**What Is an AntiPattern?**

AntiPatterns, like their design pattern counterparts, define an industry vocabulary for the common defective processes and implementations within organizations. A higher-level vocabulary simplifies communication between software practitioners and enables concise description of higher-level concepts.

An AntiPattern is a literary form that describes a commonly occurring solution to a problem that generates decidedly negative consequences. The AntiPattern may be the result of a manager or developer not knowing any better, not having sufficient knowledge or experience in solving a particular type of problem, or having applied a perfectly good pattern in the wrong context.

AntiPatterns provide real-world experience in recognizing recurring problems in the software industry and provide a detailed remedy for the most common predicaments. AntiPatterns highlight the most common problems that face the software industry and provide the tools to enable you to recognize these problems and to determine their underlying causes.

Furthermore, AntiPatterns present a detailed plan for reversing these underlying causes and implementing productive solutions. AntiPatterns effectively describe the measures that can be taken at several levels to improve the developing of applications, the designing of software systems, and the effective management of software projects.



[**Software Development AntiPatterns**](https://sourcemaking.com/antipatterns/software-development-antipatterns)

A key goal of development AntiPatterns is to describe useful forms of software refactoring. Software refactoring is a form of code modification, used to improve the software structure in support of subsequent extension and long-term maintenance. In most cases, the goal is to transform code without impacting correctness.



[**Software Architecture AntiPatterns**](https://sourcemaking.com/antipatterns/software-architecture-antipatterns)

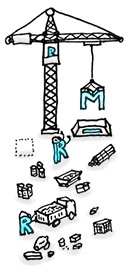
Architecture AntiPatterns focus on the system-level and enterprise-level structure of applications and components. Although the engineering discipline of software architecture is relatively immature, what has been determined repeatedly by software research and experience is the overarching importance of architecture in software development.



[**Software Project Management AntiPatterns**](https://sourcemaking.com/antipatterns/software-project-management-antipatterns)

In the modern engineering profession, more than half of the job involves human communication and resolving people issues. The management AntiPatterns identify some of the key scenarios in which these issues are destructive to software processes.

# Refactoring techniques



### [Composing methods](https://sourcemaking.com/refactoring/composing-methods)

Much of refactoring is devoted to correctly composing methods. In most cases, excessively long methods are the root of all evil. The vagaries of code inside these methods conceal the execution logic and make the method extremely hard to understand – and even harder to change.

The refactoring techniques in this group streamline methods, remove code duplication, and pave the way for future improvements.

* [**Extract Method**](https://sourcemaking.com/refactoring/extract-method)
* [**Inline Method**](https://sourcemaking.com/refactoring/inline-method)
* [**Extract Variable**](https://sourcemaking.com/refactoring/extract-variable)
* [**Inline Temp**](https://sourcemaking.com/refactoring/inline-temp)
* [**Replace Temp with Query**](https://sourcemaking.com/refactoring/replace-temp-with-query)
* [**Split Temporary Variable**](https://sourcemaking.com/refactoring/split-temporary-variable)
* [**Remove Assignments to Parameters**](https://sourcemaking.com/refactoring/remove-assignments-to-parameters)
* [**Replace Method with Method Object**](https://sourcemaking.com/refactoring/replace-method-with-method-object)
* [**Substitute Algorithm**](https://sourcemaking.com/refactoring/substitute-algorithm)

**Extract Method**

### Problem

You have a code fragment that can be grouped together.

**void** **printOwing**() {

printBanner();

// Print details.

System.out.println("name: " + name);

System.out.println("amount: " + getOutstanding());

}

### Solution

Move this code to a separate new method (or function) and replace the old code with a call to the method.

**void** **printOwing**() {

printBanner();

printDetails(getOutstanding());

}

**void** **printDetails**(**double** outstanding) {

System.out.println("name: " + name);

System.out.println("amount: " + outstanding);

}

### Why Refactor

The more lines found in a method, the harder it is to figure out what the method does. This is the main reason for this refactoring.

Besides eliminating rough edges in your code, extracting methods is also a step in many other refactoring approaches.

### Benefits

* More readable code! Be sure to give the new method a name that describes the method’s purpose: createOrder(), renderCustomerInfo(), etc.
* Less code duplication. Often the code that is found in a method can be reused in other places in your program. So you can replace duplicates with calls to your new method.
* Isolates independent parts of code, meaning that errors are less likely (such as if the wrong variable is modified).

### How to Refactor

1. Create a new method and name it in a way that makes its purpose self-evident.
2. Copy the relevant code fragment to your new method. Delete the fragment from its old location and put a call for the new method there instead.

Find all variables used in this code fragment. If they are declared inside the fragment and not used outside of it, simply leave them unchanged – they will become local variables for the new method.

1. If the variables are declared prior to the code that you are extracting, you will need to pass these variables to the parameters of your new method in order to use the values previously contained in them. Sometimes it is easier to get rid of these variables by resorting to [**Replace Temp with Query**](https://sourcemaking.com/refactoring/replace-temp-with-query).
2. If you see that a local variable changes in your extracted code in some way, this may mean that this changed value will be needed later in your main method. Double-check! And if this is indeed the case, return the value of this variable to the main method to keep everything functioning.

**Split Temporary Variable**

### Problem

You have a local variable that is used to store various intermediate values inside a method (except for cycle variables).

**double** temp = 2 \* (height + width);

System.out.println(temp);

temp = height \* width;

System.out.println(temp);

### Solution

Use different variables for different values. Each variable should be responsible for only one particular thing.

**final** **double** perimeter = 2 \* (height + width);

System.out.println(perimeter);

**final** **double** area = height \* width;

System.out.println(area);

**Remove Assignments to Parameters**

### Problem

Some value is assigned to a parameter inside method’s body.

**int** **discount**(**int** inputVal, **int** quantity) {

**if** (quantity > 50) {

inputVal -= 2;

}

// ...

}

### Solution

Use a local variable instead of a parameter.

**int** **discount**(**int** inputVal, **int** quantity) {

**int** result = inputVal;

**if** (quantity > 50) {

result -= 2;

}

// ...

}

### Why Refactor

The reasons for this refactoring are the same as for [**Split Temporary Variable**](https://sourcemaking.com/refactoring/split-temporary-variable), but in this case we are dealing with a parameter, not a local variable.

First, if a parameter is passed via reference, then after the parameter value is changed inside the method, this value is passed to the argument that requested calling this method. Very often, this occurs accidentally and leads to unfortunate effects. Even if parameters are usually passed by value (and not by reference) in your programming language, this coding quirk may alienate those who are unaccustomed to it.

Second, multiple assignments of different values to a single parameter make it difficult for you to know what data should be contained in the parameter at any particular point in time. The problem worsens if your parameter and its contents are documented but the actual value is capable of differing from what is expected inside the method.

### Benefits

* Each element of the program should be responsible for only one thing. This makes code maintenance much easier going forward, since you can safely replace code without any side effects.
* This refactoring helps to extract «repetitive code to separate methods» (Extract Method).

### How to Refactor

1. Create a local variable and assign the initial value of your parameter.
2. In all method code that follows this line, replace the parameter with your new local variable.

# Inline Method

### Problem

When a method body is more obvious than the method itself, use this technique.

**class** **PizzaDelivery** {

// ...

**int** getRating() {

**return** moreThanFiveLateDeliveries() ? 2 : 1;

}

**boolean** moreThanFiveLateDeliveries() {

**return** numberOfLateDeliveries > 5;

}

}

### Solution

Replace calls to the method with the method’s content and delete the method itself.

**class** **PizzaDelivery** {

// ...

**int** getRating() {

**return** numberOfLateDeliveries > 5 ? 2 : 1;

}

}

**Why Refactor**

A method simply delegates to another method. In itself, this delegation is no problem. But when there are many such methods, they become a confusing tangle that is hard to sort through.

Often methods are not too short *originally*, but become that way as changes are made to the program. So don’t be shy about getting rid of methods that have outlived their use.

**Benefits**

* By minimizing the number of unneeded methods, you make the code more straightforward.

**How to Refactor**

1. Make sure that the method is not redefined in subclasses. If the method is redefined, refrain from this technique.
2. Find all calls to the method. Replace these calls with the content of the method.
3. Delete the method.

**Software Development AntiPatterns**



Good software structure is essential for system extension and maintenance. Software development is a chaotic activity, therefore the implemented structure of systems tends to stray from the planned structure as determined by architecture, analysis, and design.

Software refactoring is an effective approach for improving software structure.

The resulting structure does not have to resemble the original planned structure.

The structure changes because programmers learn constraints and approaches that alter the context of the coded solutions. When used properly, refactoring is a natural activity in the programming process.

For example, the solution for the Spaghetti Code AntiPattern defines a software development process that incorporates refactoring. Refactoring is strongly recommended prior to performance optimization. Optimizations often involve compromises to program structure. Ideally, optimizations affect only small portions of a program. Prior refactoring helps partition optimized code from the majority of the software.

Development AntiPatterns utilize various formal and informal refactoring approaches. The following summaries provide an overview of the Development AntiPatterns found in this chapter and focus on the development AntiPattern problem. Included are descriptions of both development and mini-AntiPatterns. The refactored solutions appear in the appropriate AntiPattern templates that follow the summaries.

* [**The Blob**](https://sourcemaking.com/antipatterns/the-blob)  
  Procedural-style design leads to one object with a lion’s share of the responsibilities, while most other objects only hold data or execute simple processes. The solution includes refactoring the design to distribute responsibilities more uniformly and isolating the effect of changes.
* [**Continuous Obsolescence**](https://sourcemaking.com/antipatterns/continuous-obsolescence)  
  Technology is changing so rapidly that developers often have trouble keeping up with current versions of software and finding combinations of product releases that work together. Given that every commercial product line evolves through new releases, the situation is becoming more difficult for developers to cope with. Finding compatible releases of products that successfully interoperate is even harder.
* [**Lava Flow**](https://sourcemaking.com/antipatterns/lava-flow)  
  Dead code and forgotten design information is frozen in an ever-changing design. This is analogous to a Lava Flow with hardening globules of rocky material. The refactored solution includes a configuration management process that eliminates dead code and evolves or refactors design toward increasing quality.
* [**Ambiguous Viewpoint**](https://sourcemaking.com/antipatterns/ambiguous-viewpoint)  
  Object-oriented analysis and design (OOA&D) models are often presented without clarifying the viewpoint represented by the model. By default, OOA&D models denote an implementation viewpoint that is potentially the least useful. Mixed viewpoints don’t allow the fundamental separation of interfaces from implementation details, which is one of the primary benefits of the object-oriented paradigm.
* [**Functional Decomposition**](https://sourcemaking.com/antipatterns/functional-decomposition)  
  This AntiPattern is the output of experienced, nonobject-oriented developers who design and implement an application in an object-oriented language. The resulting code resembles a structural language (Pascal, FORTRAN) in class structure. It can be incredibly complex as smart procedural developers devise very “clever” ways to replicate their time-tested methods in an object-oriented architecture.
* [**Poltergeists**](https://sourcemaking.com/antipatterns/poltergeists)  
  Poltergeists are classes with very limited roles and effective life cycles. They often start processes for other objects. The refactored solution includes a reallocation of responsibilities to longer-lived objects that eliminate the Poltergeists.
* [**Boat Anchor**](https://sourcemaking.com/antipatterns/boat-anchor)  
  A Boat Anchor is a piece of software or hardware that serves no useful purpose on the current project. Often, the Boat Anchor is a costly acquisition, which makes the purchase even more ironic.
* [**Golden Hammer**](https://sourcemaking.com/antipatterns/golden-hammer)  
  A Golden Hammer is a familiar technology or concept applied obsessively to many software problems. The solution involves expanding the knowledge of developers through education, training, and book study groups to expose developers to alternative technologies and approaches.
* [**Dead End**](https://sourcemaking.com/antipatterns/dead-end)  
  A Dead End is reached by modifying a reusable component if the modified component is no longer maintained and supported by the supplier. When these modifications are made, the support burden transfers to the application system developers and maintainers. Improvements in the reusable component are not easily integrated, and support problems can be blamed upon the modification.
* [**Spaghetti Code**](https://sourcemaking.com/antipatterns/spaghetti-code)  
  Ad hoc software structure makes it difficult to extend and optimize code. Frequent code refactoring can improve software structure, support software maintenance, and enable iterative development.
* [**Input Kludge**](https://sourcemaking.com/antipatterns/input-kludge)  
  Software that fails straightforward behavioral tests may be an example of an input kludge, which occurs when ad hoc algorithms are employed for handling program input.
* [**Walking through a Minefield**](https://sourcemaking.com/antipatterns/walking-through-minefield)  
  Using today’s software technology is analogous to walking through a high-tech mine field. Numerous bugs are found in released software products; in fact, experts estimate that original source code contains two to five bugs per line of code.
* [**Cut-and-Paste Programming**](https://sourcemaking.com/antipatterns/cut-and-paste-programming)  
  Code reused by copying source statements leads to significant maintenance problems. Alternative forms of reuse, including black-box reuse, reduce maintenance issues by having common source code, testing, and documentation.
* [**Mushroom Management**](https://sourcemaking.com/antipatterns/mushroom-management)  
  In some architecture and management circles, there is an explicit policy to keep system developers isolated from the system’s end users. Requirements are passed second-hand through intermediaries, including architects, managers, or requirements analysts.