Refactoring

Restructuring existing computer code

with C# Examples

Guilan University Computer Group
Student: Aryan Ebrahimpour

List of Content

List of Content

What is Refactoring?

What is refactoring?

- Refactoring is a controllable process of improving code without creating new functionality. It transforms a mess into clean code and simple design.
  - improves nonfunctional attributes of the software
- Advantages include:
  - improved code readability
  - reduced complexity
- These can improve:
  - source-code maintainability
  - create a more expressive internal architecture or object model
  - improve extensibility
- Typically, refactoring applies a series of standardized basic micro-refactorings, each of which is (usually) a tiny change in a computer program's source code

What is refactoring?

```
 9
10  public class Person
11  {
12      public string LastName { get; set; }
13      public string FirstName { get; set; }
14      public int Age { get; set; }
15
16      Person(string lastName, string firstName, int age)
17      {
18          this.LastName = LastName;
19          this.FirstName = firstName;
20          this.Age = age;
21      }
22  }
23
```

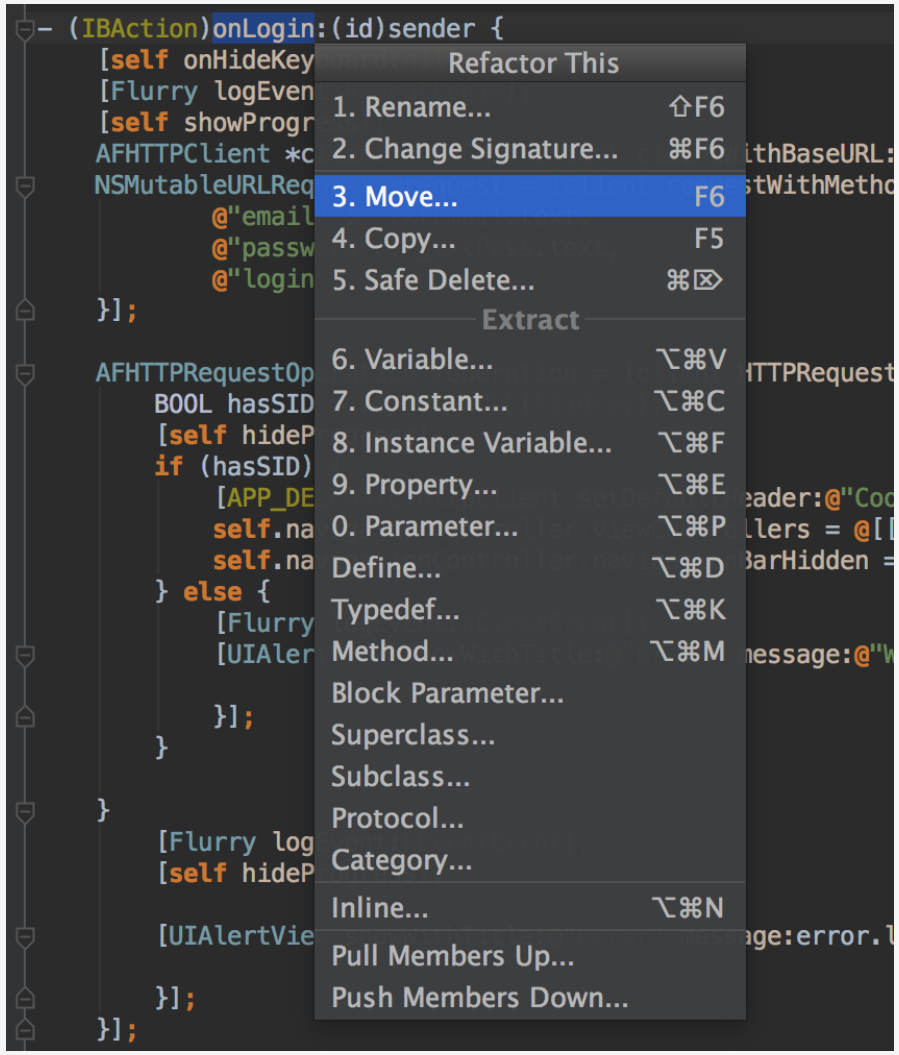
What is refactoring?

```
 9
10  public class Person
11  {
12      public string FirstName { get; set; }
13      public string LastName { get; set; }
14      public int Age { get; set; }
15
16      Person(string firstName, string lastName, int age)
17      {
18          this.FirstName = firstName;
19          this.LastName = LastName;
20          this.Age = age;
21      }
22  }
23
```

Refactoring Tools

Refactoring Tools (1 / 9)

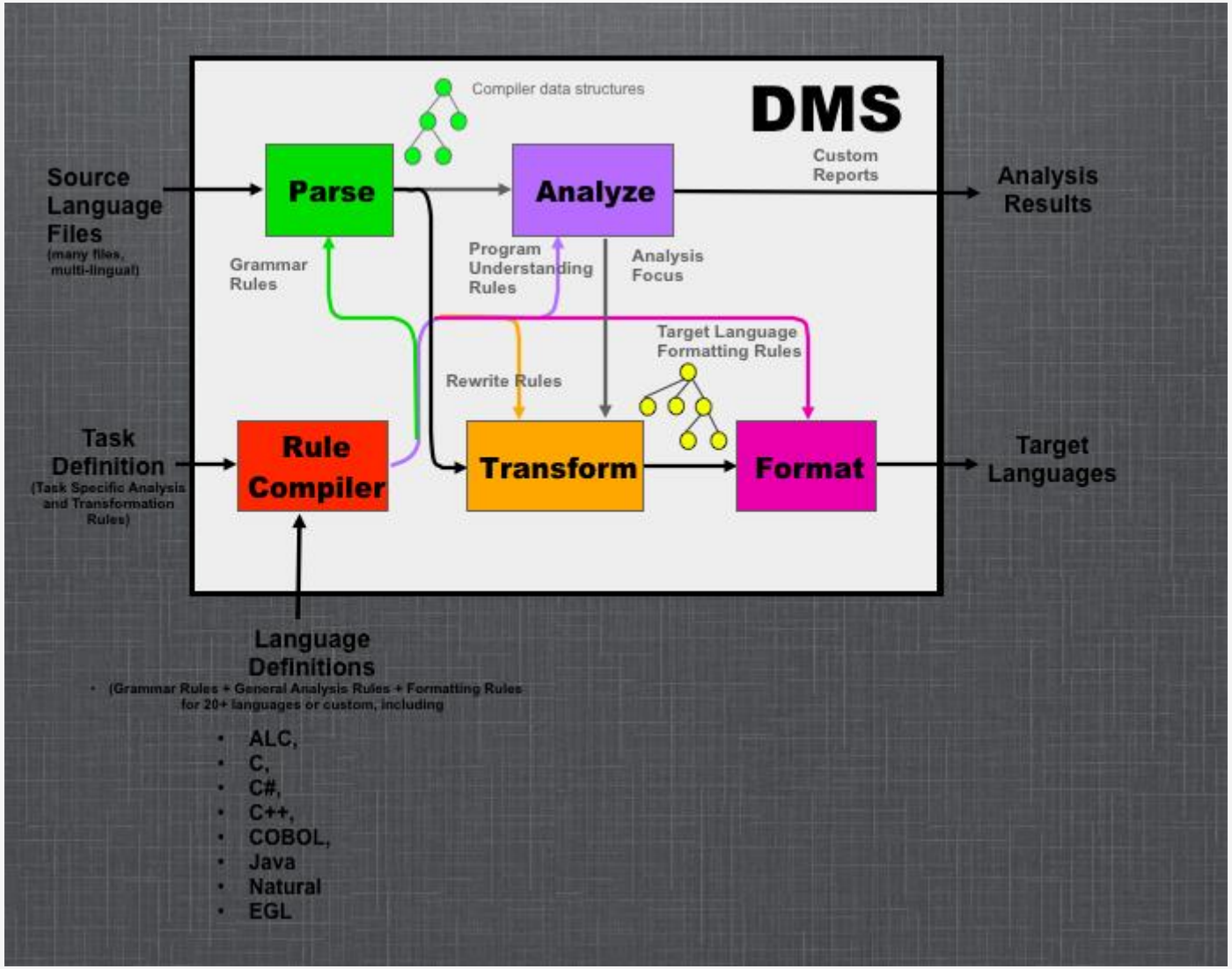
JetBrains AppCode for Objective-C, C and C++



Refactoring Tools (2 / 9)

DMS Software Reengineering toolkit

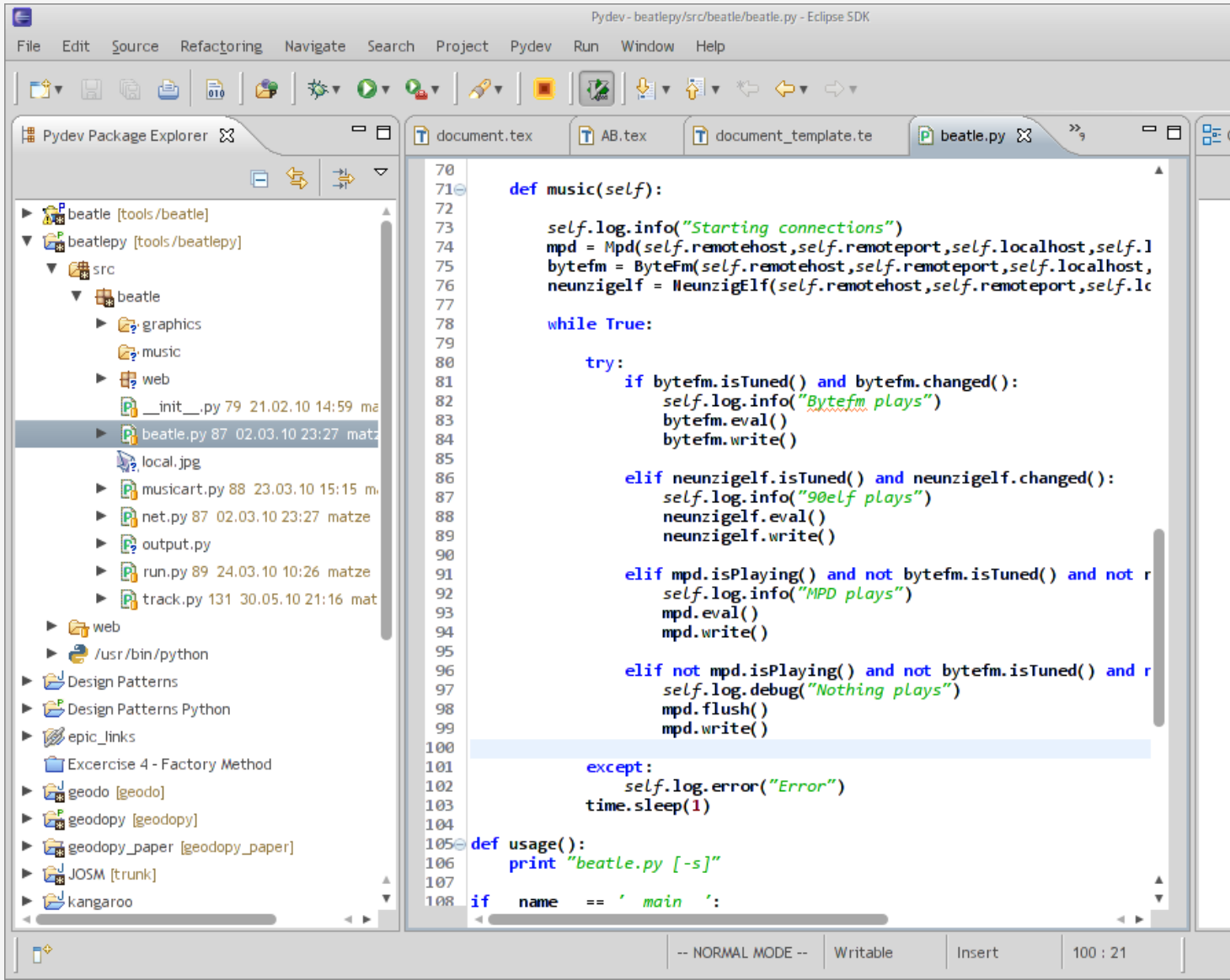
- C++
- C#
- Java
- COBOL
- SQL
- HTML
- Verilog
- …



Refactoring Tools (3 / 9)

Eclipse Based

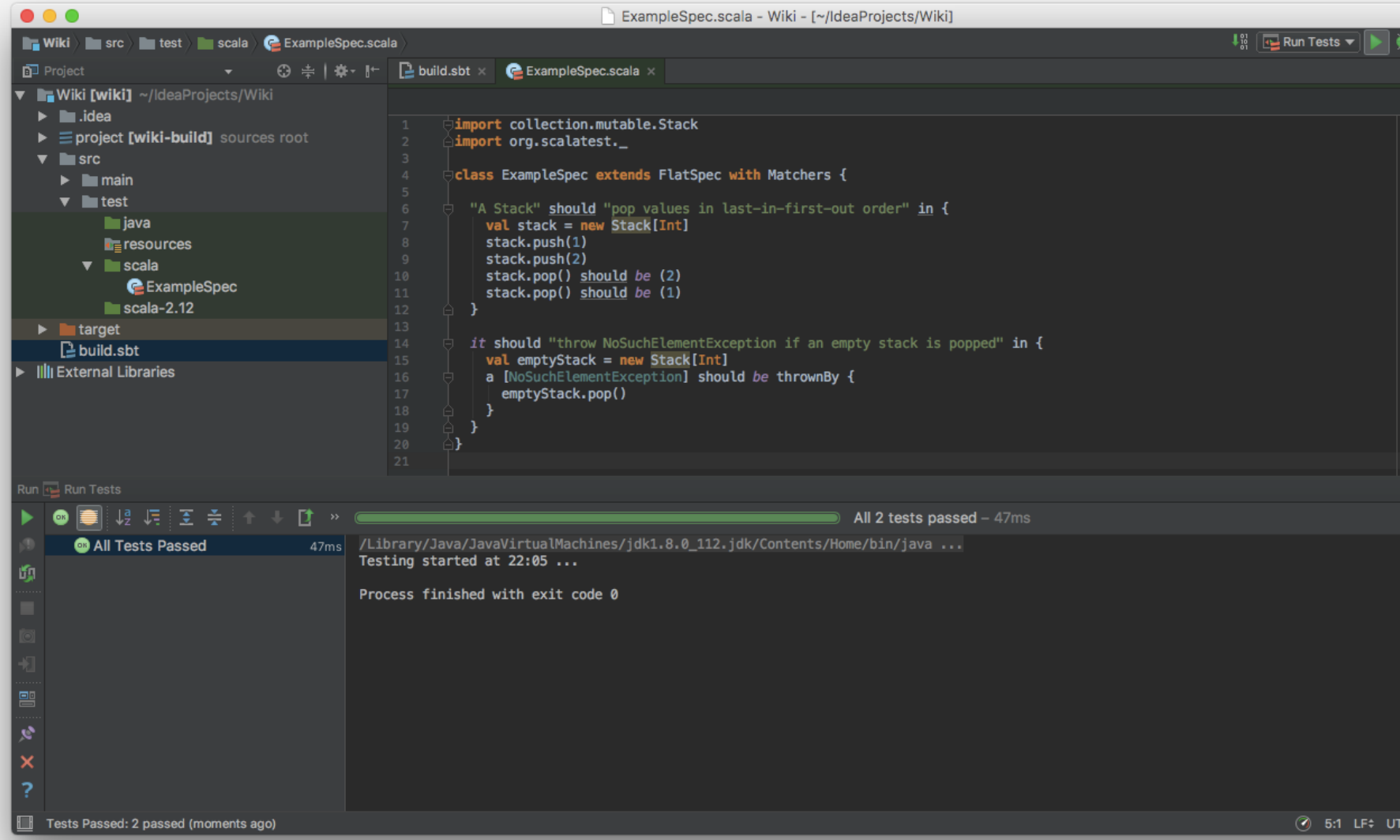
- Eclipse (Java, PHP, Ruby , ...)
- PyDev (for Python)
- Photran (a Fortran plugin for Eclipse)



Refactoring Tools (4 / 9)

IntelliJ Based

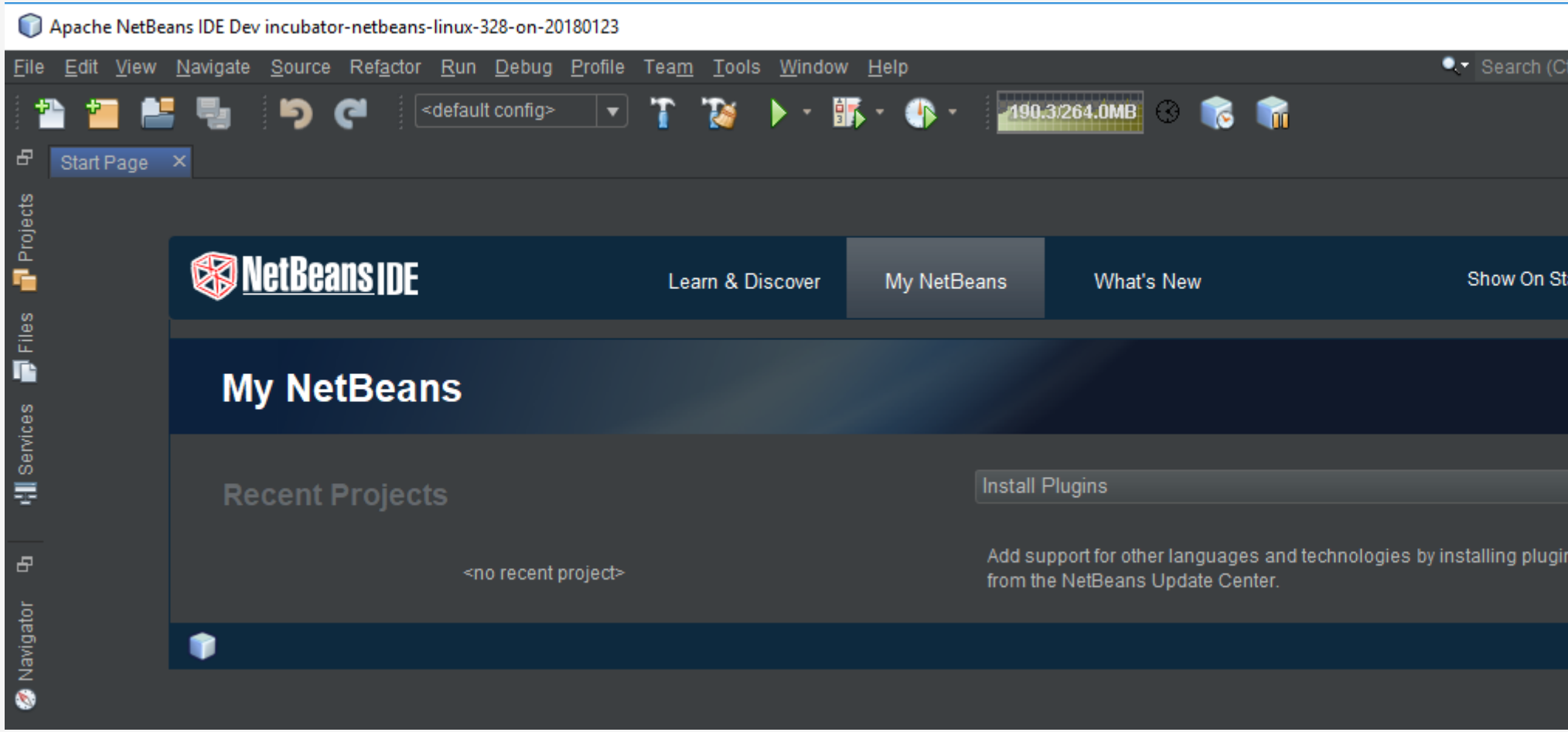
- IntelliJ IDEA (Java)
- PyCharm (Python)
- WebStorm (Js)
- Android Studio (Java)



Refactoring Tools (5 / 9)

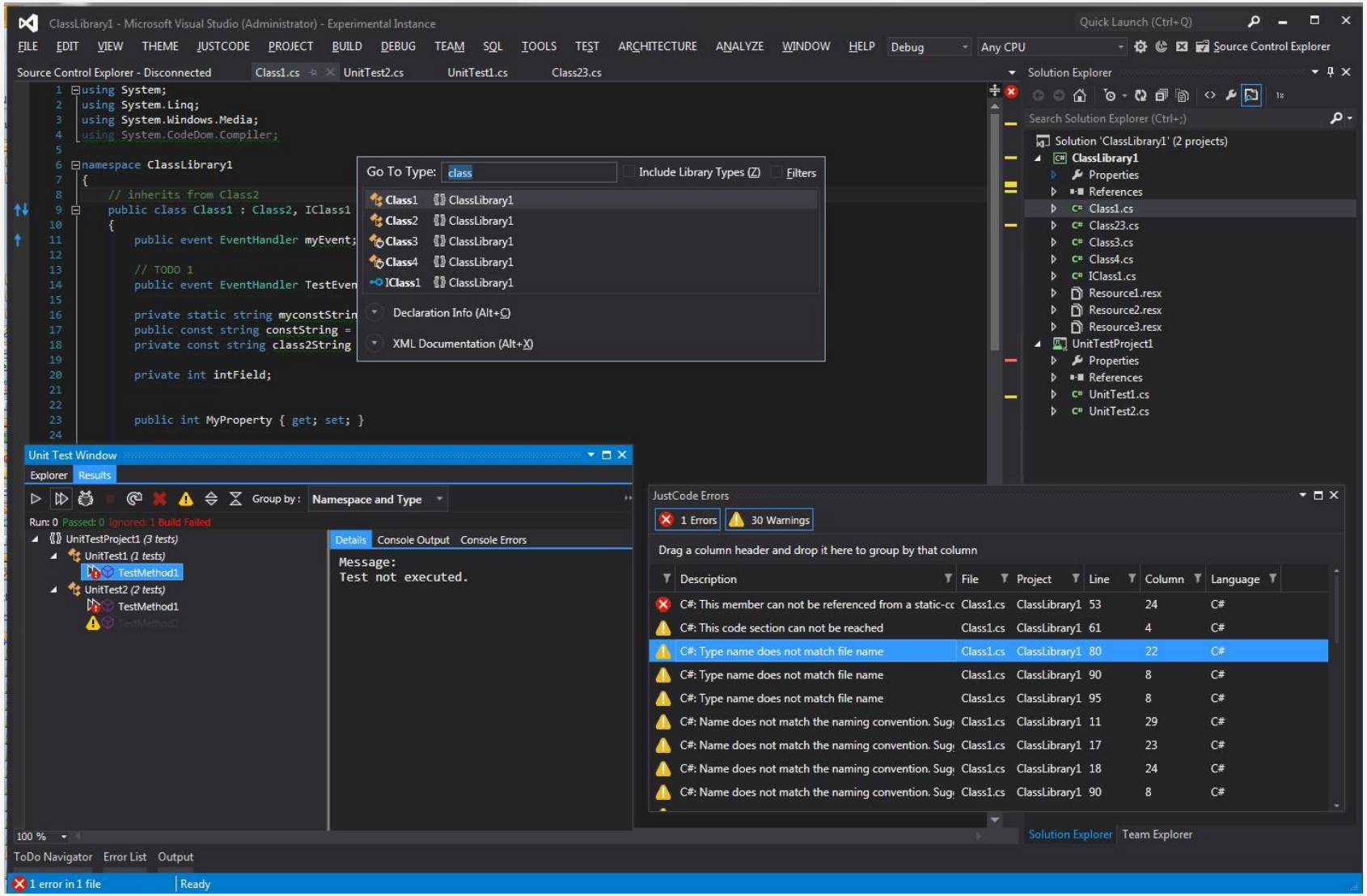
For Java

- JDeveloper
- NetBeans



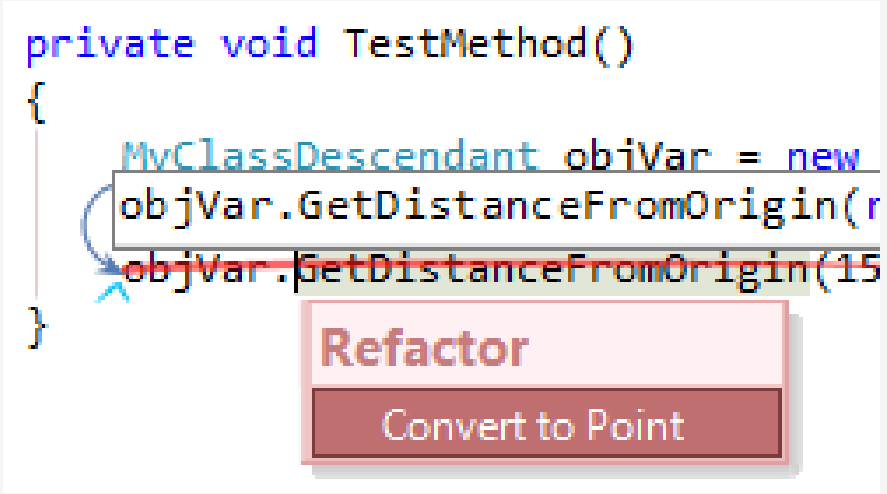
Refactoring Tools (6 / 9)

Visual Studio (for .NET and C++)



Refactoring Tools (7 / 9)

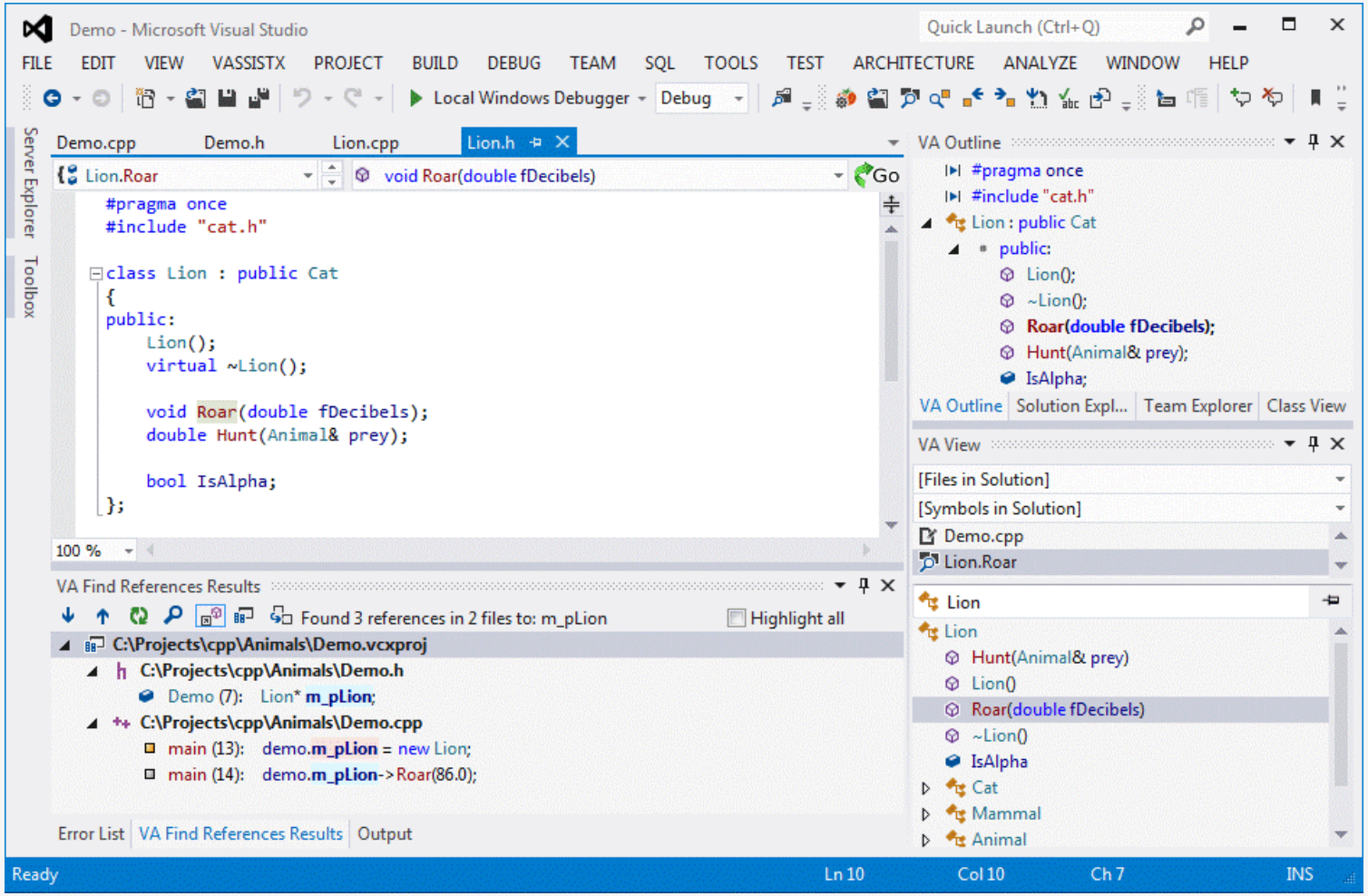
CodeRush (add-on for Visual Studio)



Refactoring Tools (8 / 9)

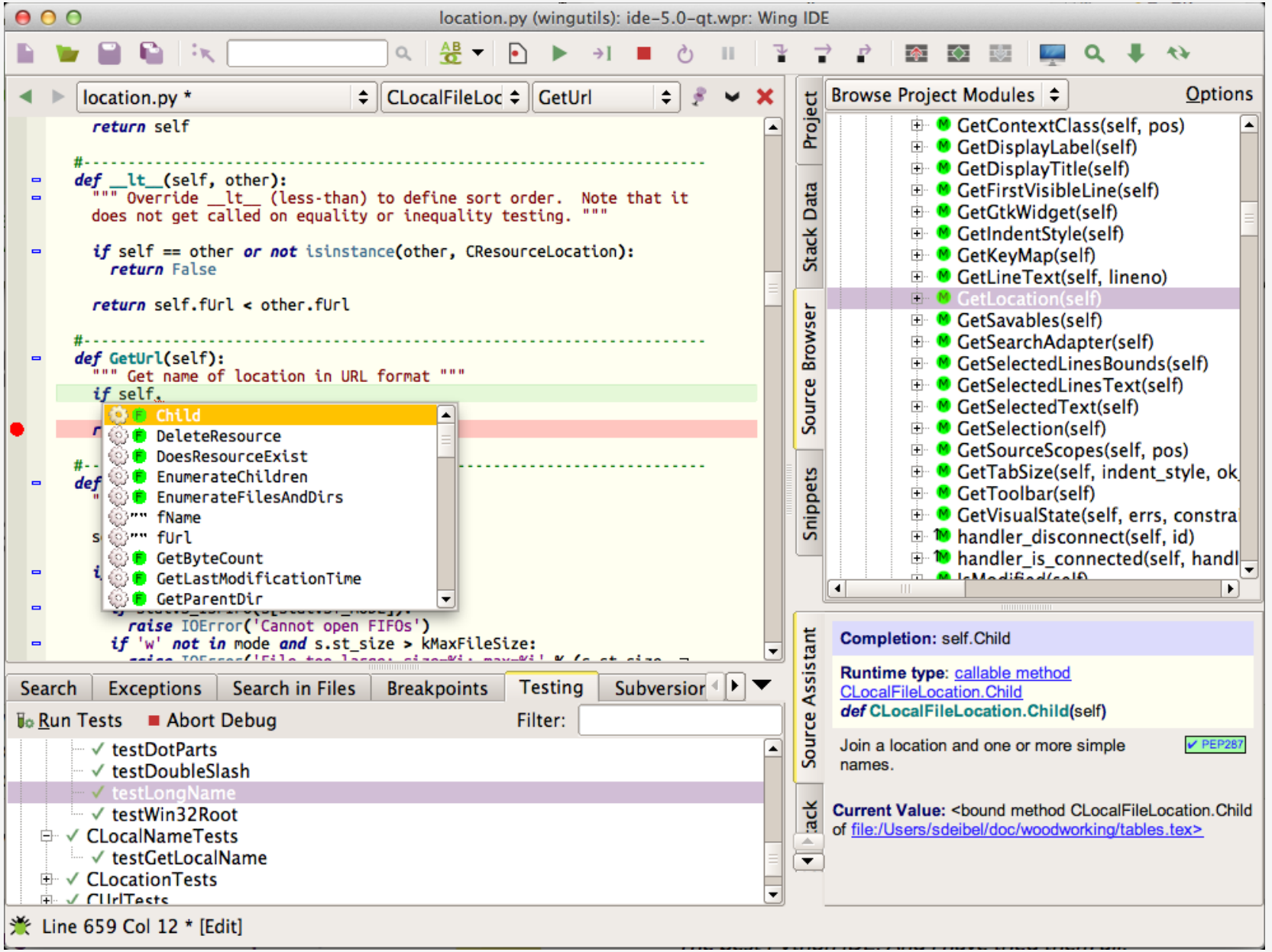
Visual Assist

add-on for Visual Studio with refactoring support for C# and C++



Refactoring Tools (9 / 9)

WING IDE (for Python)



Clean Coding

What is clean coding?

- The main purpose of refactoring is to fight technical debt.
- It transforms a mess into clean code and simple design.
- But what is clean code, anyway? Here are some of its features.

What is clean coding?

1- Clean code is obvious for other programmers.

And I'm not talking about super sophisticated algorithms. Poor variable naming, bloated classes and methods, magic numbers -you name it- all of that makes code sloppy and difficult to grasp.

What is clean coding?

2- Clean code doesn't contain duplication.

Each time you have to make a change in a duplicate code, you have to remember to make the same change to every instance. This increases the cognitive load and slows down the progress.

What is clean coding?

3- Clean code contains a minimal number of classes and other moving parts.

- Less code is less stuff to keep in your head.
- Less code is less maintenance.
- Less code is fewer bugs.
- Code is liability, keep it short and simple.

What is clean coding?

4- Clean code passes all tests.

You know your code is dirty when only 95% of your tests passed.

You know you're screwed when you test coverage is 0%.

What is clean coding?

5- Clean code is easier and cheaper to maintain!

Technical Debt

Technical debt

- Everyone does their best to write excellent code from scratch.
- There probably isn't a programmer out there who intentionally writes unclean code to the detriment of the project.
- But at what point does clean code become unclean?
- What is the term of “technical debt”?
  - If you get loan from the bank, you can make purchases faster.
  - You don't just pay off the principal, but also the additional interest on the loan.
  - Interest can exceed your total income, making full repayment impossible.
- Same thing happens with code:
  - You can temporarily speed up without writing tests for new features
  - but this will gradually slow your progress every day until you eventually pay off the debt by writing tests

Causes of technical debt

1- Business pressure.

Sometimes business circumstances might force you to roll out features before they are completely finished.

In this case, patches and crutches will appear in the code to hide the unfinished parts of the project.

Causes of technical debt

2- Lack of understanding of the consequences of technical debt.

Sometimes your employer might not understand that technical debt has "interest" insofar as it slows down the pace of development as debt accumulates.

This can make it too difficult to dedicate the team's time to refactoring because management does not see the value of it.

Causes of technical debt

3- Failing to combat the strict coherence of components.

This is when the project resembles a monolith rather than the product of individual modules.

In this case, any changes to one part of the project will affect others. Team development is made more difficult because it is difficult to isolate the work of individual members.

Causes of technical debt

4- Lack of tests.

The lack of immediate feedback encourages quick, but risky corrections (i.e. "crutches"), sometimes directly during the production phase.

The effects of this can be catastrophic. For example, an innocent hotfix might send a test email to the entire client database or delete the current customer data in the database.

Causes of technical debt

5- Lack of documentations.

This slows down the introduction of new people to the project and can grind development to a halt if key people leave the project.

Causes of technical debt

6- Lack of interaction between team members.

If the knowledge base is not distributed throughout the company, people will end up working with an outdated understanding of processes and information about the project.

This situation can be exacerbated when junior developers are incorrectly trained by their mentors.

Causes of technical debt

7- Long-term simultaneous development in several branches.

This can lead to the accumulation of technical debt, which is then increased when changes are merged.

The more changes made in isolation, the greater the total technical debt.

Causes of technical debt

8- Delayed refactoring.

The project's requirements are constantly changing and at some point it may become obvious that parts of the code are obsolete, have become cumbersome, and must be redesigned to meet new requirements.

On the other hand, the project's programmers are writing new code every day that works with the obsolete parts. Therefore, the longer refactoring is delayed, the more dependent code will have to be reworked in the future.

Causes of technical debt

9- Lack of compliance monitoring.

This happens when everyone working on the project writes code as they see fit (i.e. the same way they wrote the last project).

Causes of technical debt

10- Incompetence.

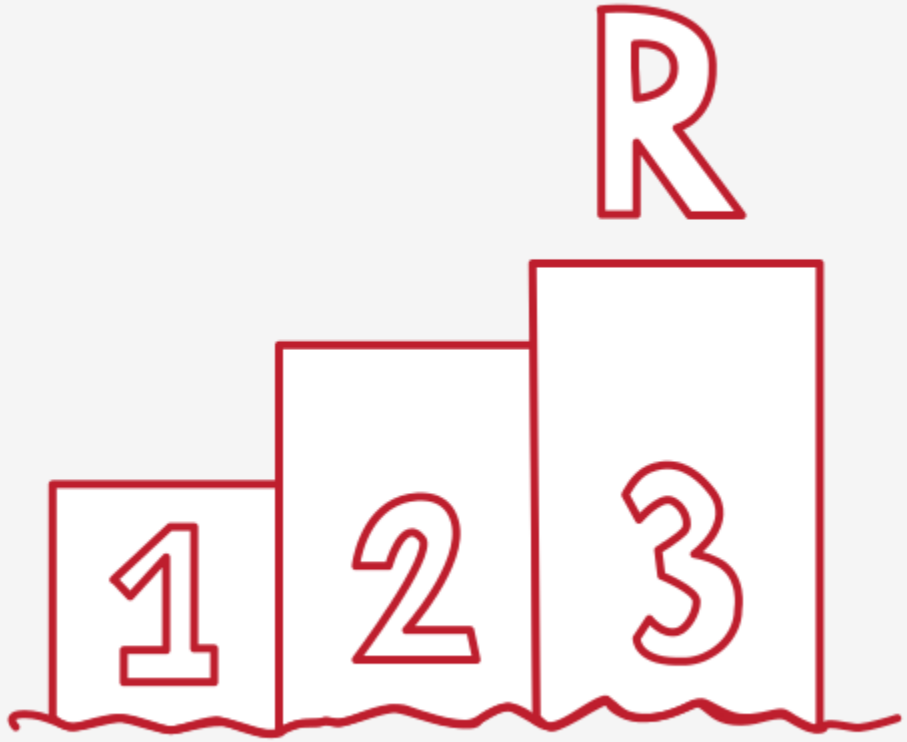
This is when the developer just doesn't know how to write decent code.

When to refactor?

When to refactor?

Rules Of Three

- When you are doing something for the first time, just get it done.
- When you are doing something similar for the second time, cringe at having to repeat but do the same thing anyway.
- When you are doing something for the third time, start refactoring.



When to refactor?

When adding a feature

- Refactoring helps you understand other people's code.
  - If you have to deal with someone else's dirty code, try to refactor it first.
  - Clean code is much easier to grasp.
  - You will improve it not only for yourself but also for those who use it after you.
- Refactoring makes it easier to add new features.
  - It's much easier to make changes in clean code.

When to refactor?

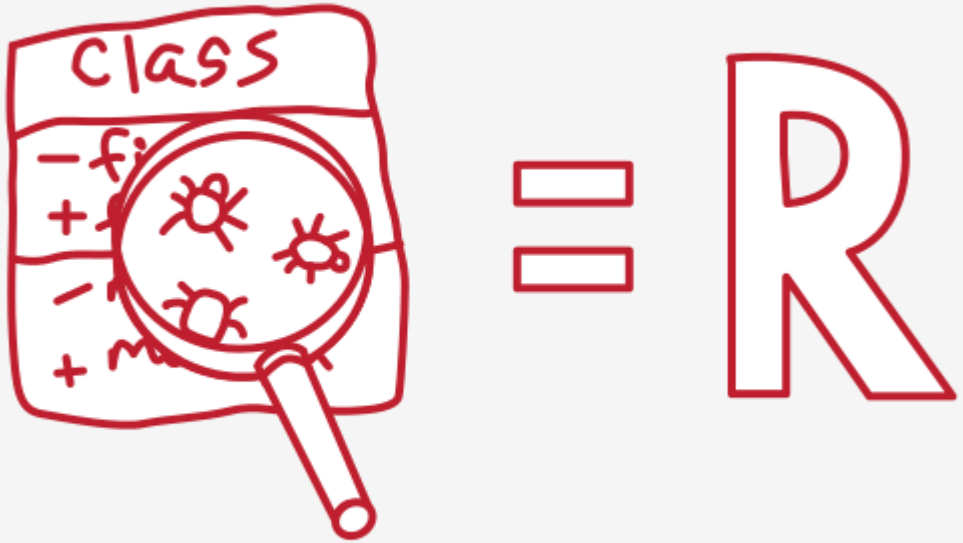
When fixing a bug

- Bugs in code behave just like those in real life: they live in the darkest, dirtiest places in the code. Clean your code and the errors will practically discover themselves.
- Managers appreciate proactive refactoring as it eliminates the need for special refactoring tasks later. Happy bosses make happy programmers!

When to refactor?

During code review

- The code review may be the last chance to tidy up the code before it becomes available to the public.
- It's best to perform such reviews in a pair with an author. This way you could fix simple problems quickly and gauge the time for fixing the more difficult ones.



How to refactor?

How to refactor?

- Set of small changes each of which makes the existing code slightly better
- Still leaving the program in working order.
- **Checklist of refactoring done right way:**

When to refactor?

1- The code should become cleaner

If the code remains just as unclean after refactoring... you've just wasted an hour of your life.

Try to figure out why this happened.

- This frequently happens when you move away from small changes to big changes.
  - In this case, It's very easy to lose your mind, especially if you have a time limit.
- It can also happen when working with extremely sloppy code.
  - Whatever you improve, the code as a whole still remains a disaster.

When to refactor?

2- New functionality should not be created during refactoring.

Do not mix refactoring and direct development of new features. Try to separate these processes at least within the confines of individual commits.

When to refactor?

3- All existing tests must pass after refactoring.

There are two cases when tests can break down after refactoring:

- You made a mistake when changing the code.
  - This is easy - just go ahead and correct the mistake.
- Your tests were too low-level (for example, testing private class methods).
  - In this case, the tests are to blame, and the only way to fix this is to refactor the tests themselves and write new, higher-level tests.
  - A great way to avoid this kind of situation is to write tests in the BDD style.

Code Smells

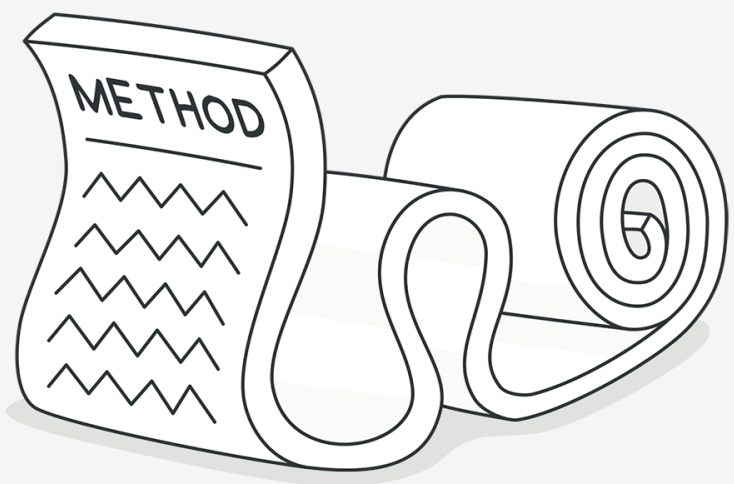
Code Smells

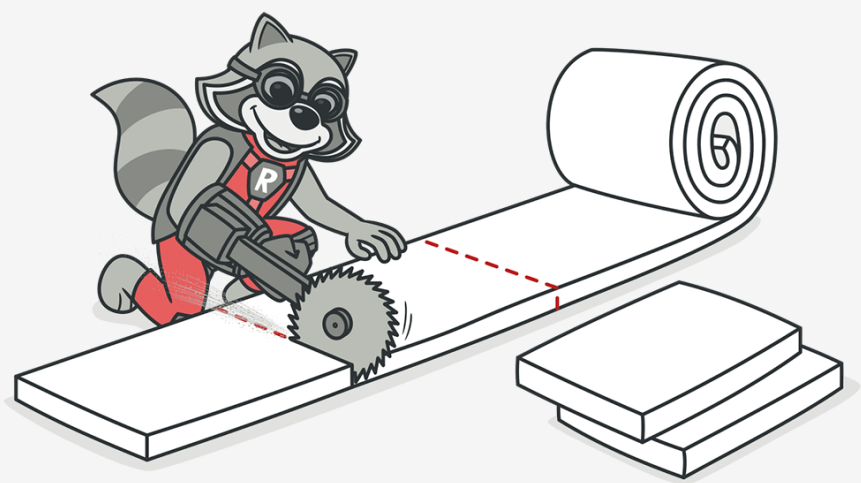
Bloaters

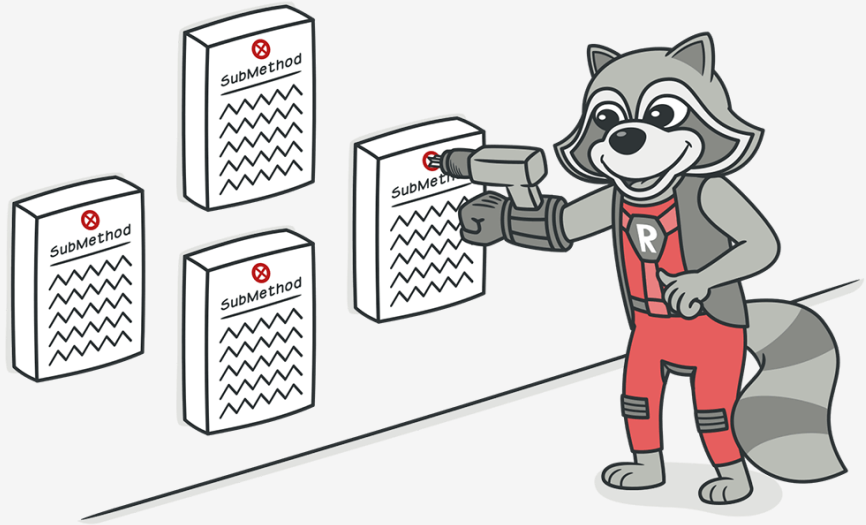
Bloaters are code, methods and classes that have increased to such gargantuan proportions that they are hard to work with. Usually these smells do not crop up right away, rather they accumulate over time as the program evolves (and especially when nobody makes an effort to eradicate them).

Long Methods

- A method contains too many lines of code.
- Generally, any method longer than ten lines should make you start asking questions.
- Use **Extract Method** (Even a single line, if it requires explanation)
- Performance: In almost all cases the impact of performance is so negligible that it's not even worth worrying about.







Extract Method (Before)

```
void PrintOwing()
{
    PrintBanner();

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

Extract Method (After)

```
void PrintOwing()
{
    PrintBanner();
    PrintDetails(GetOutstanding());
}

void PrintDetails(double outstanding)
{
    Console.WriteLine("name: " + name);
    Console.WriteLine("amount: " + outstanding);
}
```

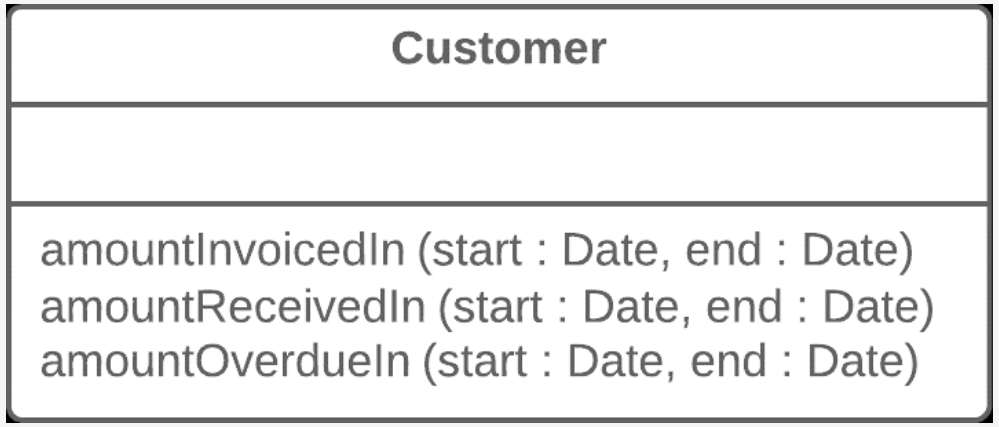
Replace Temp with Query (Before)

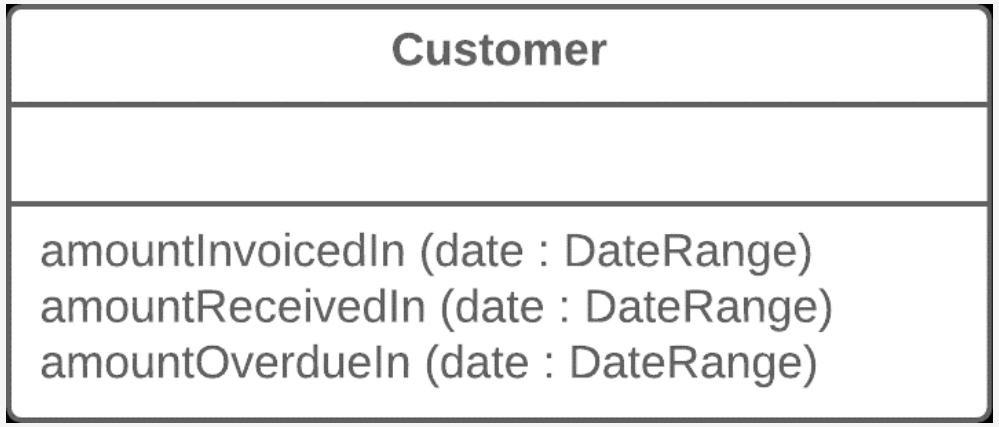
```
double CalculateTotal()
{
    double basePrice = quantity * itemPrice;

    if (basePrice > 1000)
    {
        return basePrice * 0.95;
    }
    else
    {
        return basePrice * 0.98;
    }
}
```

Replace Temp with Query (After)

```
double CalculateTotal()
{
    if (BasePrice > 1000)
    {
        return BasePrice * 0.95;
    }
    else
    {
        return BasePrice * 0.98;
    }
}

double BasePrice => quantity * itemPrice;
```

Introduce Parameter Object





Preserve Whole Object

```
public void Method(){

    int low = daysTempRange.GetLow();
    int high = daysTempRange.GetHigh();
    bool withinPlan = plan.WithinRange(low, high);

}
```

```
public void Method(){

    bool withinPlan = plan.WithinRange(daysTempRange);

}
```

Decompose Conditional (Before)

```
if (date < SummerStart || date > SummerEnd)
{
    charge = quantity * winterRate + winterServiceCharge;
}
else
{
    charge = quantity * summerRate;
}
```

Decompose Conditional (After)

```
if (NotSummer(date))
{
    charge = WinterCharge(quantity);
}
else
{
    charge = SummerCharge(quantity);
}
```

Replace Method with MethodObject (Before)

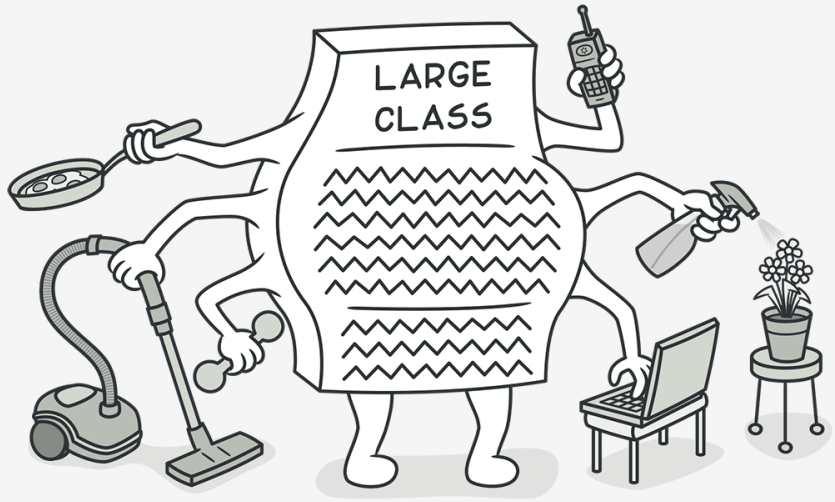
```
public class Order
{
    //...
    public double Price()
    {
        double primaryBasePrice;
        double secondaryBasePrice;
        double tertiaryBasePrice;
        // long computation.
        //...
    }
}
```

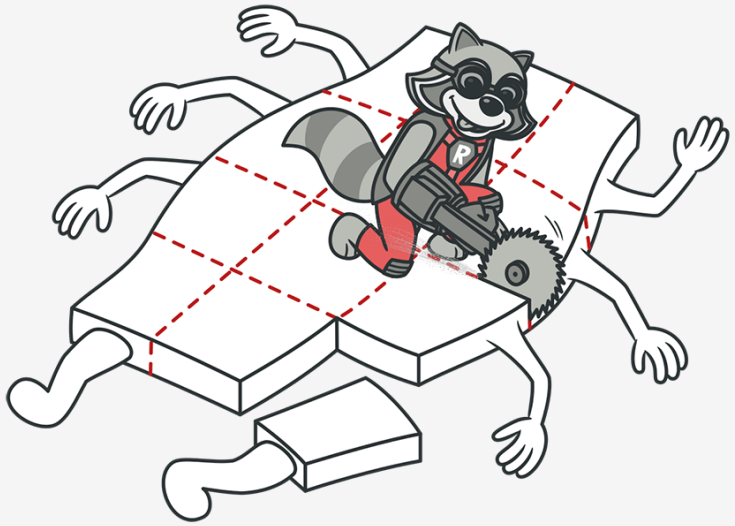
Code Smells

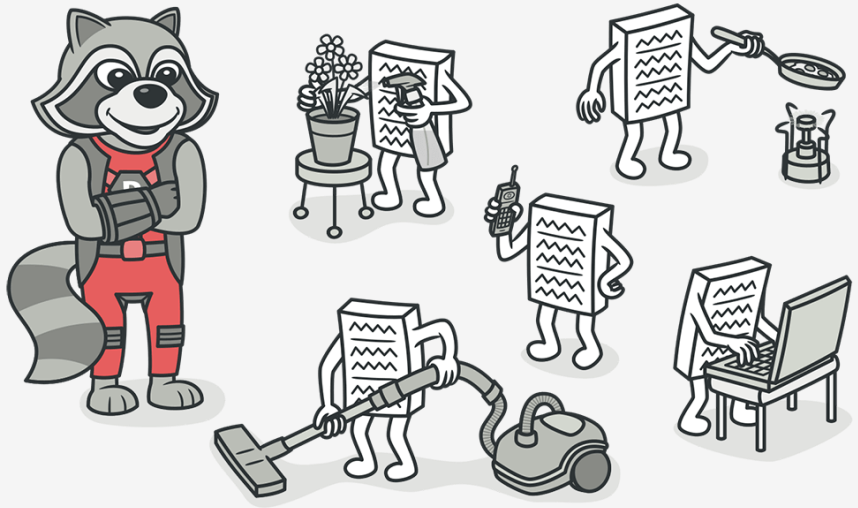
```
public class Order
{
    //...
    public double Price()
    {
        return new PriceCalculator(this).Compute();
    }
}

public class PriceCalculator
{
    private double primaryBasePrice;
    private double secondaryBasePrice;
    private double tertiaryBasePrice;

    public PriceCalculator(Order order)
    {
        // copy relevant information from order object.
        //...
    }

    public double Compute()
    {
        // long computation.
        //...

        return 0;
    }
}
```

Large Classes

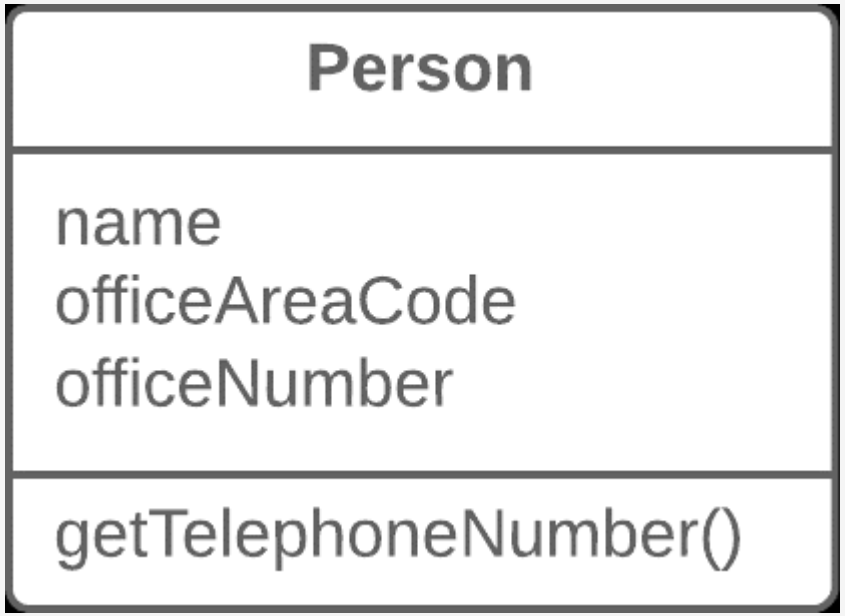
- A class contains many fields/methods/lines of code.
- Classes usually start small. But over time, they get bloated as the program grows.
- Like long methods, programmers usually find it mentally less taxing to place a new feature in an existing class than to create a new class for the feature.

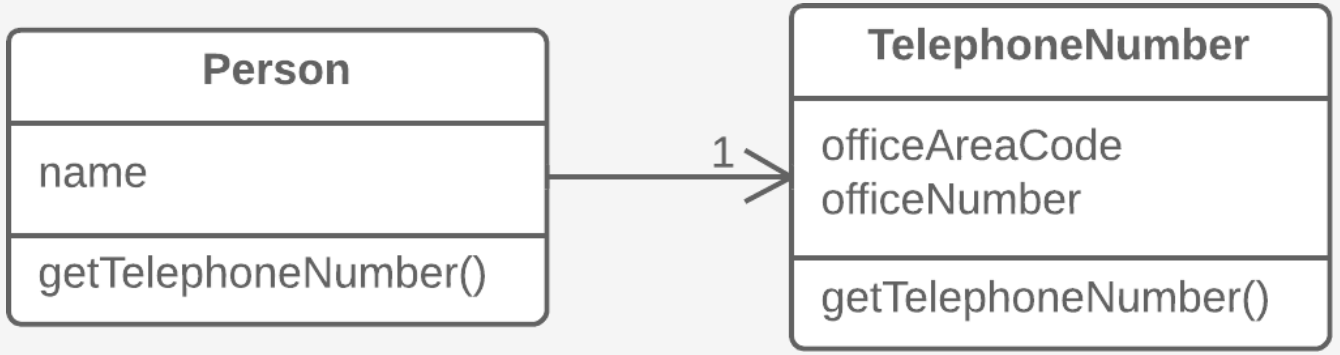




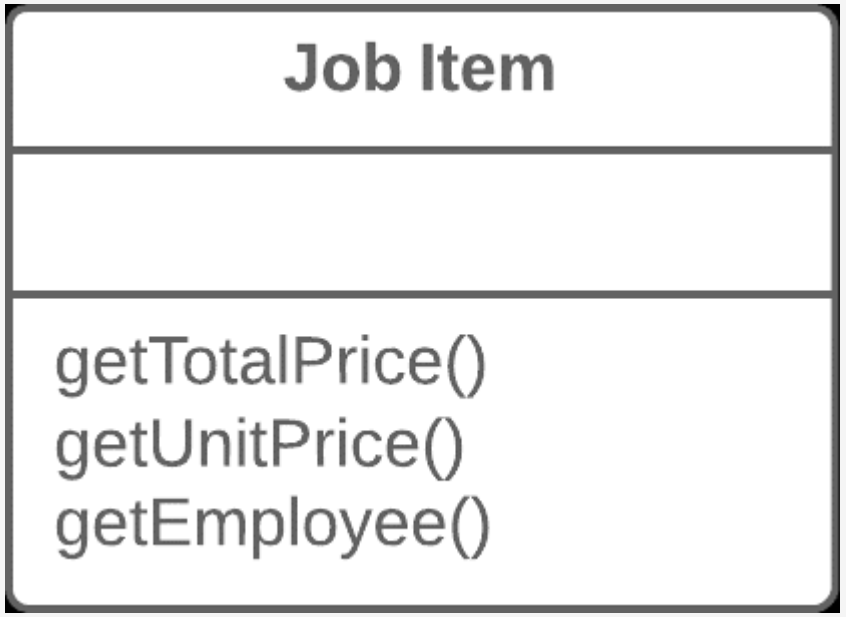


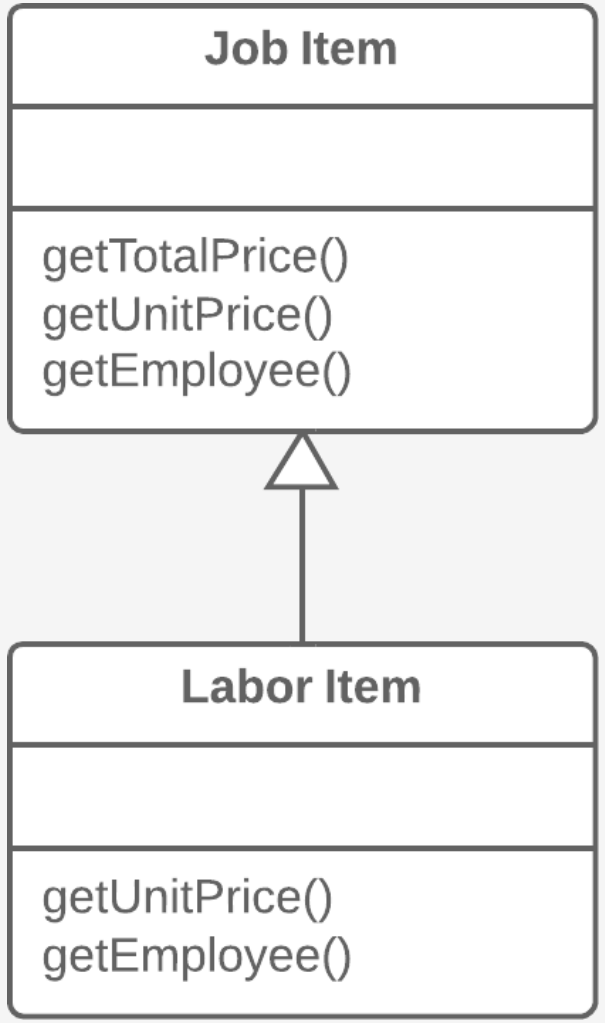
Extract Classes



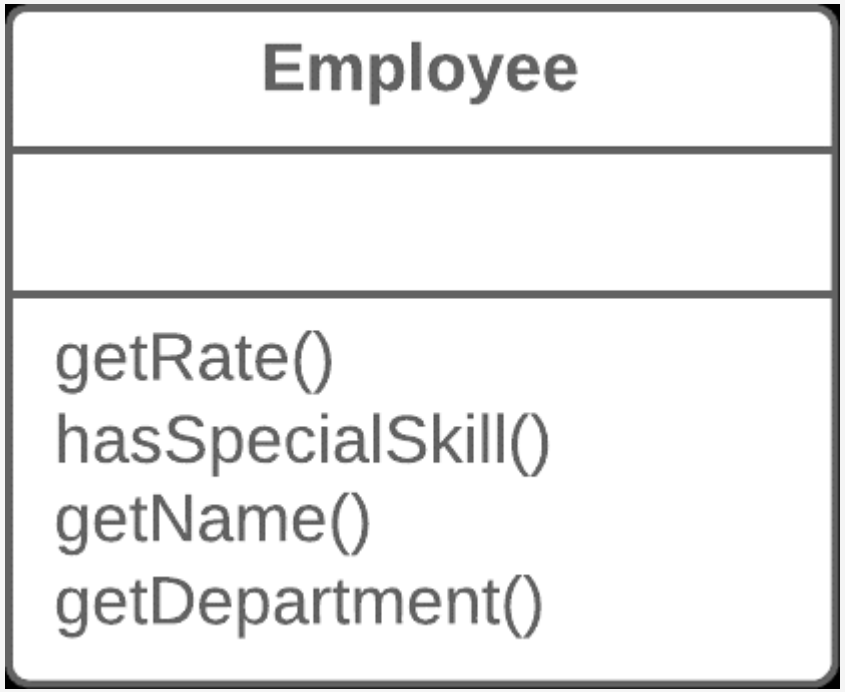


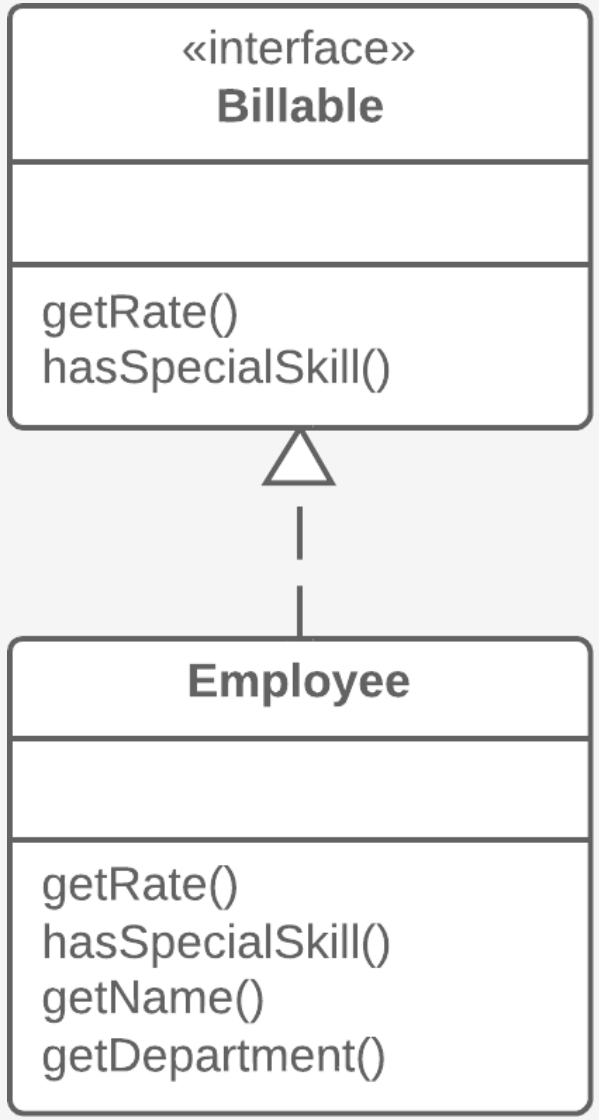
Extract Subclasses





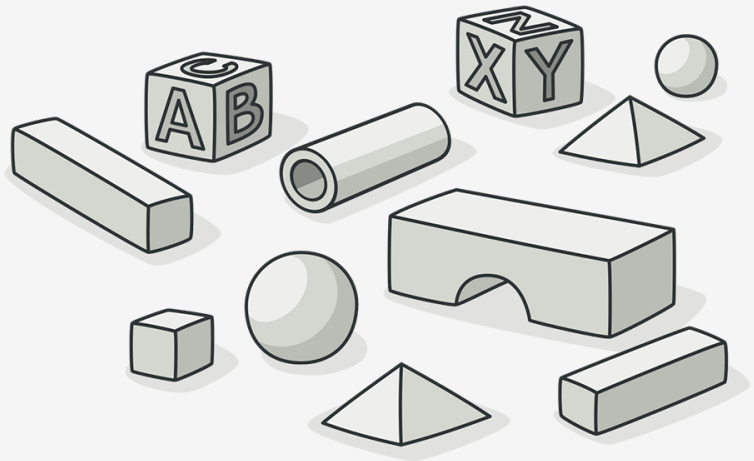
Extract Interface

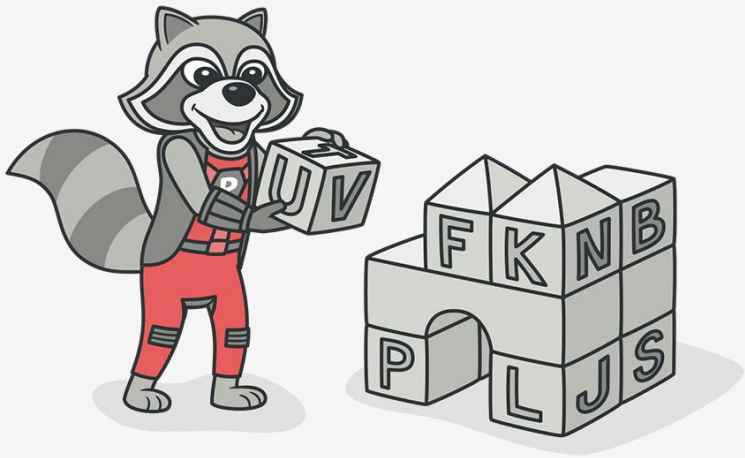


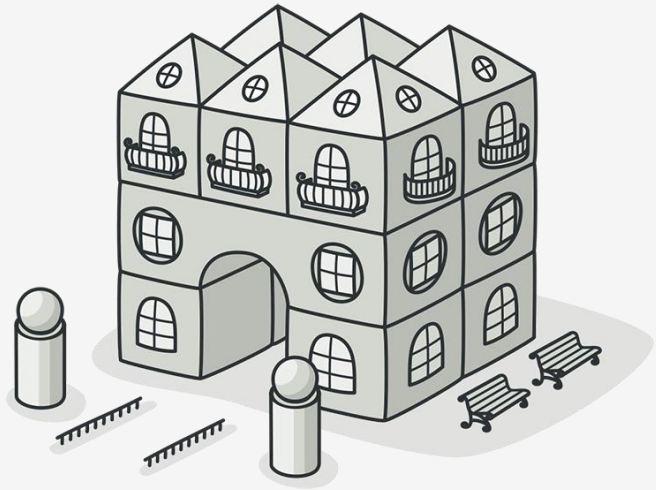


Primitive Obsession

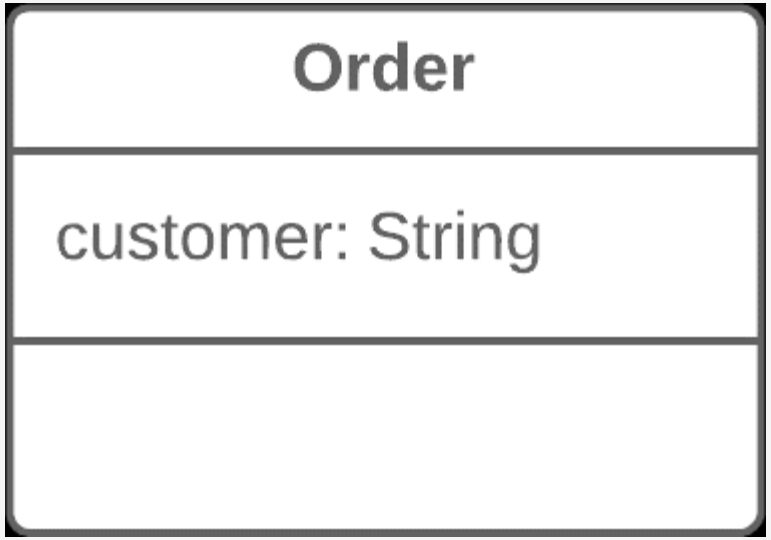
- Use of primitives instead of small objects for simple tasks (such as currency, ranges, special strings for phone numbers, etc.)
- Use of constants for coding information (such as a constant USER_ADMIN_ROLE = 1 for referring to users with administrator rights.)
- Use of string constants as field names for use in data arrays.
- In method parameters you can use **Introduce Parameter Object** and **Preserve Whole Object.**

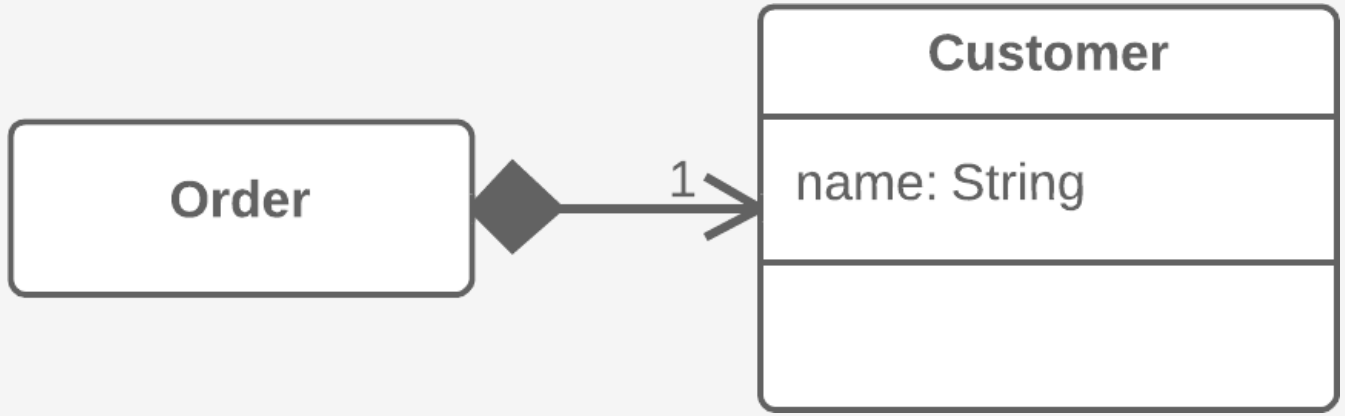




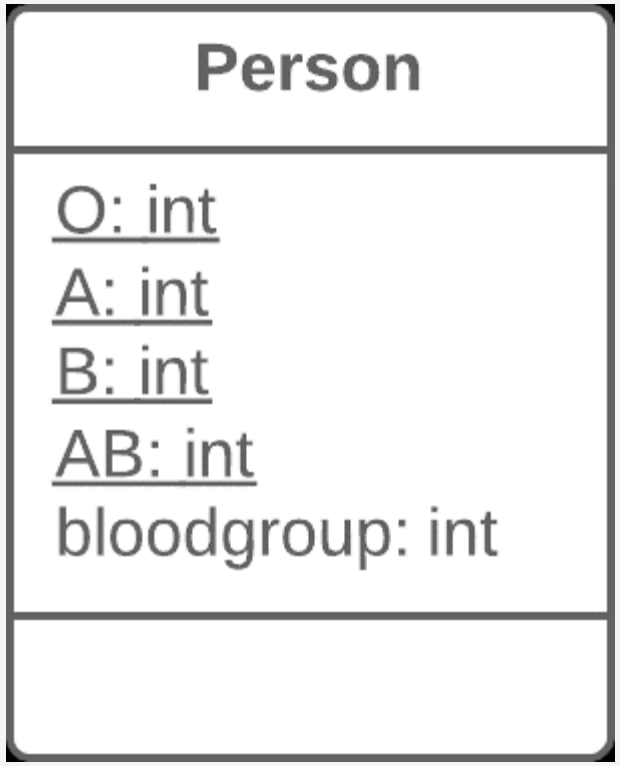


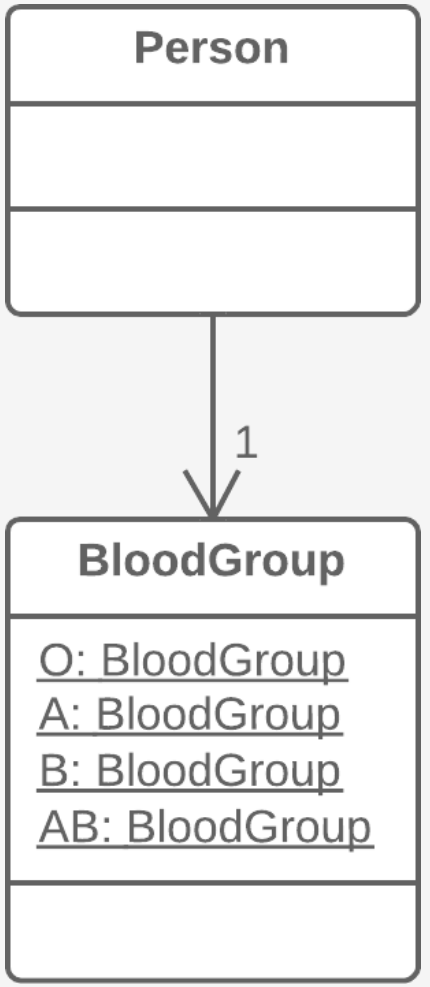
Replace Data Value with Object



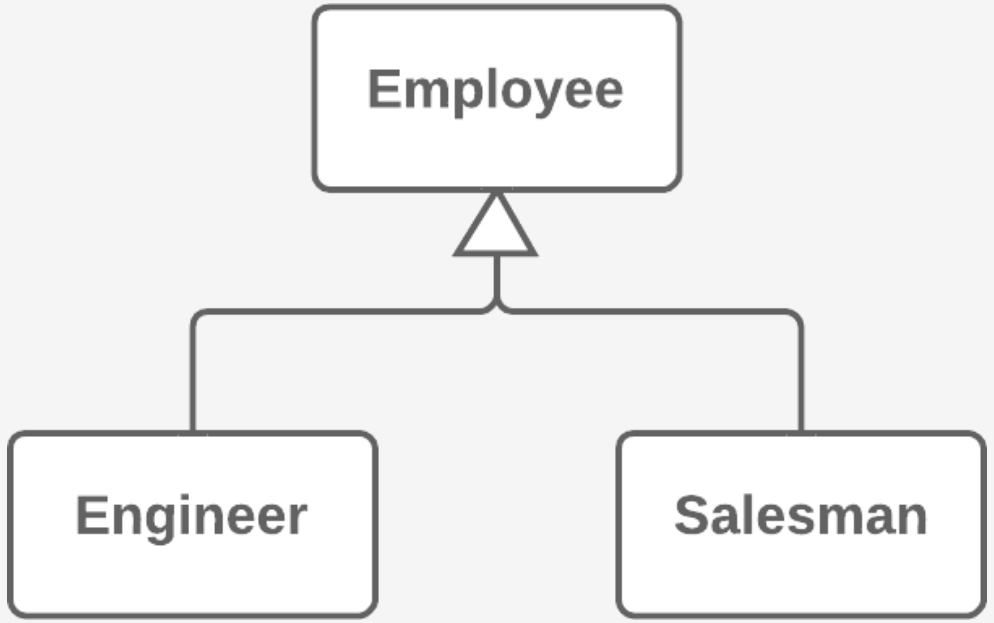


Replace Type Code with Class (Enum)

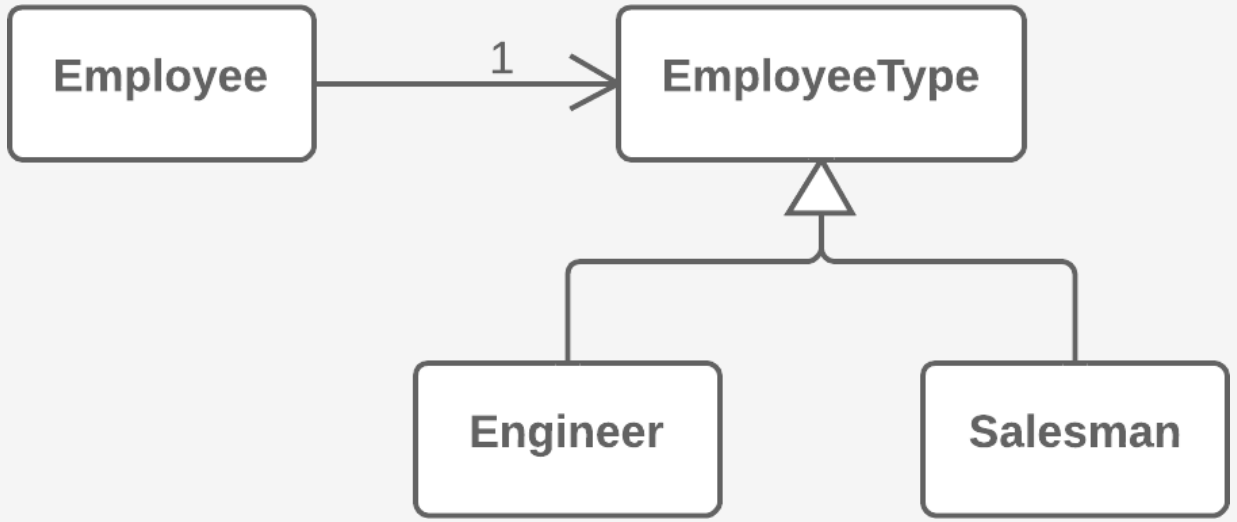




Replace Type Code with Subclasses



Replace Type Code with State/Strategy



Code Smells

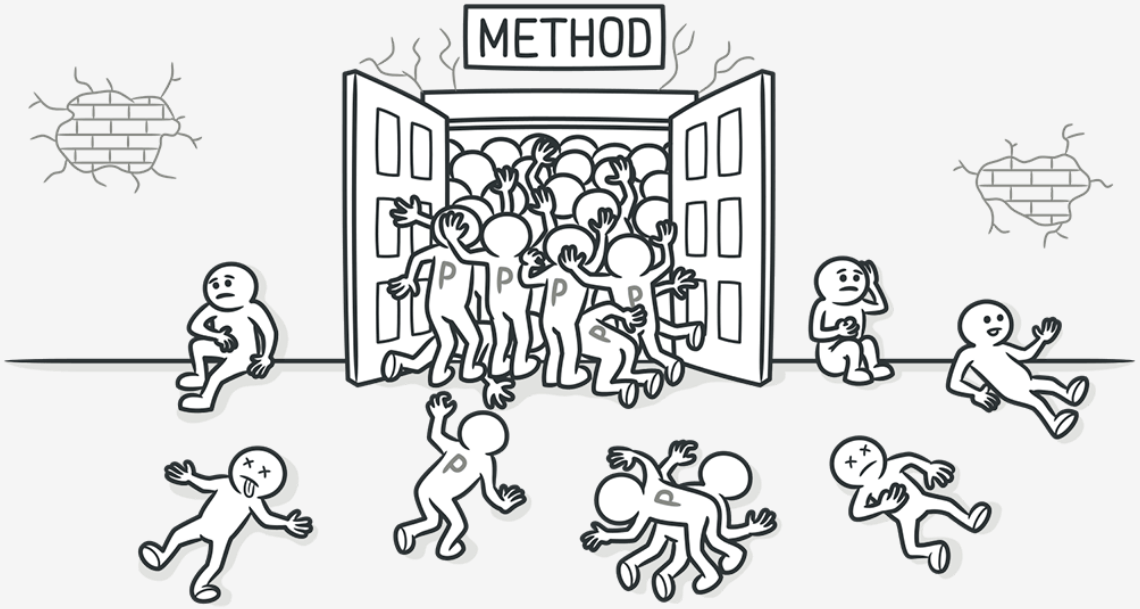
Replace Array with Object

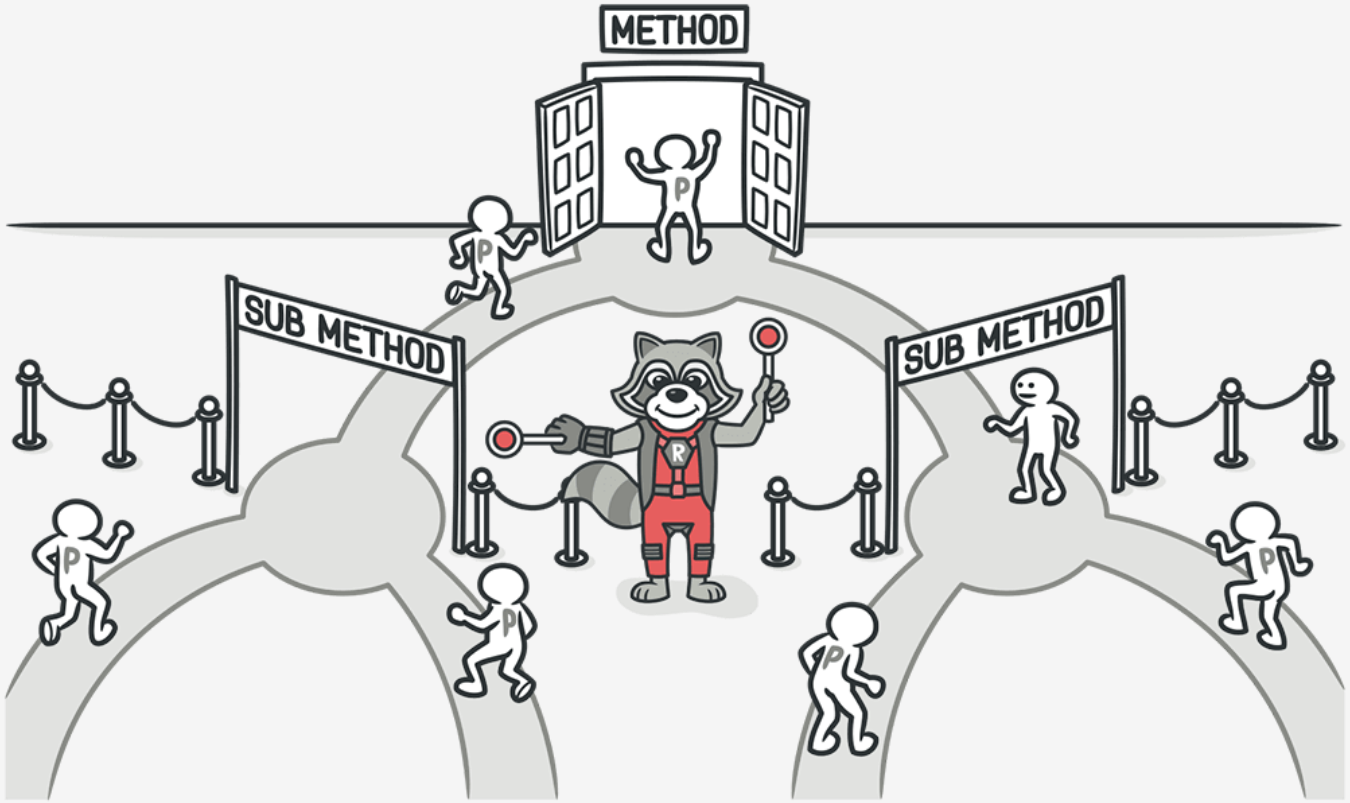
```
string[] row = new string[2];
row[0] = "Liverpool";
row[1] = "15";
```

```
Performance row = new Performance
{
    Name = "Liverpool",
    Wins = "15"
};
```

Long Parameter List

- More than three or four parameters for a method.
- More readable, shorter code.
- Refactoring may reveal previously unnoticed duplicate code.
- **Do not get rid of parameters if doing so would cause unwanted dependency between classes.**
- You can apply **Preserve Whole Object** and **Introduce Parameter Object**.





Code Smells

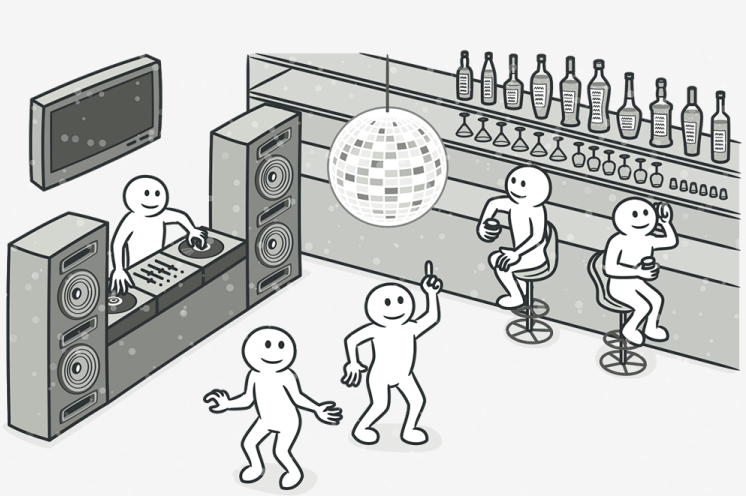
Replace Parameter with Method Call

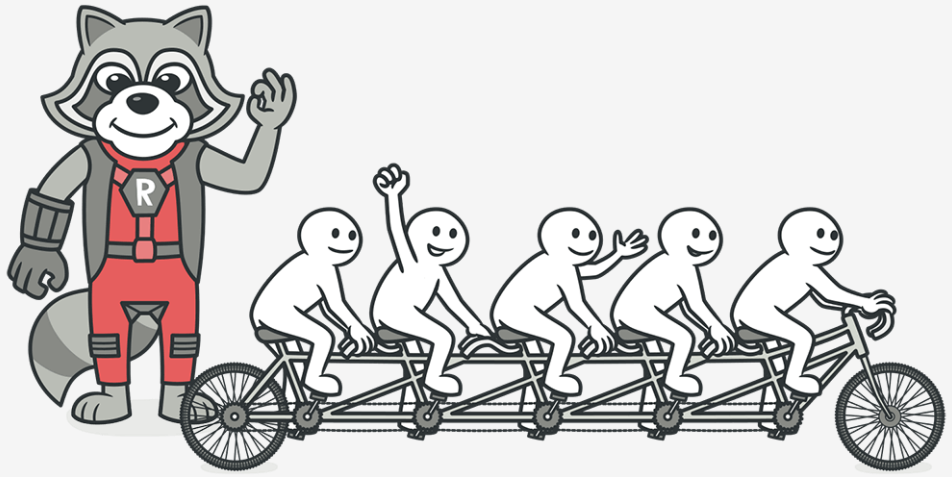
```
int basePrice = quantity * itemPrice;
double seasonDiscount = this.GetSeasonalDiscount();
double fees = this.GetFees();
double finalPrice = DiscountedPrice(basePrice, seasonDiscount, fees);
```

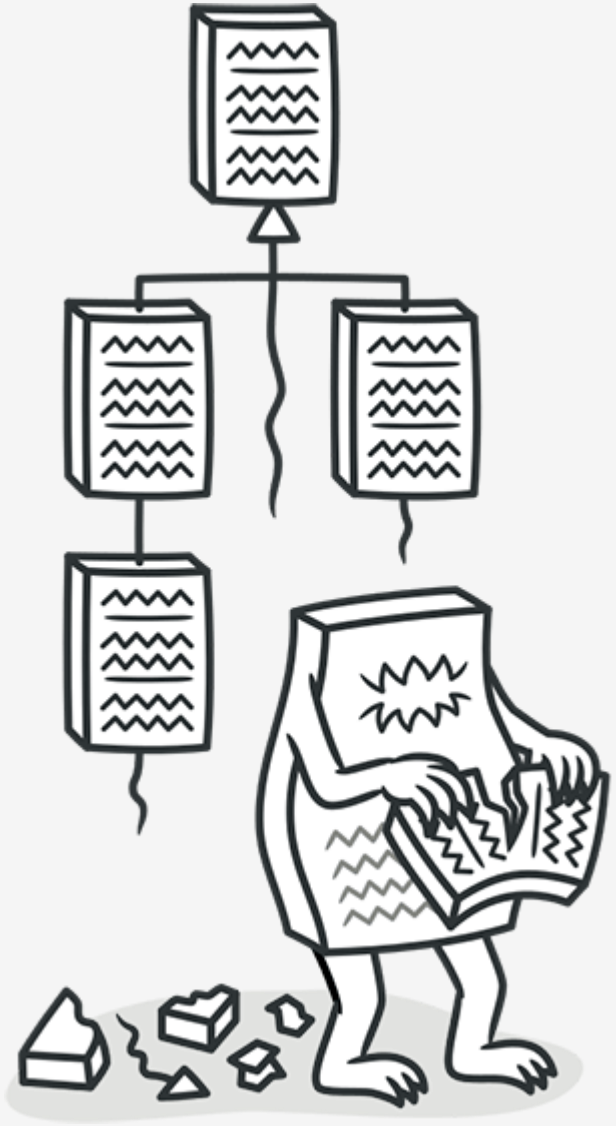
```
int basePrice = quantity * itemPrice;
double finalPrice = DiscountedPrice(basePrice);
```

Data Clumps

- Parts of the code contain identical groups of variables (for example: connecting to DB variables)
- These clumps must be turned into their own classes.
- You can apply **Extract Class, Introduce Parameter Object** and **Preserve whole object**.





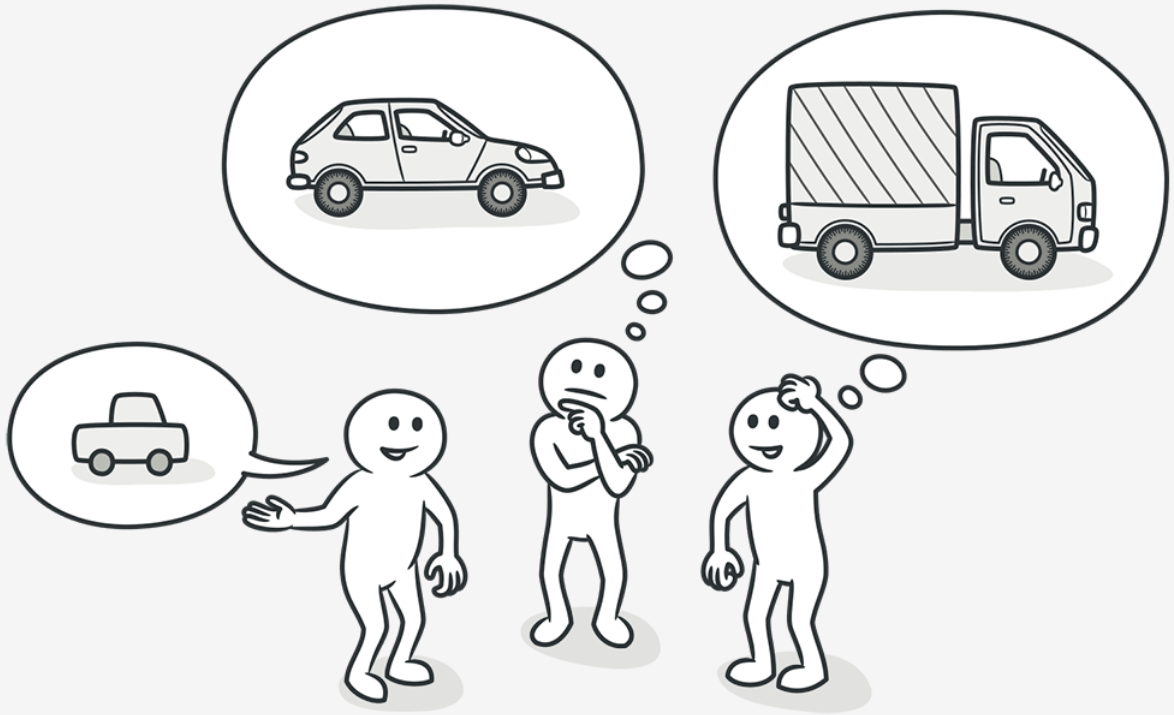


Object-Orientation Abusers

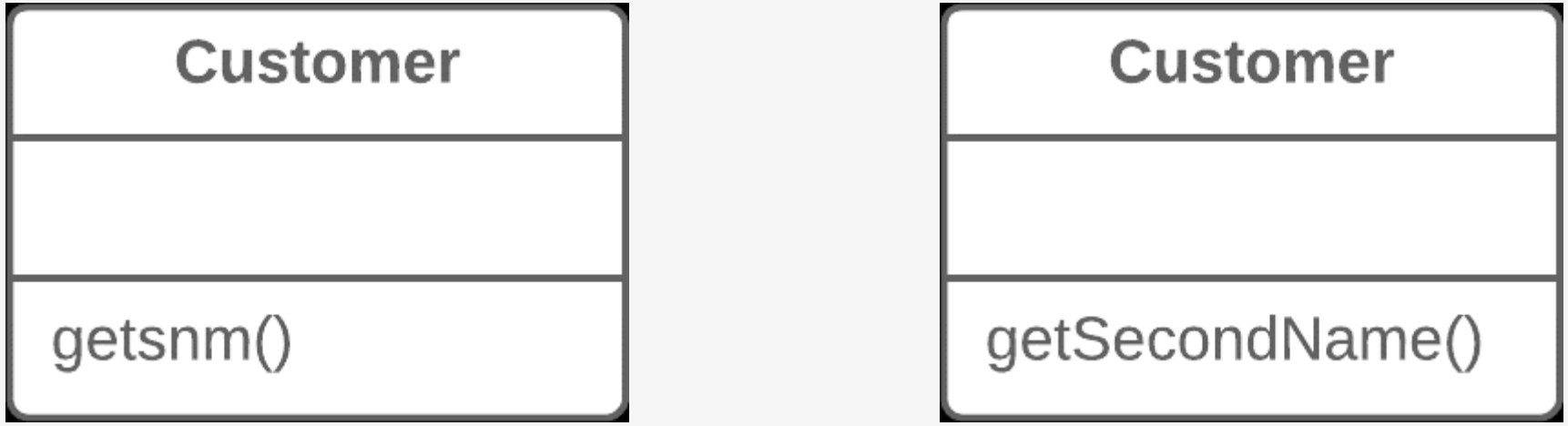
All these smells are incomplete or incorrect application of object-oriented programming principles.

Alternative Classes with Different Interfaces

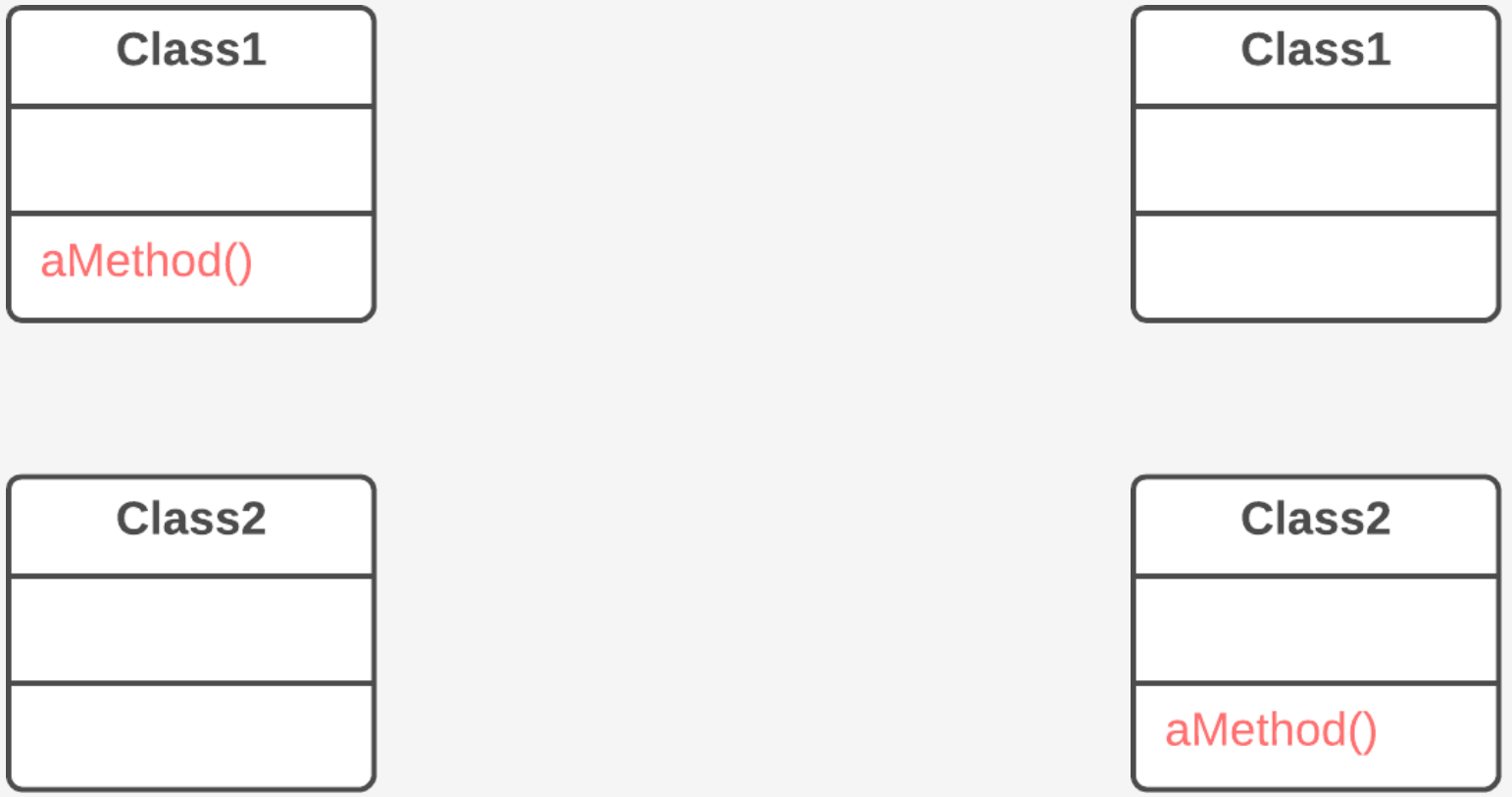
- Two classes perform identical functions but have different method names.
- You may be able to delete one of the classes after:
  - **Rename Methods** to make them identical in all alternative classes.
  - **Move method**, **Add Parameter**, **Parameterize Method** to make the signature and implementation of methods the same.
  - **Extract superclass**. In this case, the existing classes will become subclasses.



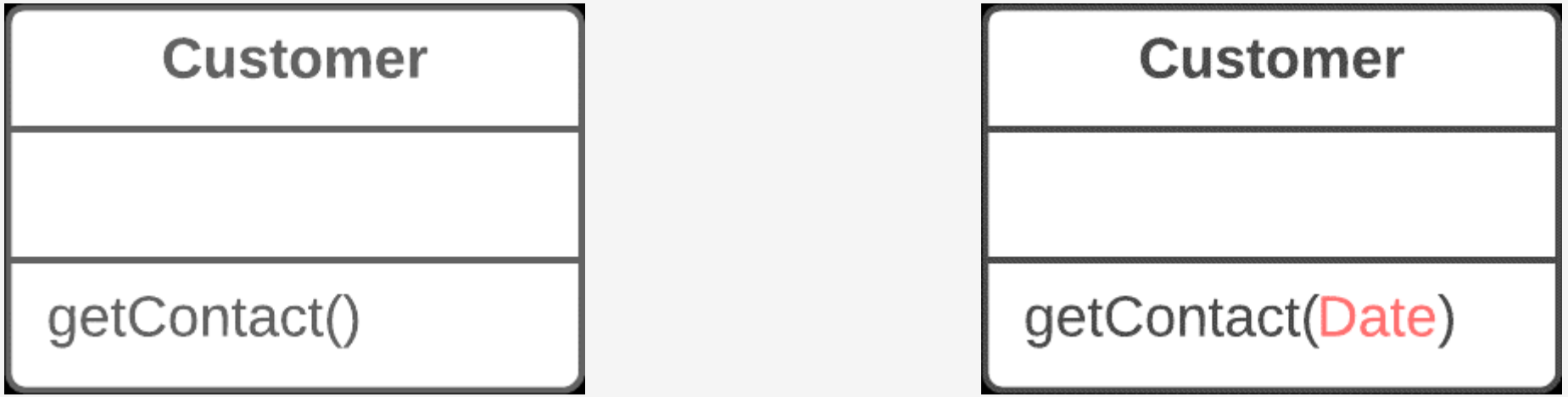
Rename Method



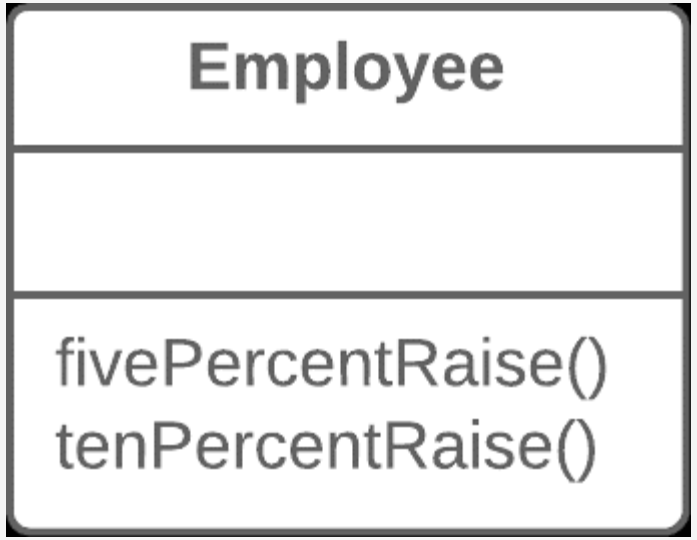
Move Methods

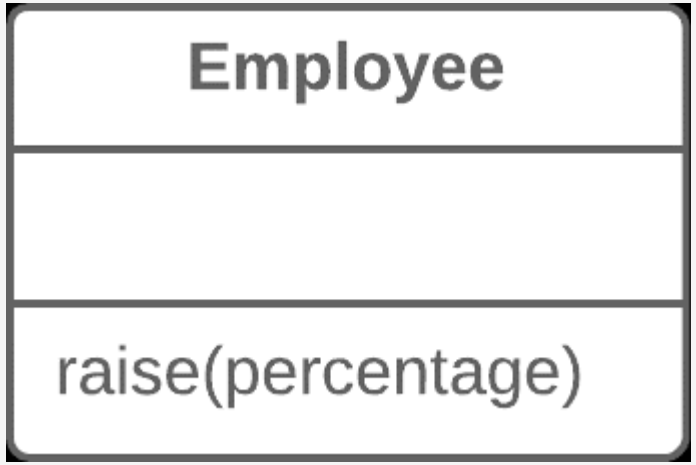


Add Parameter

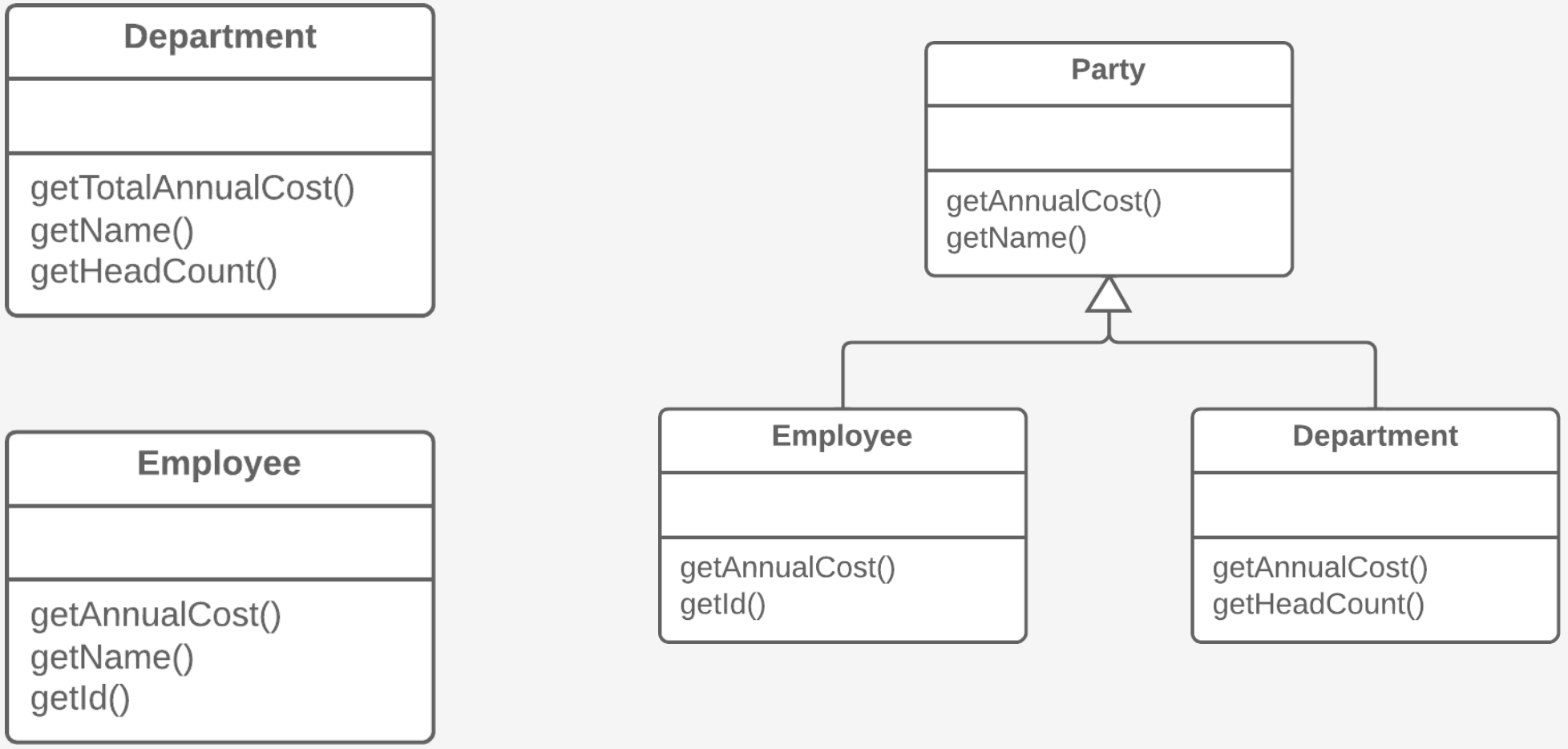


Parameterize Method



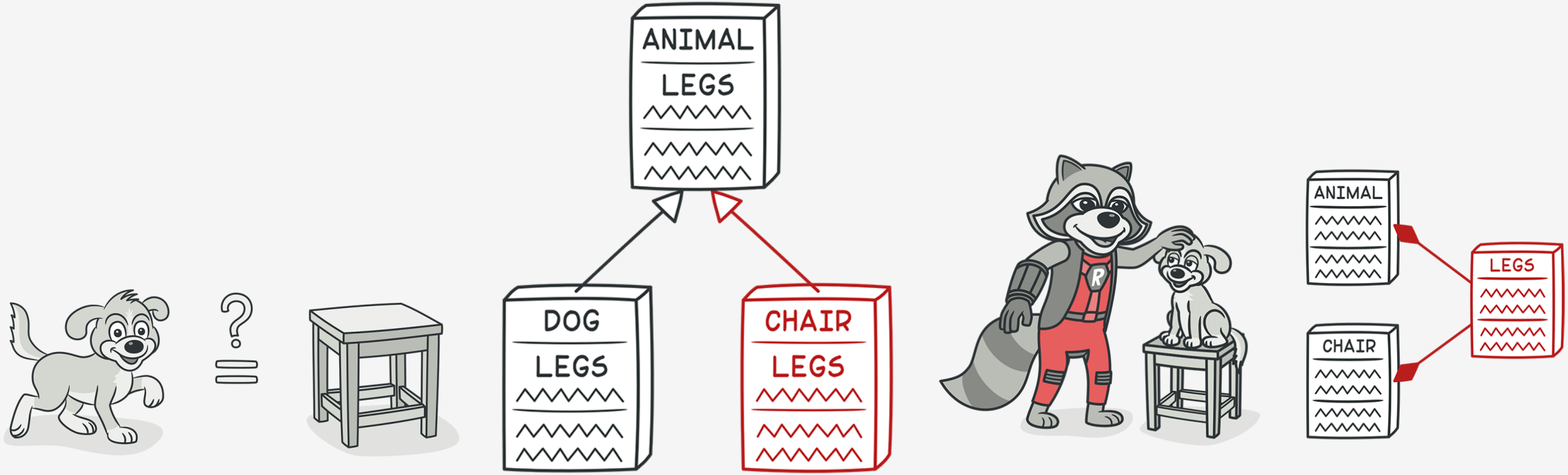


Extract Superclass

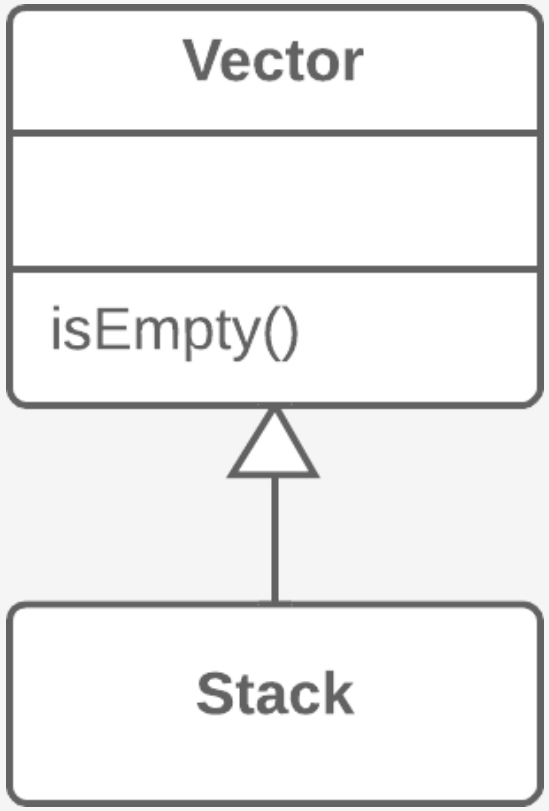


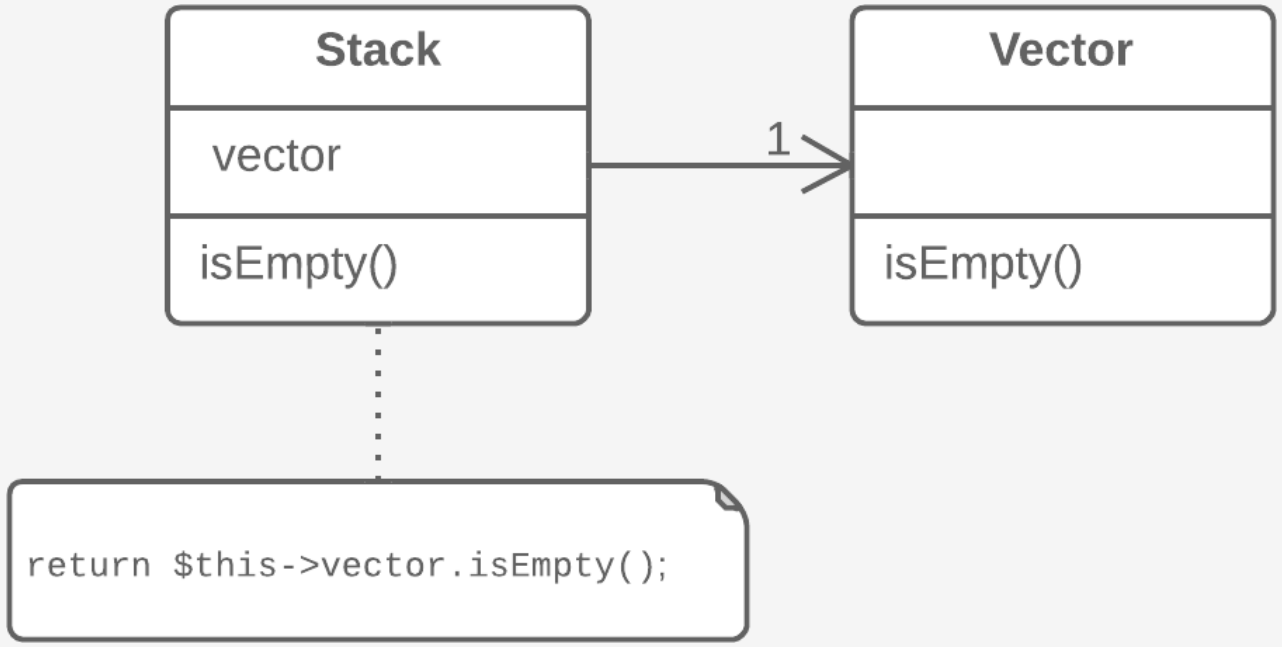
Refused Bequest

- If a subclass uses only some of the methods and properties inherited from its parents, the hierarchy is off-kilter
- If inheritance makes no sense, use **Replace inheritance with delegation**.
- If inheritance is appropriate, **Extract superclass**



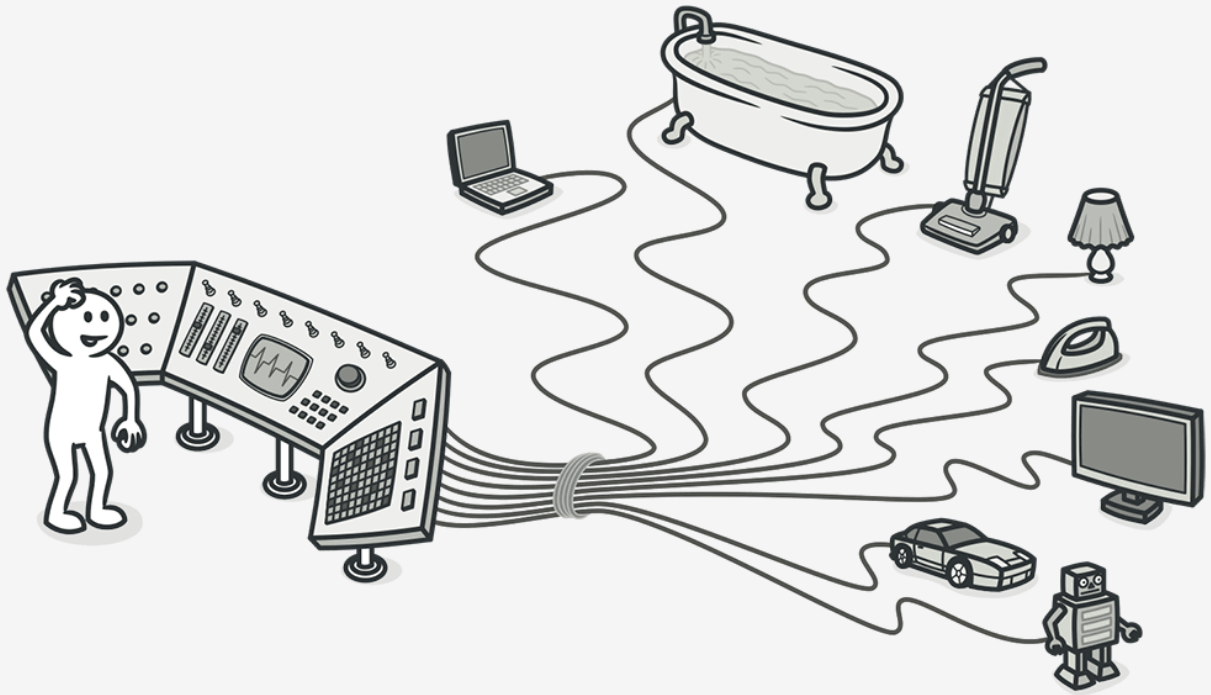
Replace Inheritance with Delegation





Switch Statements

- You have a complex **switch** operator or sequence of **if** statements.
- To isolate switch and put it in the right class, use **Extract Method** and then **Move Method.**
- If it is based on type codes (enums), use **Replace Type Code with Subclasses** or **Replace Type Code with State/Strategy**.
- After specifying the inheritance structure, use **Replace Conditional with Polymorphism**.



Code Smells

Replace Parameter with Explicit Methods (Before)

```
void SetValue(string name, int value)
{
    if (name.Equals("height"))
    {
        height = value;
        return;
    }
    if (name.Equals("width"))
    {
        width = value;
        return;
    }
    Assert.Fail();
}
```

Code Smells

Replace Parameter with Explicit Methods (After)

```
public int Height { get; set; }
public int Width { get; set; }
```

Introduce Null Objects (Before)

```
if (customer == null)
{
    plan = BillingPlan.Basic();
}
else
{
    plan = customer.GetPlan();
}
```

Code Smells

```
public sealed class NullCustomer : Customer
{
    public override bool IsNull
    {
        get { return true; }
    }

    public override Plan GetPlan()
    {
        return new NullPlan();
    }
    // Some other NULL functionality.
}

// Replace null values with Null-object.
customer = order.Customer ?? new NullCustomer();

// Use Null-object as if it's normal subclass.
plan = customer.GetPlan();
```

Code Smells

Replace Conditional with Polymorphism (Before)

```
public class Bird : Animal
{
    BirdType Type { get; set; }
    //...
    public double GetSpeed()
    {
        switch (Type)
        {
            case BirdType.European:
                return GetBaseSpeed();
            case BirdType.African:
                return GetBaseSpeed() - GetLoadFactor() * numberOfCoconuts;
            case BirdType.NorwegianBlue:
                return isNailed ? 0 : GetBaseSpeed(voltage);
            default:
                throw new Exception("Should be unreachable");
        }
    }

}
```

Code Smells

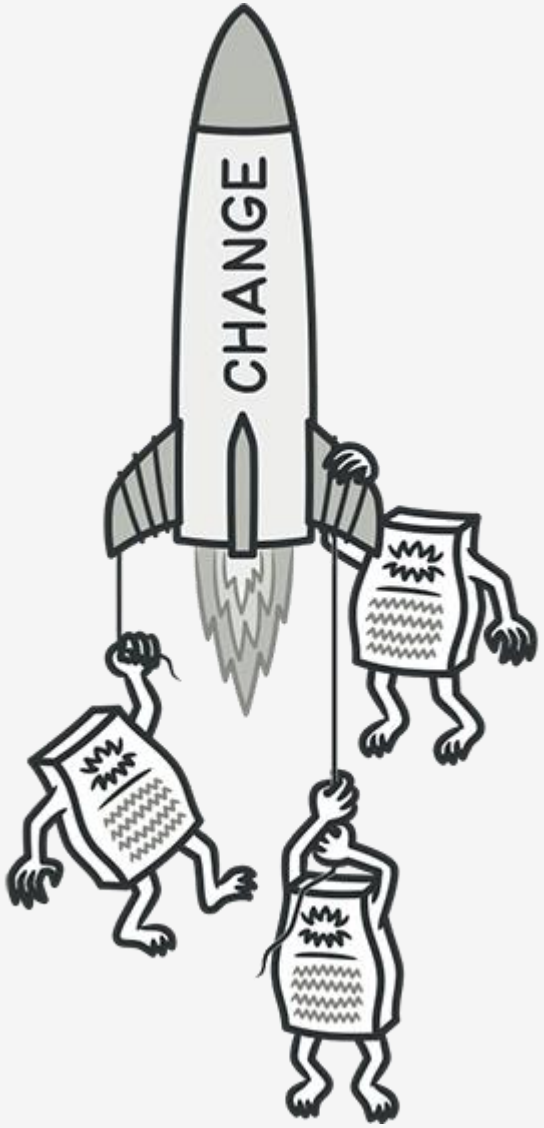
Replace Conditional with Polymorphism (After)

```
public abstract class Bird : Animal
{
    //...
    public abstract double GetSpeed();
}

class NorwegianBlue : Bird
{
    public override double GetSpeed()
    {
        return isNailed ? 0 : GetBaseSpeed(voltage);
    }
}


 // Somewhere in client code
 speed = bird.GetSpeed();
```

Temporary Fields

- Temporary fields get their values only under certain circumstances. Outside of these circumstances, they are empty.
- Put temporary fields in separate class, using **Extract Class.** In fact, you are creating **MethodObject.**
- You can perform **Replace Method with Method Object**..
- **Introduce Null Object** to check the temporary field values for existence.

Code Smells

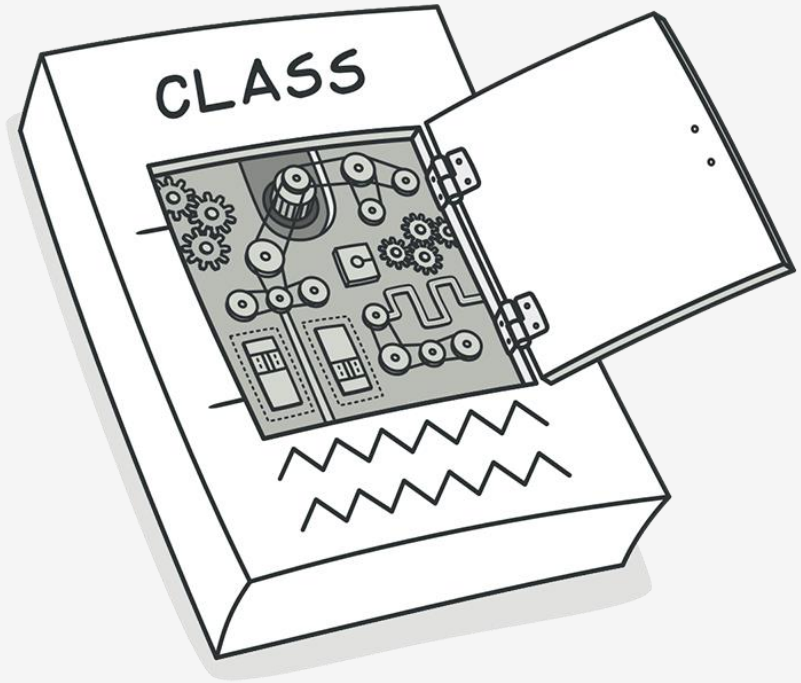


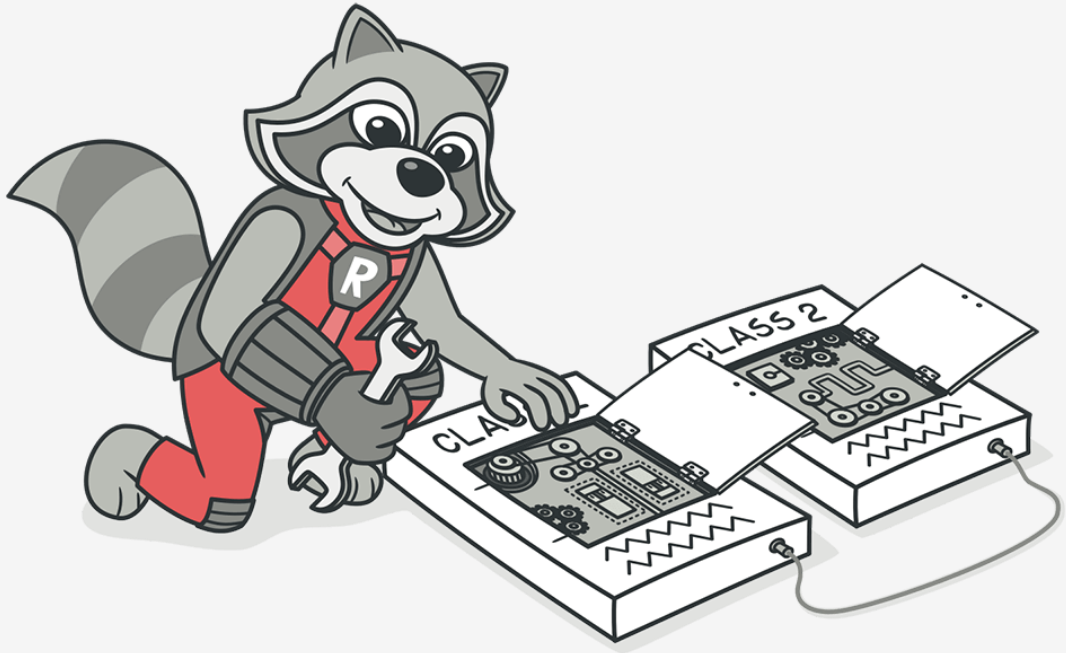
Change Preventers

These smells mean that if you need to change something in one place in your code, you have to make many changes in other places too. Program development becomes much more complicated and expensive as a result.

Divergent Change

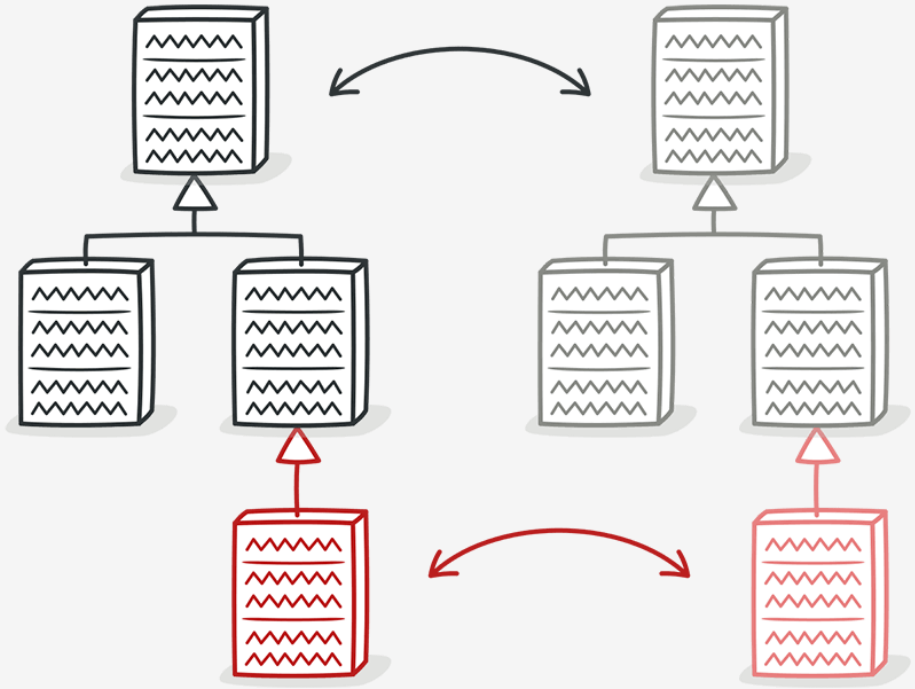
- You have to change many part of a class, with a single change.
- For example, when adding a new product type you have to change the methods for finding, displaying, and ordering products.
- Split up behavior via **Extract Class**.
- If different classes have same behavior, combine classes with inheritance (**Extract Sub or Super Class**).

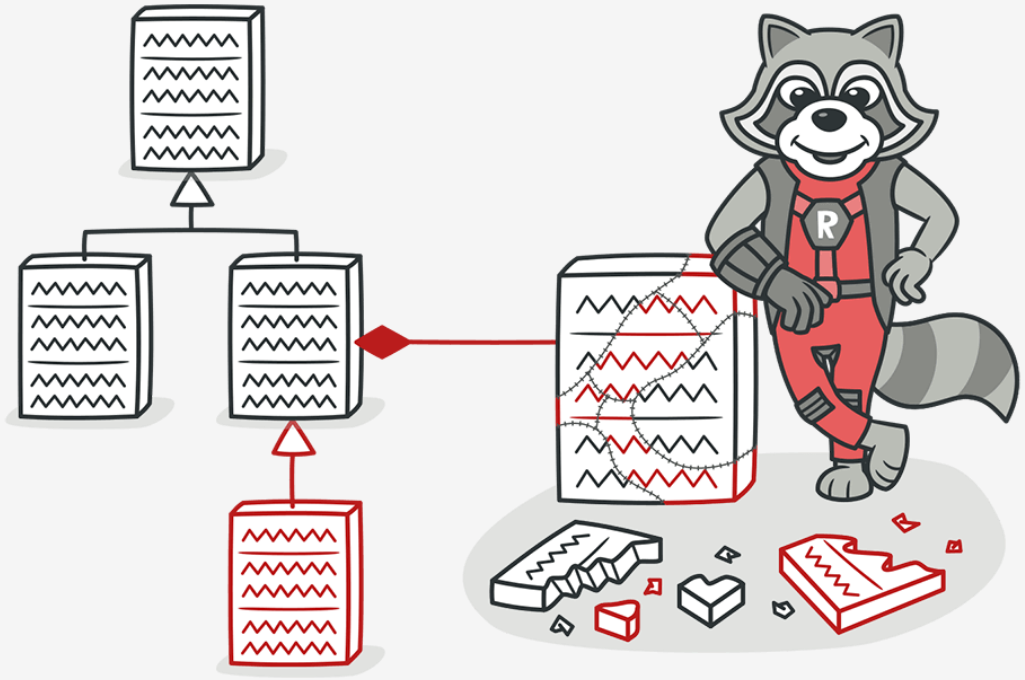




Parallel Inheritance Hierarchies

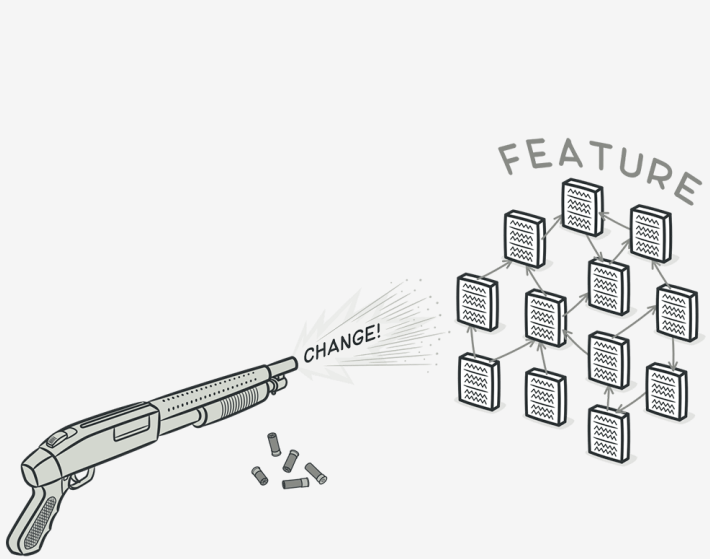
- Whenever you create a subclass for a class, you find yourself needing to create a subclass for another class.
- De-duplicate parallel class hierarchies in two steps:
  - Make instances of one hierarchy refer to instances of another hierarchy
  - Remove the hierarchy in the referred class, by using **Move Method** and **Move Field**.

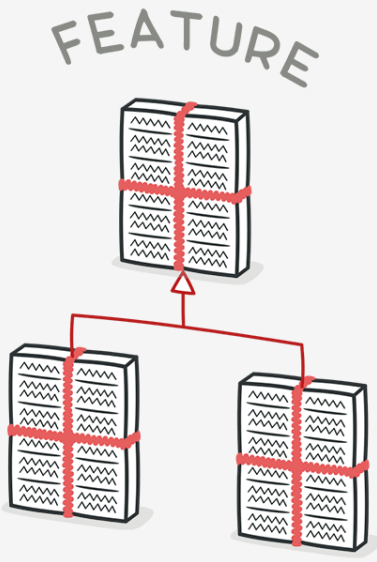




Shotgun Surgery

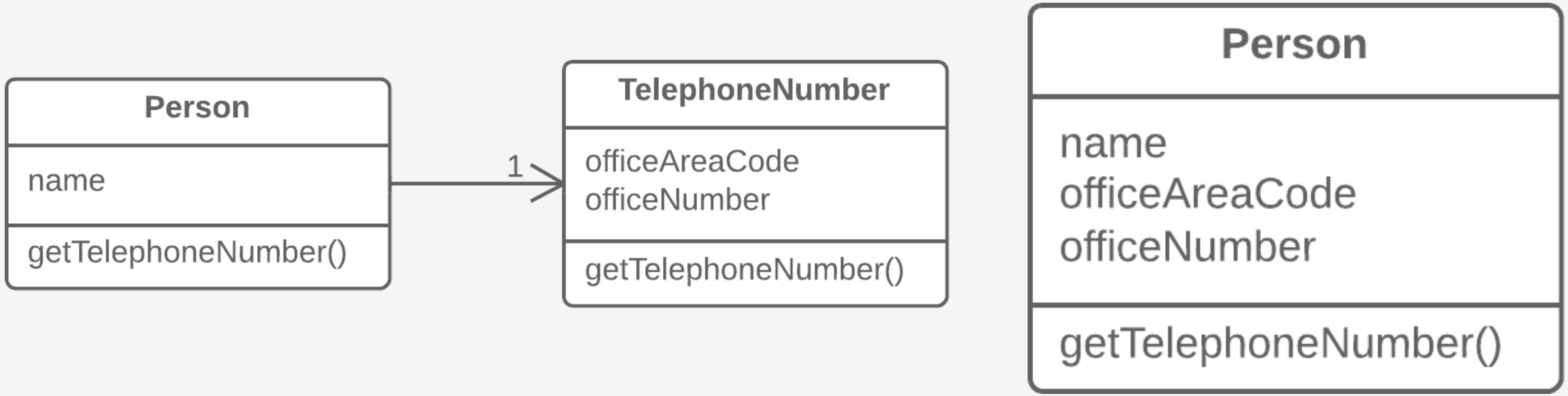
- You have to make many small changes to many different classes, after a small change.
- A single responsibility has been split up among a large number of classes. This can happen after overzealous application of **Divergent Change**.
- Use **Move Method** and **Move Field** to move existing class behaviors into a single class.
- If there is no class appropriate for this, create a new one.
- If moving code to the same class leaves the original classes almost empty, try to get rid of these now-redundant classes via **Inline Class.**





Inline Class

A class does almost nothing and is not responsible for anything, and no additional responsibilities are planned for it.

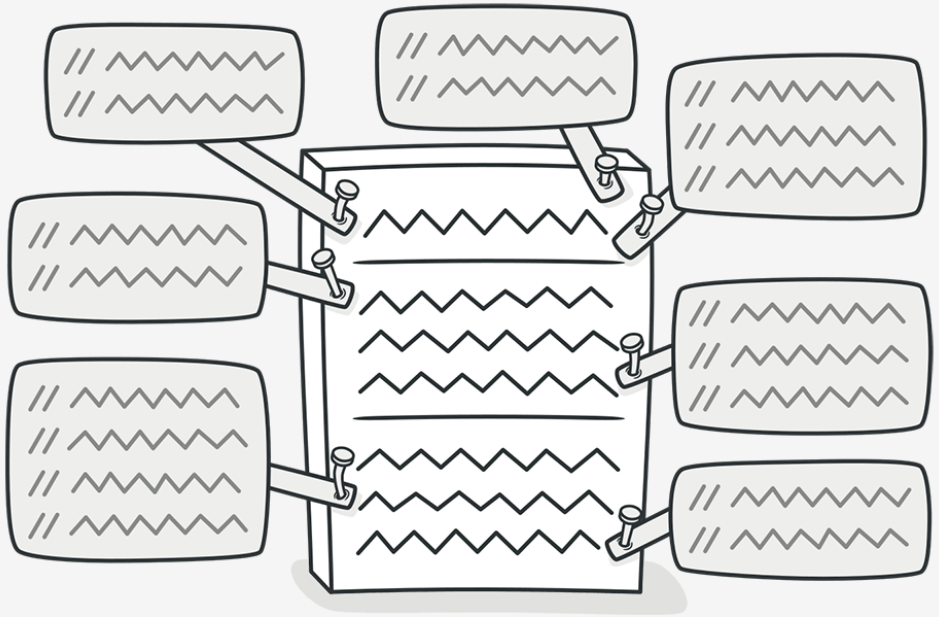


Dispensables

A dispensable is something pointless and unneeded whose absence would make the code cleaner, more efficient and easier to understand.

Comments

- A method is filled with explanatory comments.
- The best comment is a good name for a method or class.
- If a comment is intended to explain a complex expression, the expression should be split into understandable subexpressions using **Extract Variable**.
- If a comment explains a section of code, this section can be turned into a separate method via **Extract Method**.
- If a method has already been extracted, but comments are still necessary to explain: **Rename Method**
- If you need to assert rules about a state that is necessary for the system to work, use **Introduce Assertion.**



Code Smells

Extract Variable

```
if ((platform.ToUpper().IndexOf("MAC") > -1) &&
    (browser.ToUpper().IndexOf("IE") > -1) &&
     WasInitialized() && resize > 0)
{
   // do something
}
```

```
bool isMacOs = platform.ToUpper().IndexOf("MAC") > -1;
bool isIE = browser.ToUpper().IndexOf("IE") > -1;
bool wasResized = resize > 0;

if (isMacOs && isIE && WasInitialized() && wasResized)
{
   // do something
}
```

Code Smells

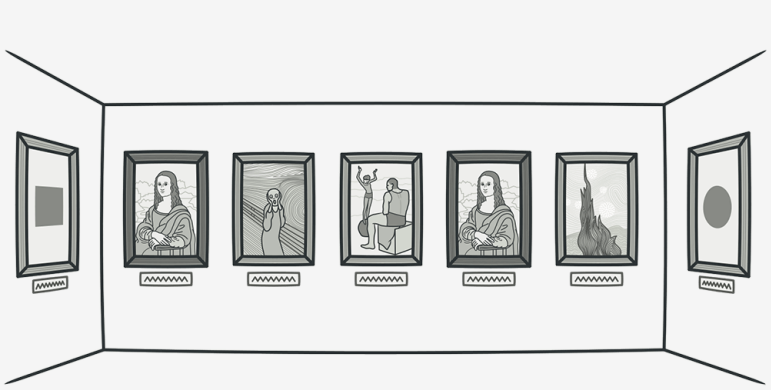
Introduce Assertion

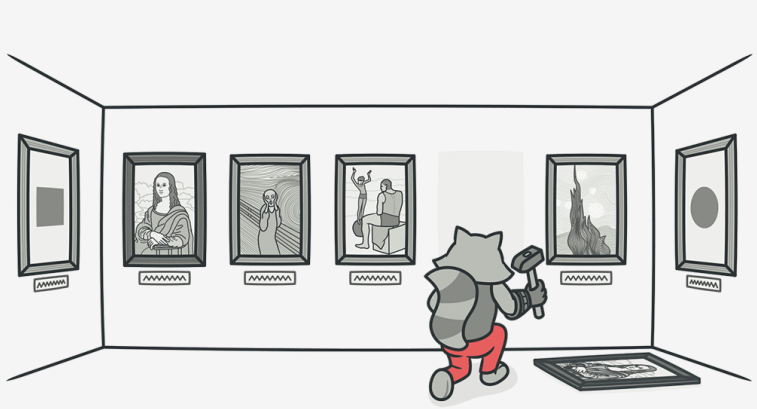
```
double GetExpenseLimit()
{
    // should have either expense limit or a primary project
    return (expenseLimit != (double)ExpenseType.NullExpense) ?
      expenseLimit :
      primaryProject.GetMemberExpenseLimit();
}
```

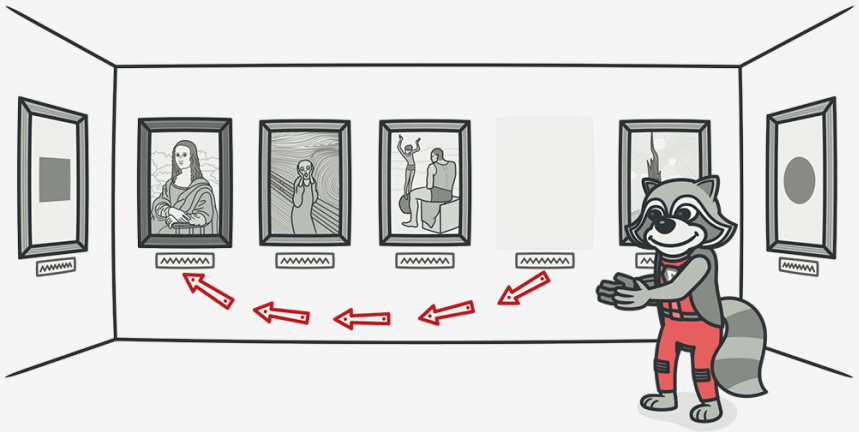
```
double GetExpenseLimit()
{
    Assert.IsTrue(expenseLimit != (double)ExpenseType.NullExpense ||
        primaryProject != null);

    return (expenseLimit != (double)ExpenseType.NullExpense) ?
      expenseLimit :
      primaryProject.GetMemberExpenseLimit();
}
```

Duplicate Code

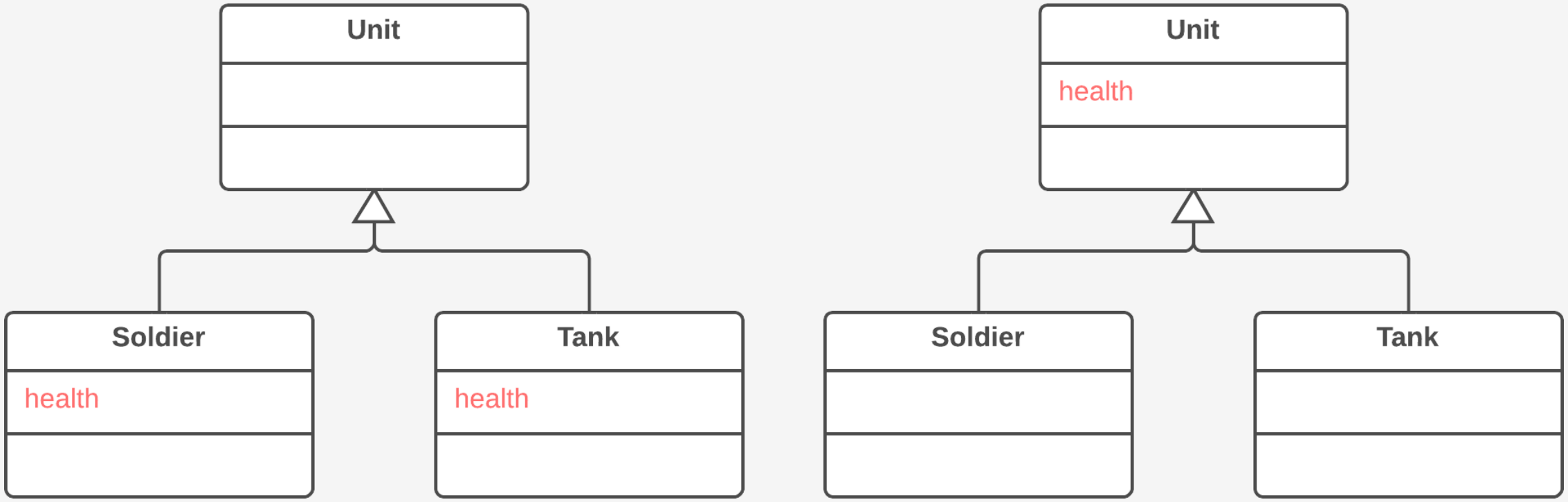
- Two code fragments look almost identical.
- Same code found in two subclass of same level?
  - Two subclass in the same level: **Extract Method**, followed by **Pull Up Field**.
  - Duplicate code inside a constructor? Use **Pull Up Constructor Body**.
  - Duplicate code is similar, but not completely identical? Use **Form Template Method**.
  - Codes does same things, but different algorithms? Select best algorithm and use **Substitute Algorithm**.
- Same code found in two different classes?
  - Try to **Extract Subclasses or Classes**
- Large conditional expressions perform the same code?
  - **Consolidate Conditional Expression** and **Extract Method**
- Same code is performed in all branches of a conditional expression?
  - Use **Consolidate Duplicate Conditional Fragments.**







Pull up field



Code Smells

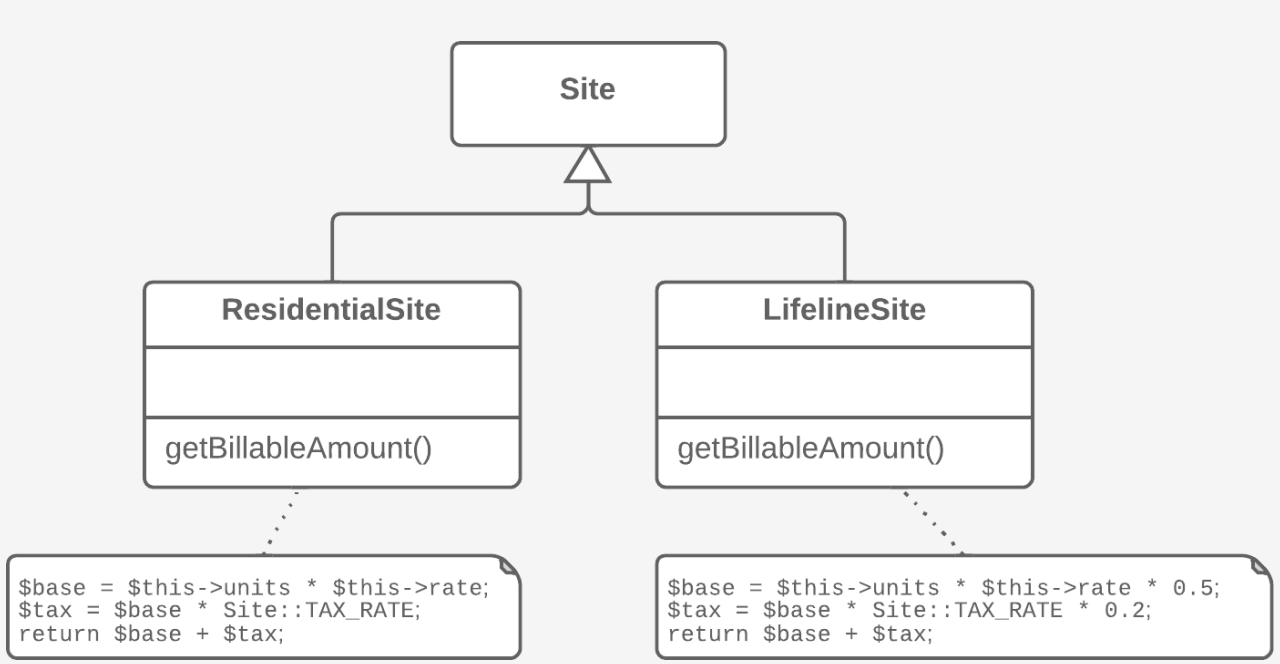
Pull Up Constructor Body (Before)

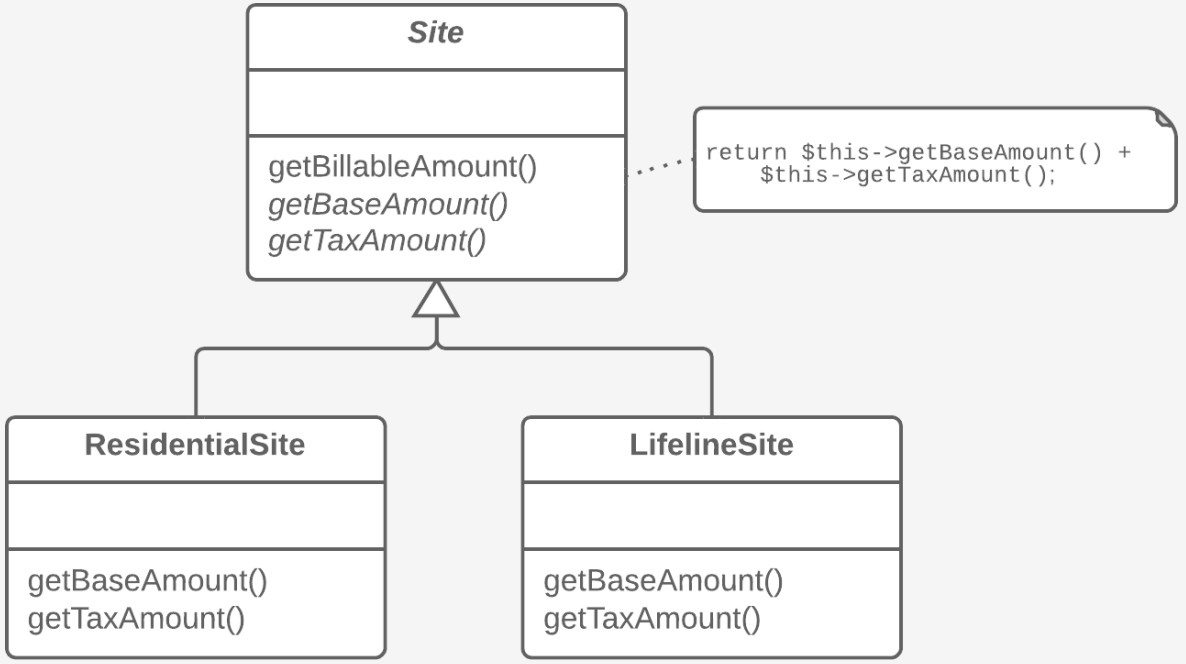
```
public class Manager : Employee
{
    public Manager(string name, string id, int grade)
    {
        this.Name = name;
        this.Id = id;
        this.Grade = grade;
    }
    //...
}
```

Pull Up Constructor Body (After)

```
public class Manager : Employee
{
    public Manager(string name, string id, int grade) : base(name, id)
    {
        this.Grade = grade;
    }
    //...
}
```

Form Template Method





Code Smells

Substitute Algorithm (Before)

```
string FoundPerson(string[] people)
{
    for (int i = 0; i < people.Length; i++)
    {
        if (people[i].Equals("Don"))
        {
            return "Don";
        }
        if (people[i].Equals("John"))
        {
            return "John";
        }
        if (people[i].Equals("Kent"))
        {
            return "Kent";
        }
    }
    return String.Empty;
}
```

Code Smells

Substitute Algorithm (After)

```
string FoundPerson(string[] people)
{
    List<string> candidates = new List<string>() { "Don", "John", "Kent" };

    for (int i = 0; i < people.Length; i++)
    {
        if (candidates.Contains(people[i]))
        {
            return people[i];
        }
    }

    return String.Empty;
}
```

Code Smells

Consolidate Conditional Expression (Before)

```
double DisabilityAmount()
{
    if (seniority < 2)
    {
        return 0;
    }
    if (monthsDisabled > 12)
    {
        return 0;
    }
    if (isPartTime)
    {
        return 0;
    }
    // compute the disability amount
    //...

    return calculatedAmount;
}
```

Code Smells

Consolidate Conditional Expression (After)

```
double DisabilityAmount()
{
    if (IsNotEligableForDisability())
    {
        return 0;
    }
    // compute the disability amount
    //...

    return calculatedAmount;
}
```

Code Smells

Consolidate Duplicate Conditional Expression (Before)

```
if (IsSpecialDeal())
{
    total = price * 0.95;
    Send();
}
else
{
    total = price * 0.98;
    Send();
}
```

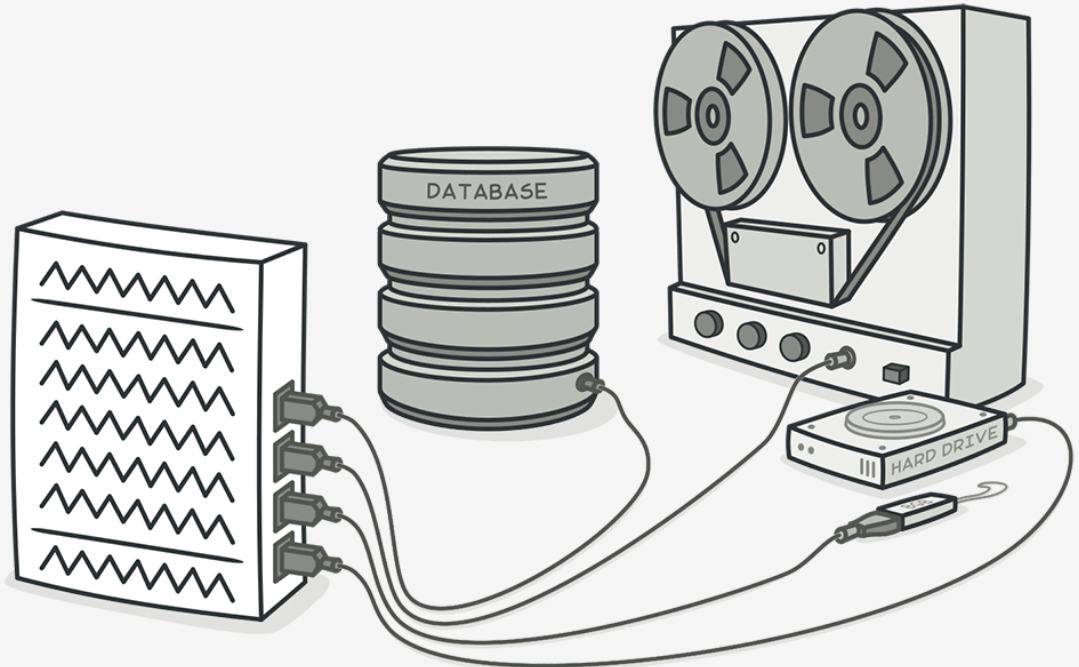
Code Smells

Consolidate Duplicate Conditional Expression (After)

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

Send();
```

Data Class

- Classes that contains only fields and crude methods for accessing them (getters and setters).
- Use properties instead of public fields using **Encapsulate Field**.
- Any data store collections? Use **Encapsulate Collection.**
- Functionality that would be better located in the data class itself? **Move Method** or **Extract Method.**
- Get rid of old method with broad access to data? **Remove setting method** and **Hide method.**



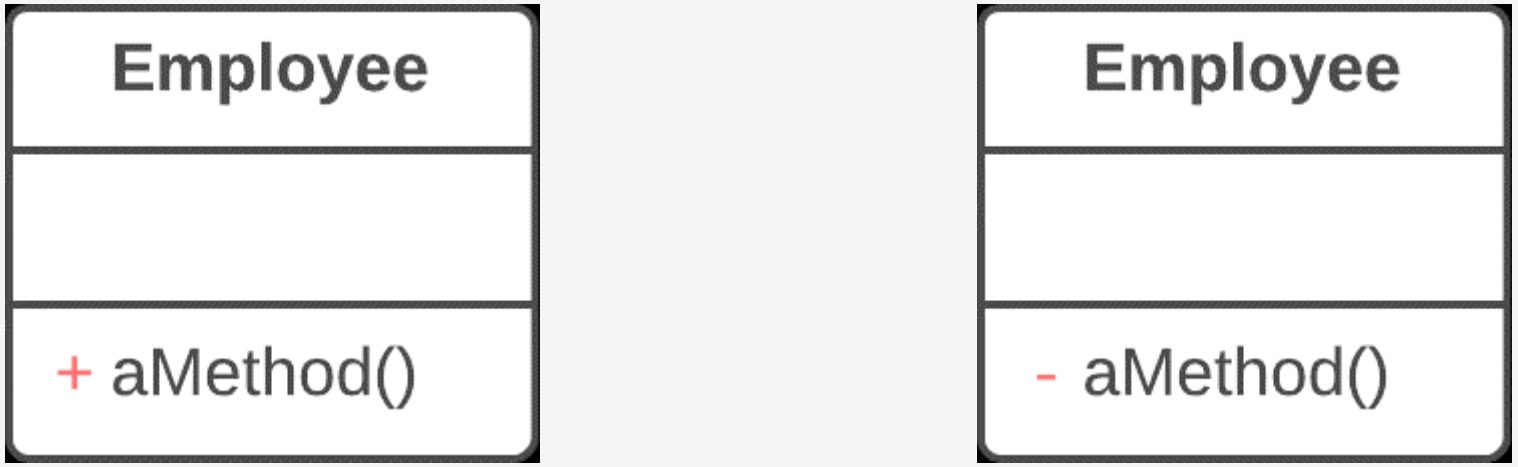
Code Smells

Encapsulate Field

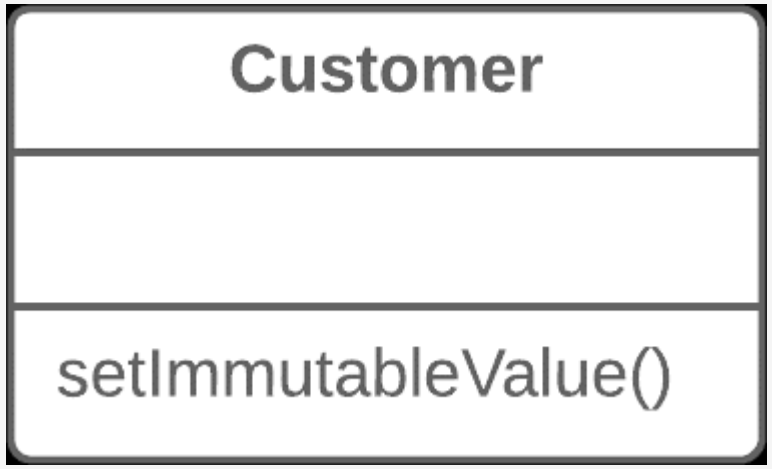
```
class Person
{
    public string name;
}
```

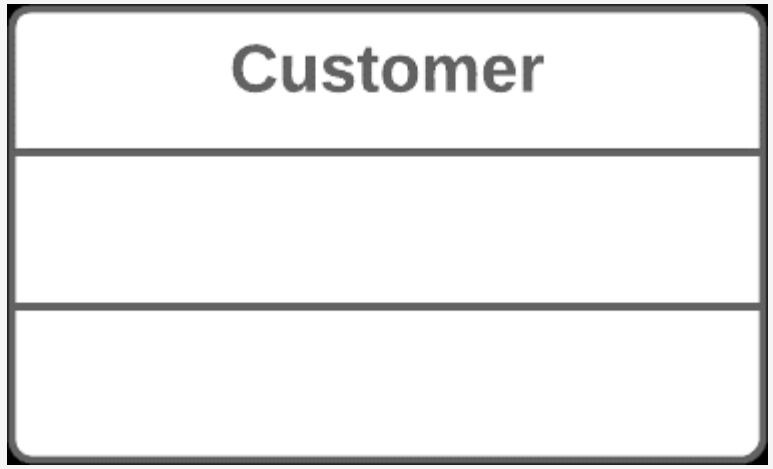
```
class Person
{
    private string name;

    public string Name
    {
        get { return name; }
        set { name = value; }
    }
}
```

Hide Method



Remove Setting Method





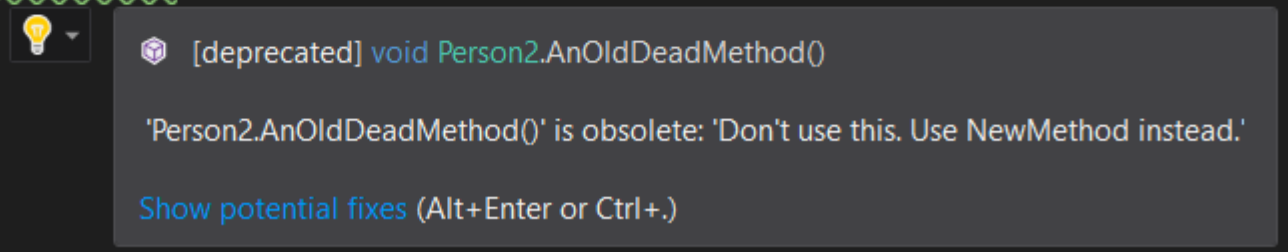
Dead Code

- A variable, parameter, field, method or class is no longer used (usually because it is obsolete).
- Quickest way: Use an IDE
- Delete unneeded code
- Mark code as **Obsolete** and remove after a breaking change.
- Use **Inline Class**, **Collapse Hierarchy**, **Remove Parameter.**

Code Smells

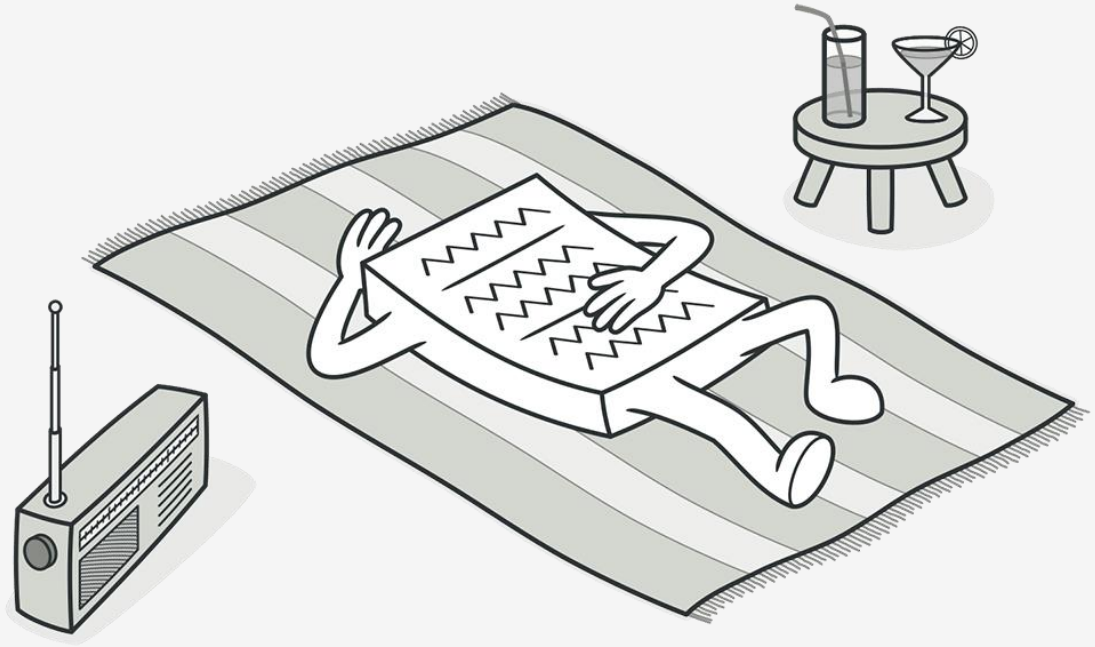
Mark code as Obsolete

```
[Obsolete(message: "Don't use this. Use NewMethod instead.")]
public void AnOldDeadMethod()
{

}

public void Method()
{
    AnOldDeadMethod();
}
```



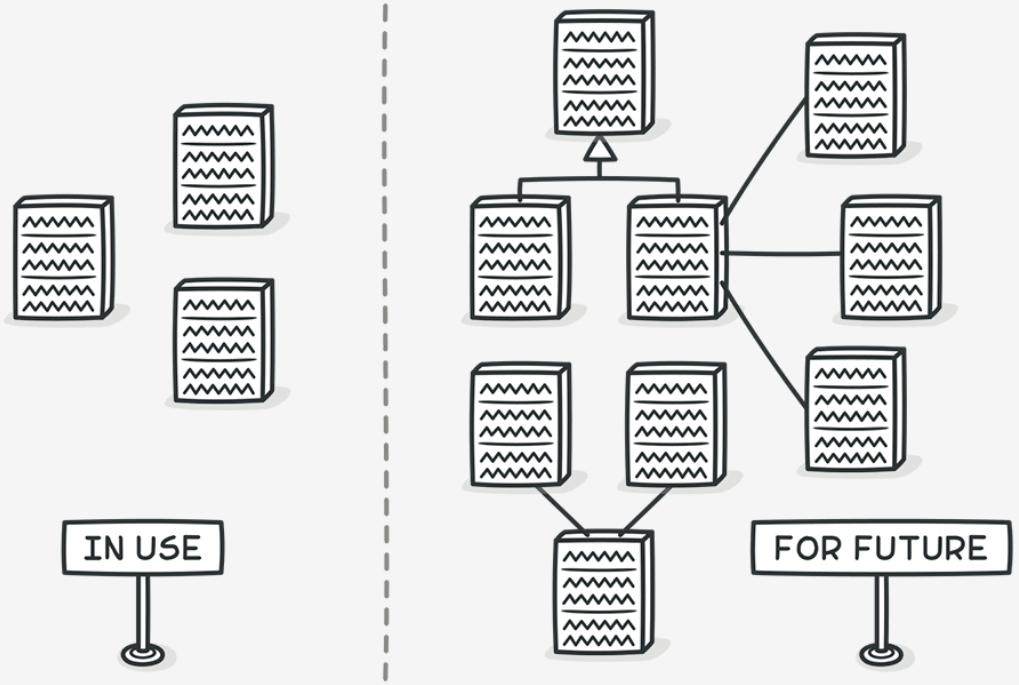
Lazy Class

- Understanding and maintaining classes always costs time and money.
- If a class doesn't do enough to earn your attention, it should be deleted.
- Use **Inline class** and **Collapse Hierarchy.**



Speculative Generality

- There is an unused class, method, field or parameter.
- Created to support anticipated future features that never get implemented.
- Use **Inline class, Collapse Hierarchy, Inline Method, Remove Parameter.**
- Unused fields can simply be deleted.



Couplers

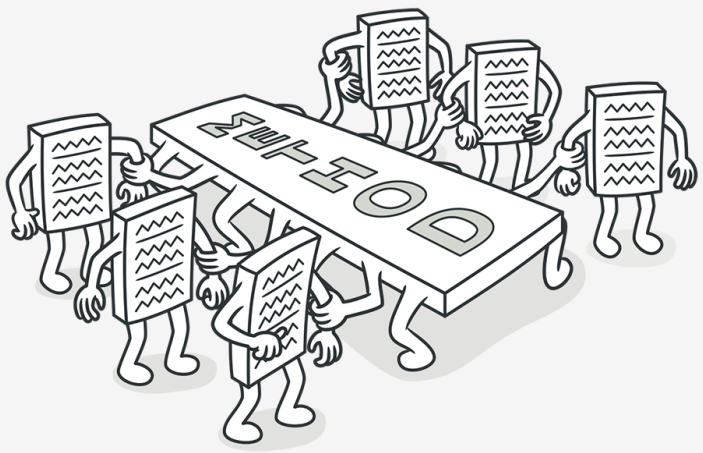
All the smells in this group contribute to excessive coupling between classes or show what happens if coupling is replaced by excessive delegation.

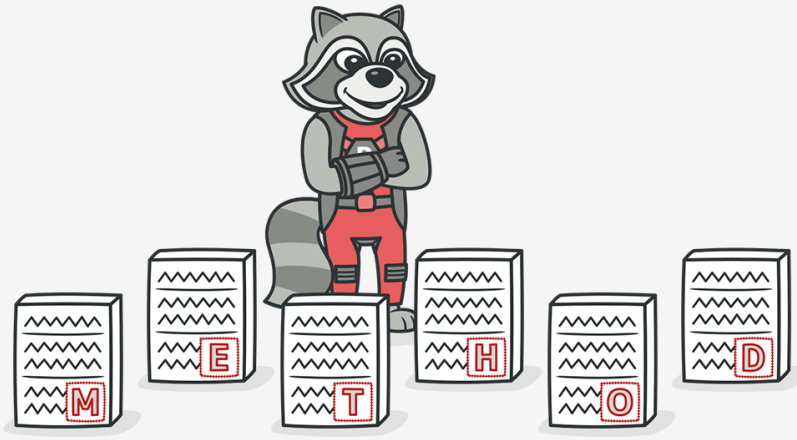
Feature Envy

- A method accesses the data of another object more than its own data.
- If a method clearly should be moved to another place, use **Move Method**.
- If only part of a method accesses the data of another object, use **Extract Method** to move the part in question.

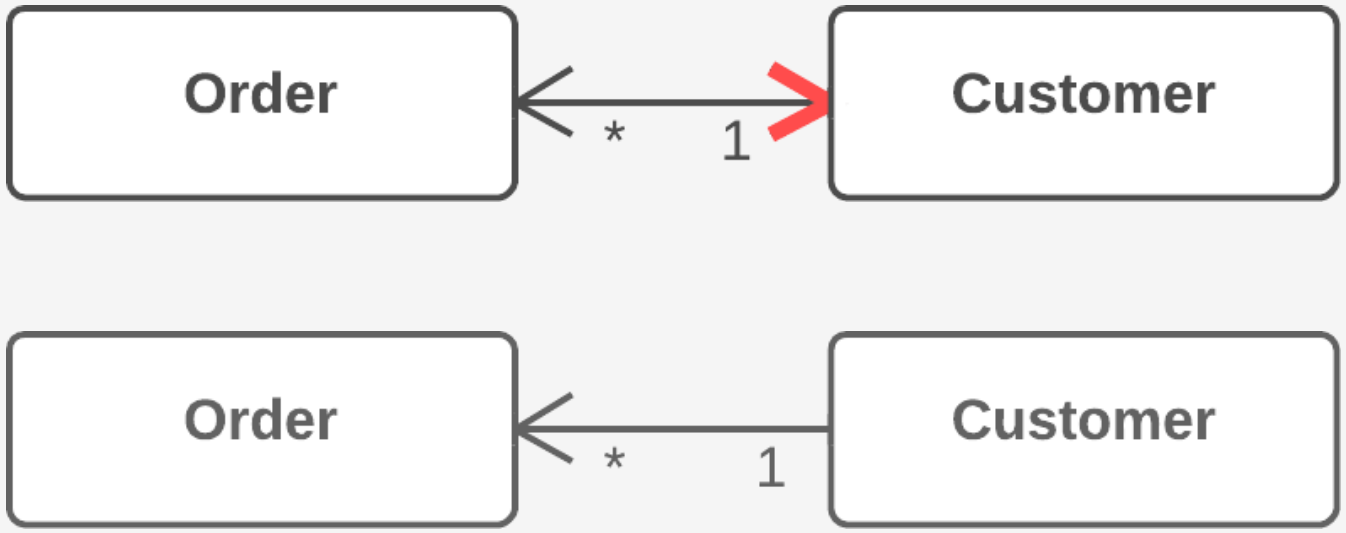
Inappropriate Intimacy

- One class uses the internal fields and methods of another class.
- Simplest solution, use **Move Method and Move Field**.
- If the classes are mutually interdependent, you should use **Change Bidirectional Association to Unidirectional**.
- If this "intimacy" is between a subclass and the superclass, consider **Replace Delegation with Inheritance**.

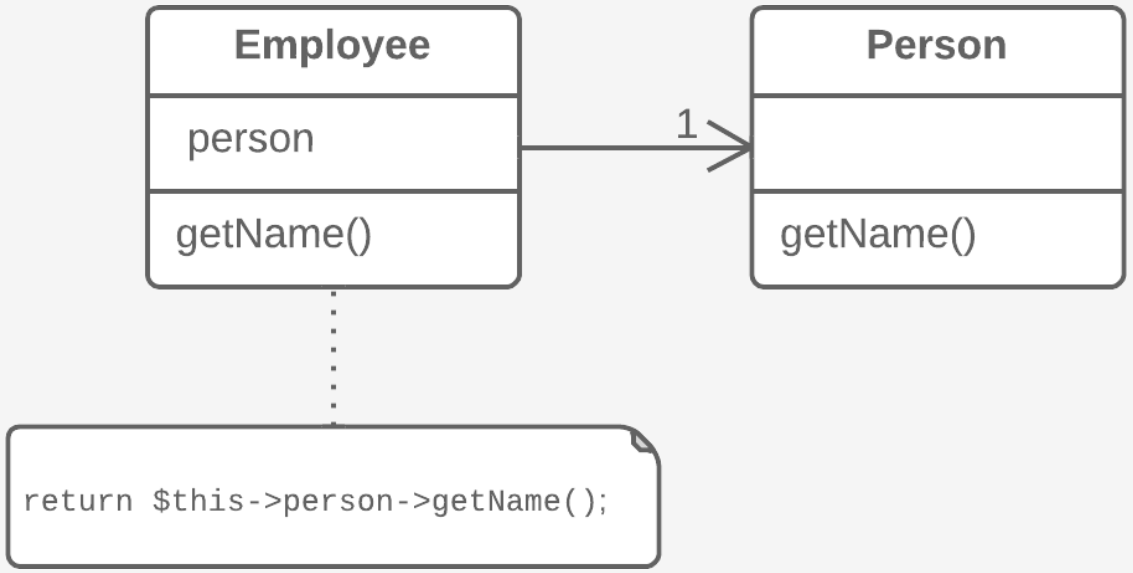


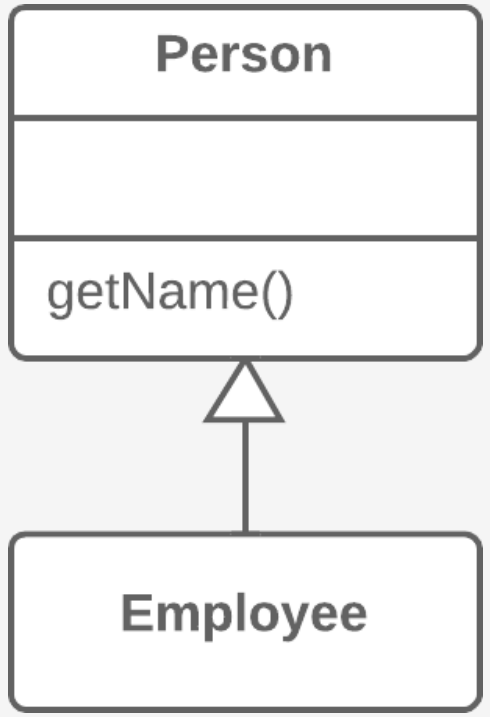


Change Bidirectional Association to Unidirectional



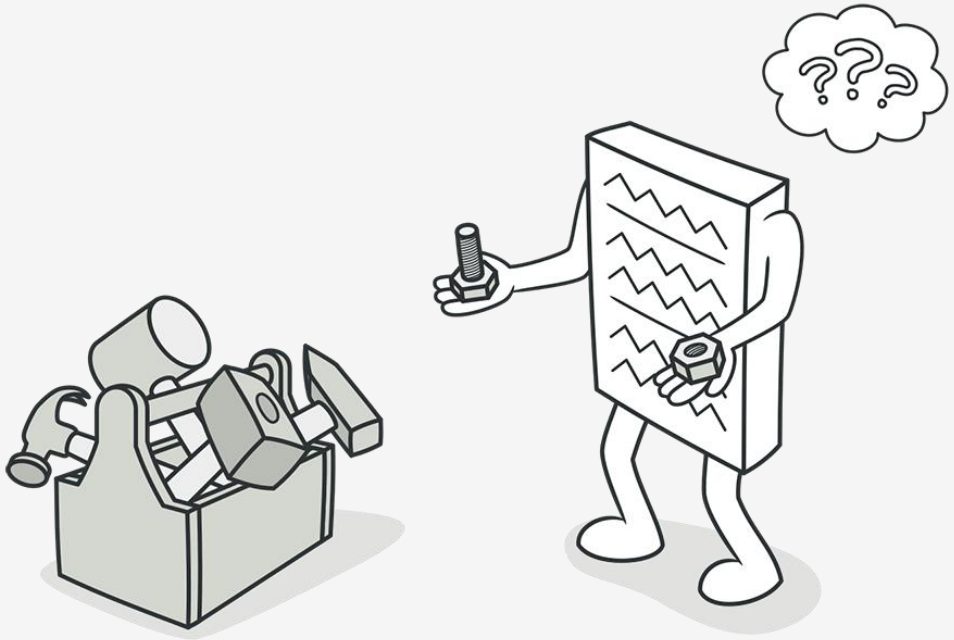
Replace Delegation with Inheritance

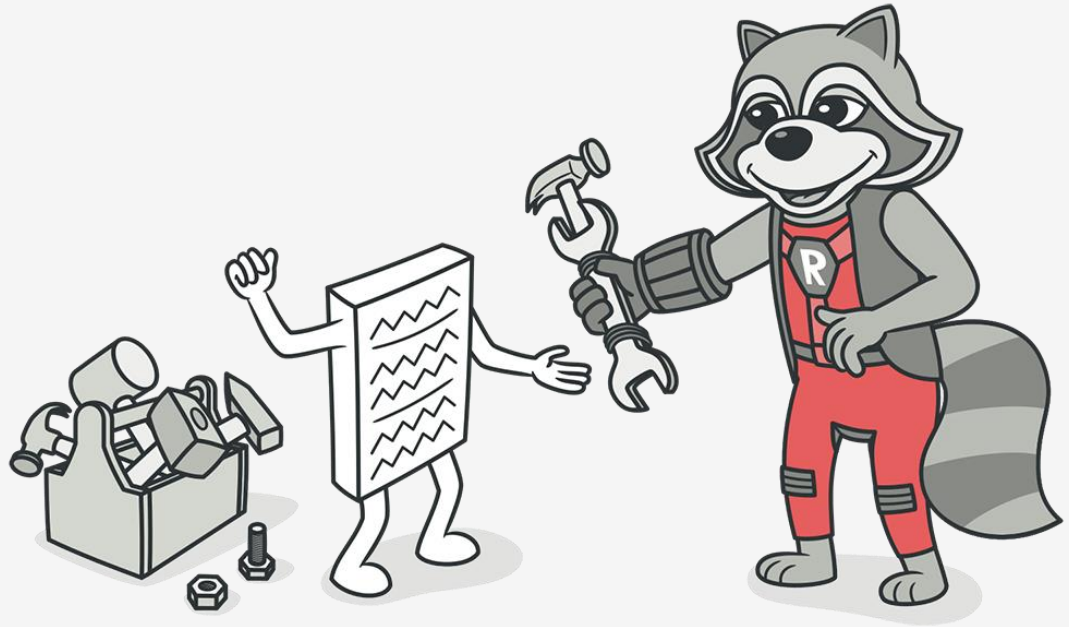




Incomplete Library Class

- Sooner or later, libraries stop meeting user needs.
- The only solution to the problem – changing the library – is often impossible since the library is read-only.
- To introduce a few methods to a library class, use **Introduce Foreign Method**.
  - In C# syntax, you can use *Extension Methods*.
- For big changes in a class library, use **Introduce Local Extension**.





Code Smells

Introduce Foreign Method (Before)

```
class Report
{
    //...
    void SendReport()
    {
        DateTime nextDay = previousEnd.AddDays(1);
        //...
    }
}
```

Code Smells

Introduce Foreign Method (After)

```
class Report
{
    //...
    void SendReport()
    {
        DateTime nextDay = previousEnd.NextDay();
        //...
    }
}

public static DateTime NextDay(this DateTime dt)
{
    return dt.AddDays(1);
}
```

Code Smells

Introduce Local Extensions

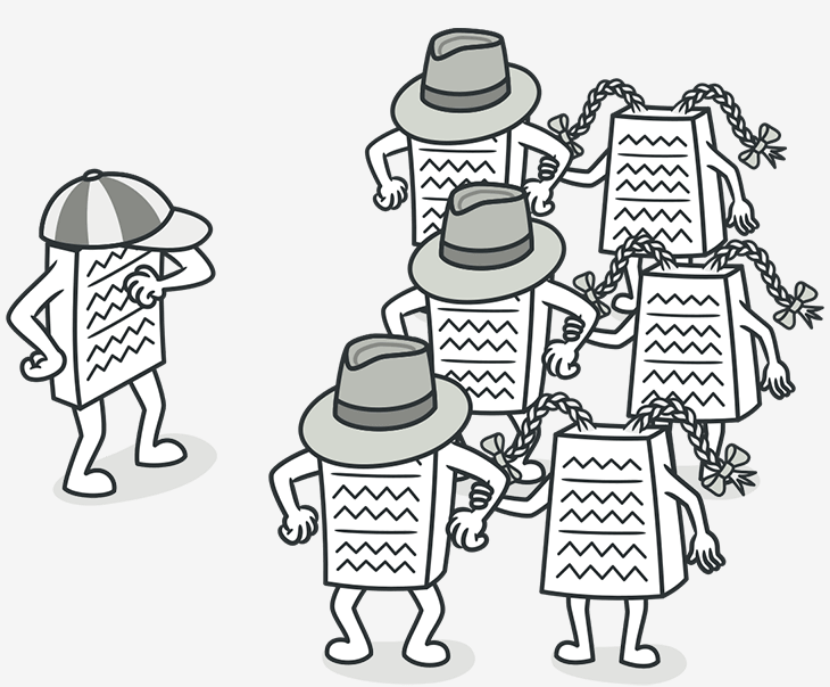
```
public static class DateTimeExtensions
{

    public static DateTime NextDay(this DateTime dt)
    {
        return dt.AddDays(1);
    }

    public static DateTime Yesterday(this DateTime dt)
    {
        return dt.AddDays(-1);
    }


}
```

Message Chains

- In code, you see a series of calls like this: A().B().C().D()…
- To delete a Chain, use **Hide Delegate**.
- Why the end object is being used?
- **Extract Method** for this functionality, and then
- Move it to the beginning of the chain, by using **Move Method**.

Middle Man

- If a class performs only one action, delegating work to another class, why does it exist at all?
- This smell can be the result of overzealous elimination of **Message Chains**.
- If most of a method's classes delegate to another class, Remove Middle Man is in order.



Resources

Resources

- Fowler, Martin (1999). Refactoring. Improving the Design of Existing Code. Addison-Wesley. ISBN 0-201-48567-2.
- Refactoring Guru
- Arsenovski, Danijel (2009). Professional Refactoring in C# and ASP.NET. Wrox. ISBN 978-0-470-43452-9.
- Ritchie, Peter (2010). Refactoring with Visual Studio 2010. Packt. ISBN 978-1-84968-010-3.