”;
row [1] = "15";
```

After Refactoring:

```
Performance row = new Performance();
row.setName("Liverpool");
row.setWins("15");
```

Replace magic number with a constant

- You have a literal number with a particular meaning. *Create a constant, name it after the meaning, and replace the number with it.*

```
double potentialEnergy(double mass, double height)
{
  return mass * 9.81 * height;
}

double potentialEnergy(double mass, double height)
{
  return mass * GRAVITATIONAL_CONSTANT * height;
}

static final double GRAVITATIONAL_CONSTANT = 9.81;
```

Encapsulate field

- There is a public field. *Make it private and provide accessors.*

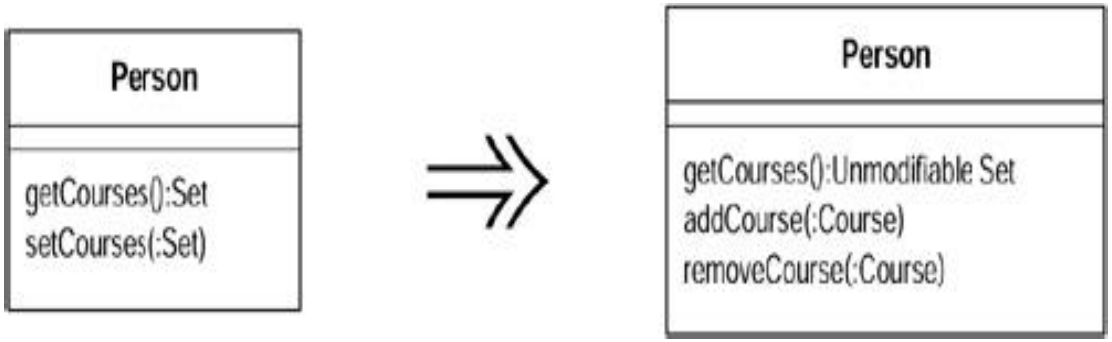
```
public String name;
```

After Refactoring:

```
private String name;
public String getName() {return name;}
public void setName(String arg) {name = arg;}
```

Encapsulate collection

- A method returns a collection. *Make it return a read-only view and provide add/remove methods.*



Motivation

Often a class contains a collection of instances. This collection might be an array, list, set, or vector. Such cases often have the usual getter and setter for the collection.

lasaris

Decompose conditional

- You have a complicated conditional (if-then-else) statement. *Extract methods from the condition, then part, and else parts.*

```
if (date.before (SUMMER_START) || date.after(SUMMER_END))
  charge = quantity * winterRate + winterServiceCharge;
else
  charge = quantity * summerRate;
```

After Refactoring:

```
if (notSummer(date))
  charge = winterCharge(quantity);
else
  charge = summerCharge (quantity);
```

Consolidate conditional expression

- You have a sequence of conditional tests with the same result. *Combine 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;
  ..
```

After Refactoring:

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


lasaris

Consolidate duplicate conditional fragments

- The same fragment of code is in all branches of a conditional expression. *Move it outside of the expression*

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

After Refactoring:

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


lasaris

Replace nested conditional with guard classes

- A method has conditional behavior that does not make clear the normal path of execution. *Use guard clauses for all the special cases*

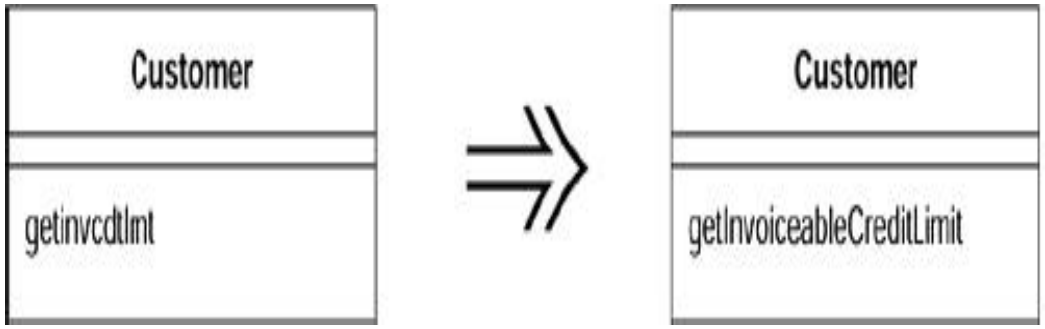
```
double getPayAmount() {
  double result;
  if (isDead) result = deadAmount();
  else
  {
    if (isSeparated) result = separatedAmount();
    else {
      if (isRetired) result = retiredAmount();
      else result = normalPayAmount();
    };
  }
  return result;
};
```

After Refactoring:

```
double getPayAmount() {
  if (isDead) return deadAmount();
  if (isSeparated) return separatedAmount();
  if (isRetired) return retiredAmount();
  return normalPayAmount();
};
```

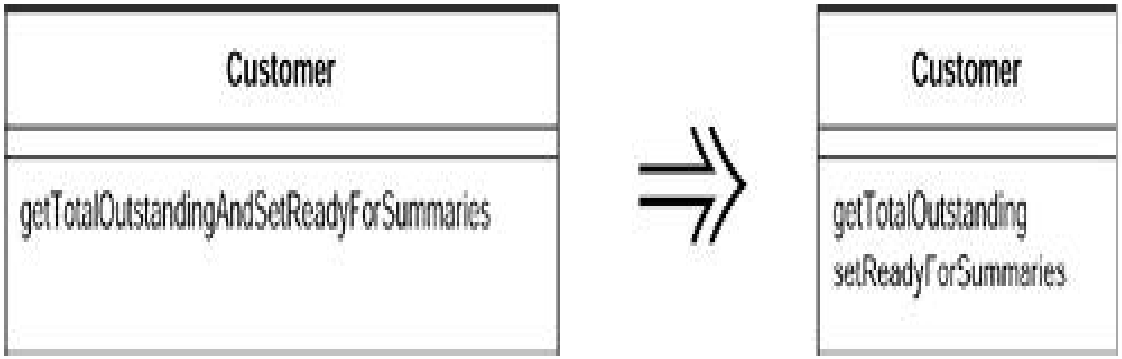
Rename method

- The name of a method does not reveal its purpose. *Change the name of the method*



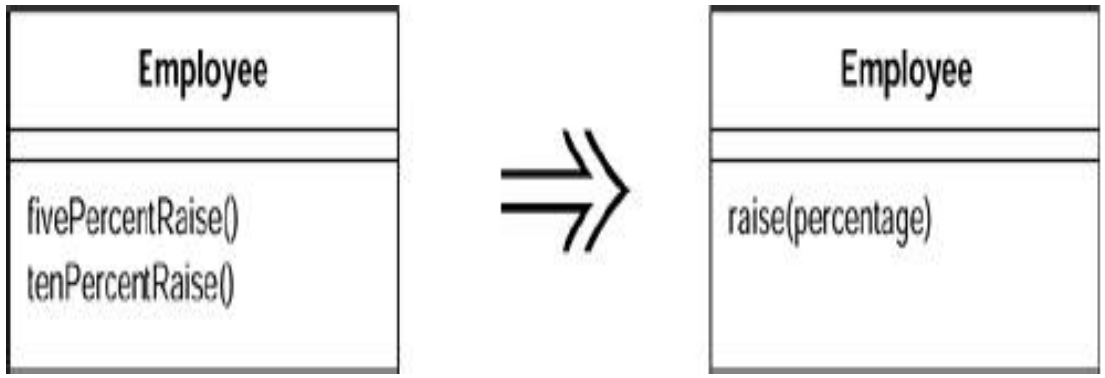
Separate query from modifier

- You have a method that returns a value but also changes the state of an object. *Create two methods, one for the query and one for the modification.*



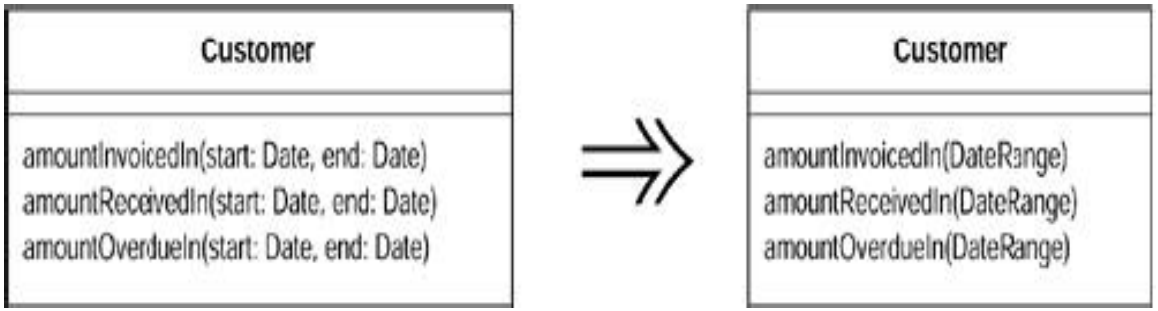
Parameterized method

- Several methods do similar things but with different values contained in the method body. *Create one method that uses a parameter for the different values.*



Introduce parameter to object

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



Remove setting methods

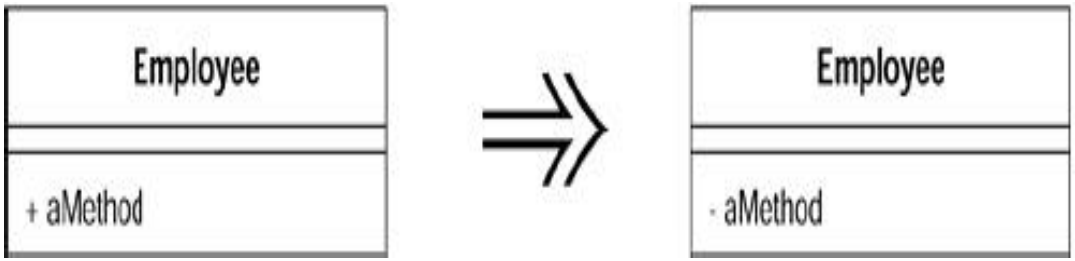
- A field should be set at creation time and never altered. *Remove any setting method for that field.*

- Providing a setting method indicates that a field may be changed. If you don't want that field to change once the object is created, then don't provide a setting method (and make the field final). That way your intention is clear and you often remove the very possibility that the field will change.

lasaris

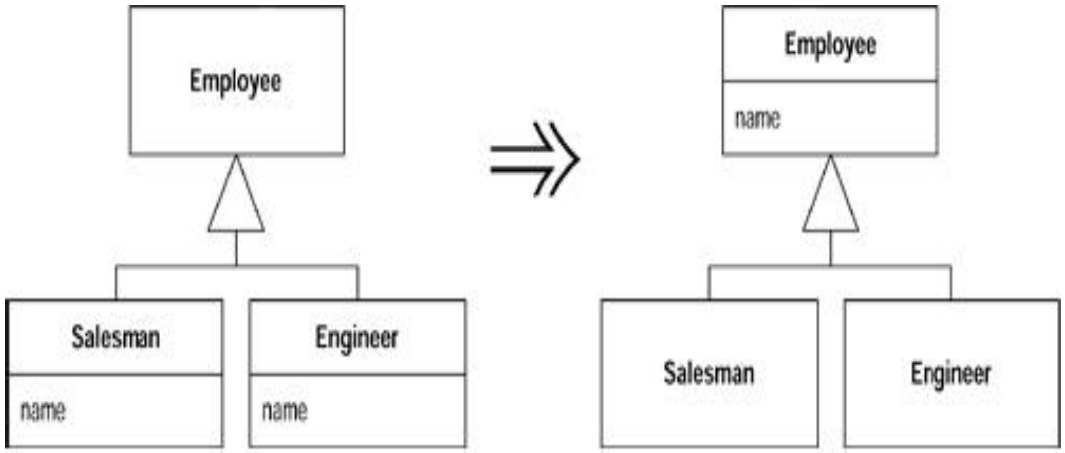
Hide method

- A method is not used by any other class. *Make the method private*



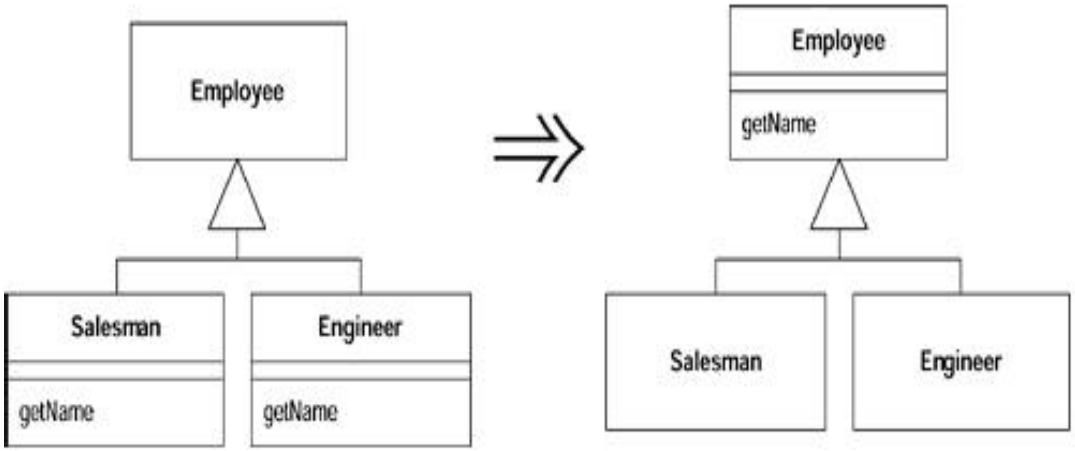
Pull up field

- Two subclasses have the same field. *Move the field to the superclass.*



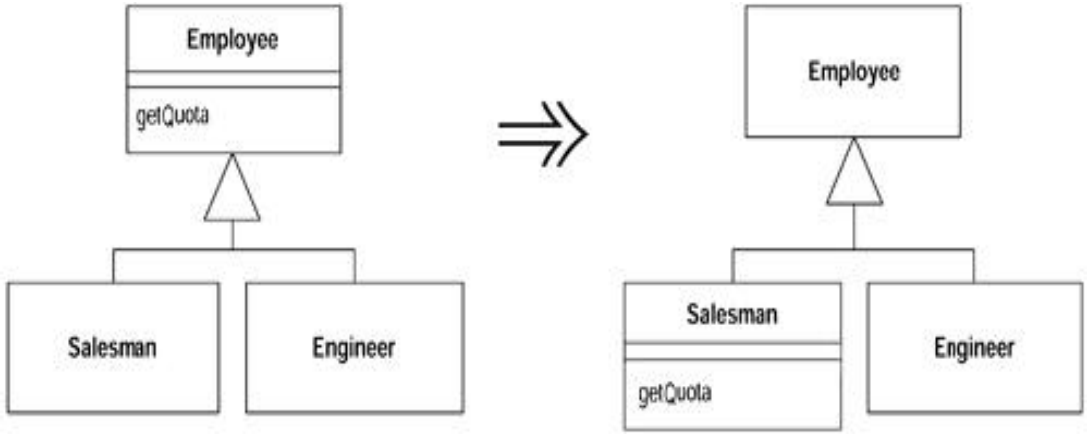
Pull up method

- Two subclasses have the same field. *Move the field to the superclass.*



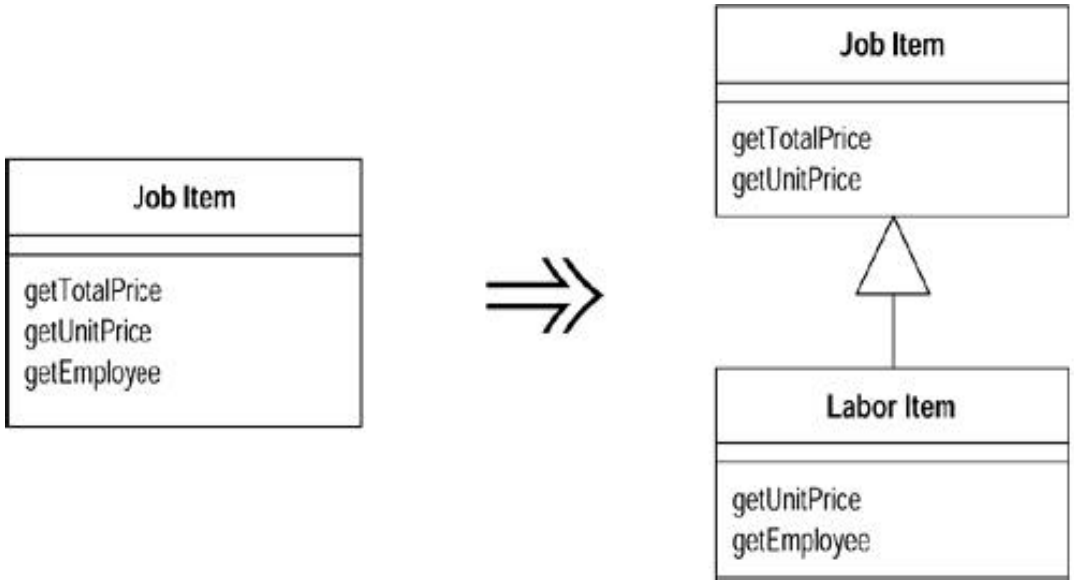
Push down method

- Behavior on a superclass is relevant only for some of its subclasses. Move it to those subclasses



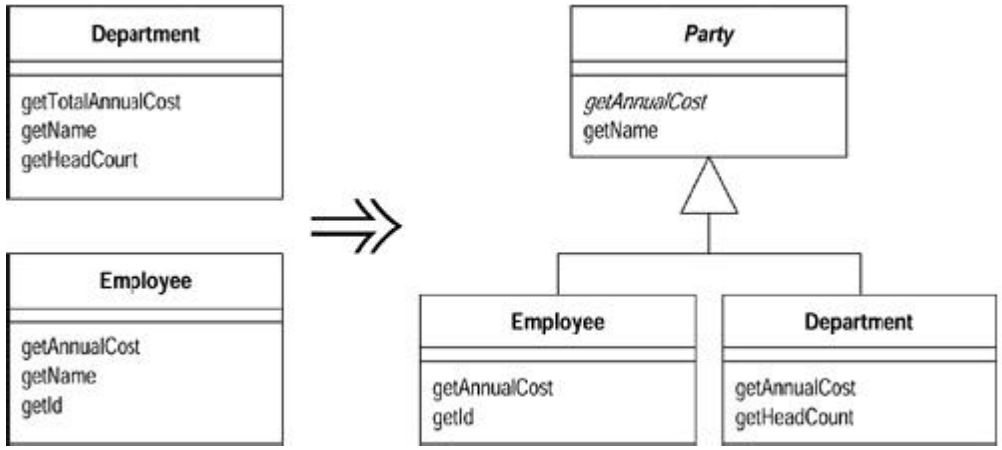
Extract subclass

- A class has features that are used only in some instances. Create a subclass for that subset of features



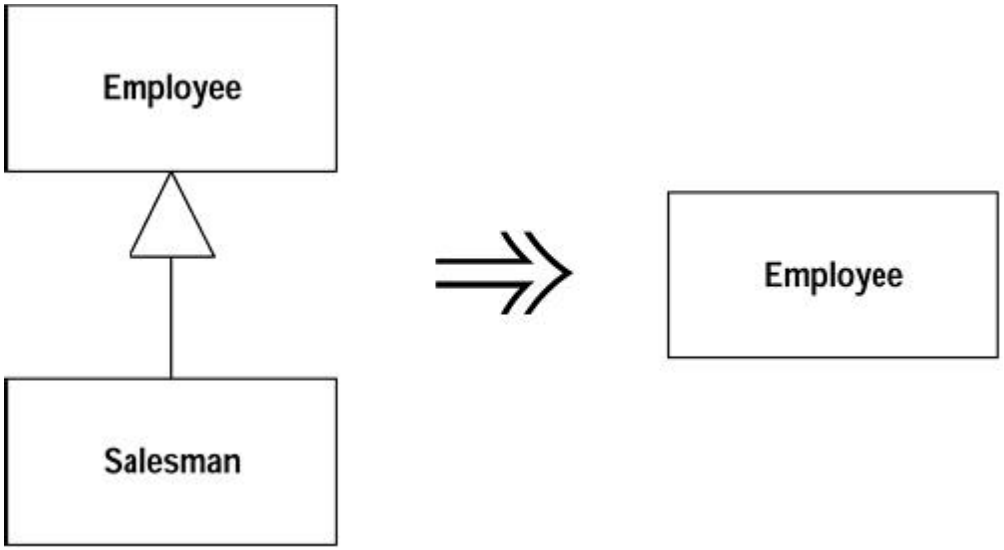
Extract superclass

- You have two classes with similar features. *Create a superclass and move the common features to the superclass.*



Collapse hierarchy

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



Refactoring and performance

- To make the software easier to understand, you often make changes that will cause the program to run more slowly.
- However, the calling overhead is very small and pays off!

Takeaways

- It always pays off to make your code **easier to understand**.
- Write self-documenting code, use **naming** (not comments) to express **code's intent**.
- Understand **S.O.L.I.D. principles** and do not be afraid to restructure your code into **MANY small classes** and methods.
- **Refactoring and testing** make you **FASTER**!
  - **Bad code smells** will navigate you.

thanks for listening

Barbora Bühnová, FI MU Brno
contact me
buhnova@fi.muni.cz
www.fi.muni.cz/~buhnova

References

- [1] V. Dusch: Stop wasting time through clean code
- [2] Ondřej Krajíček: PV260 lecture in Spring 2015
- [3] Adnan Masood: Refactoring Code to a Solid Foundation
- [4] Los Techies: Pablo's SOLID Software Development
- [5] Martin Osovský: PV260 lecture in Spring 2015
- [6] Igor Crvenov: Refactoring Tips by Marin Fowler

lasaris