Lecture 5. Refactoring

Table of Contents

[1. General Idea 1](#_Toc185177019)

[2. ACTIVITIES IN A REFACTORING PROCESS 2](#_Toc185177020)

[2.1. Identify What to Refactor 2](#_Toc185177021)

[2.2. Determine Which Refactorings Should be Applied 3](#_Toc185177022)

[2.3. Ensure that Refactoring Preserves the Behavior of the Software 5](#_Toc185177023)

[2.4. Apply the Refactorings to the Chosen Entities 6](#_Toc185177024)

[2.5. Evaluate the Impacts of the Refactorings on Quality 7](#_Toc185177025)

[2.6. Maintain Consistency of Software Artifacts 7](#_Toc185177026)

[3. FORMALISMS FOR REFACTORING 7](#_Toc185177027)

[3.1. Assertions 8](#_Toc185177028)

[3.2. Graph Transformation 8](#_Toc185177029)

[3.3. Software Metrics 9](#_Toc185177030)

[4. MORE EXAMPLES OF REFACTORINGS 16](#_Toc185177031)

[5. INITIAL WORK ON SOFTWARE RESTRUCTURING 18](#_Toc185177032)

[5.1. Factors Influencing Software Structure 18](#_Toc185177033)

[5.2. Classification of Restructuring Approaches 19](#_Toc185177034)

[5.3. Restructuring Techniques 21](#_Toc185177035)

# General Idea

Restructuring means reorganizing software to give it a different look, or structure. Source code is restructured to improve some of its *non-functional* requirements, without modifying its *functional* requirements. For example, one may restructure source code to improve its *readability*, *extensibility*, *maintainability*, and *modularity*. Though restructuring does not modify the system’s functional requirements, it can take place while adding new features, that is, functional requirements, to the system.

Software restructuring activities should be seen as increasing the value of the software under consideration. While choosing a software restructuring approach it is important to define goals so that achievement of those goals should produce better software value. The cost of restructuring should be justified in terms of the expected gain in software value. Ideally, the expected gain is desired to be quantified. However, in the absence of widely available quantified data about the effectiveness of structuring approaches, one could use qualitative data. Some simple restructuring activities are

as follows:

* *Pretty printing:* Align code statements so that code becomes easier to understand as logical units.
* *Meaningful names for variables:* Variable names are chosen to give an indication of programming plans.
* *One statement per line:* Write one code statement in one line, as opposed to many statements in one line.

# ACTIVITIES IN A REFACTORING PROCESS

To restructure a software system, programmers follow a process with well defined activities. Those activities are as follows:

* Identify what to refactor.
* Determine which refactorings should be applied.
* Ensure that refactoring preserves the software’s behavior.
* Apply the refactorings to the chosen entities.
* Evaluate the impacts of the refactorings.
* Maintain consistency.

Next, we explain the above items one by one.

## Identify What to Refactor

In this step, the programmer identifies what to refactor from a set of software artifacts. Some examples of software artifacts that the programmer can consider are source code, design documents, and requirements documents. Having identified the top level artifact, the programmer can focus on specific portions of the chosen artefact for refactoring. Specific modules, functions, classes, methods, and data structures can be identified from the source code for refactoring. For programs written in non-object-oriented languages, restructuring is generally limited to the level of a function or a block of code. For programs written in object-oriented languages, the richness of the languages, namely, interfaces, dynamic binding, subtyping, overriding, and polymorphism, make restructuring difficult. The broad concept of *code smell* is applied to source code to detect where refactorings should be applied. A *code smell* is any symptom in the source code of a software that possibly indicates a deeper problem. Existence of code smells in source code do not imply the existence of problems in source code, such as, faults, low level of performance, and low level of reliability. On the other hand, what code smells do imply is that if the problems are not resolved sooner, then it is likely that future changes may introduce more faults and future changes may be more expensive to execute. Some examples of code smells are as follows.

* *Duplicate Code:* This smell occurs when segments of source code are repeated in many places in the program. If there is a need to change the duplicate code, it must be ensured that all segments are changed. If some segment is missed, which is likely to occur, then faults will be introduced. The solutions lie in the nature of code duplications. For example, if duplications occur in different methods of the same class, then duplicates can be extracted into a new method. On the other hand, if duplicate methods occur in subclasses, then the duplicate codes can be moved to the subclass. In simple cases, code duplication can be eliminated by introducing a new function and by inserting function calls. In complex cases, an intermediate subclass can be inserted to factor out the common code.
* *Long Parameter List:* When a method has too many formal parameters, say, more than four, programmers may make errors while designing calls to the method. A common error is to reorder the list of actual parameters. A solution may be found in designing a parameter object, thereby passing a single parameter instead of a long list of parameters.
* *Long Methods:* This occurs if a method has a long sequence of statements, say, hundreds of lines of code. A solution lies in extracting methods from long fragments of code.
* *Large Classes:* A smell is said to occur if a class has too many methods, say, more than 8, and too many variables, say, more than 15. A solution lies in splitting the class into component classes and creating super-classes.
* *Message Chain:* A message chain occurs when one calls several methods successively. An example of a message chain is: student.getID().getRecord().getGrade(course). Such a chain can be simplified by means of a helper function that performs part of the computation of another function. Thus, the above message chain can be rewritten as: student.getGrade(course).

At the design level, the entities that can be considered for refactoring are software architecture, global control flow, and database schemas. Class diagrams, state-chart diagrams, and activity diagrams are extensively used to describe various aspects of software design. Therefore, refactoring of software design involves manipulating those diagrams. To describe program structures at a very high level, designers apply design patterns. Therefore, refactoring of design can involve restructuring or replacing occurrences of poor design patterns in a legacy system with good design patterns.

## Determine Which Refactorings Should be Applied

In this step, the programmer identifies which refactorings to apply to the portions of the software identified in the aforementioned first step. For ease of understanding, in the following we give some examples of refactorings in the context of a class diagram shown in Figure 7.1.



The intention is to show what refactorings look like. The class diagram is about a local area network (LAN) simulator.

* *R1:* Rename method *print* to *process* in class *PrintServer*. Perform this refactoring together with *R2*.
* *R2:* Rename method *save* to *process* in class *FileServer*. Perform this refactoring together with *R1*. In *R1* and *R2*, the new names of *print and save* are the same, because *process* prepares for the application of refactoring *R4*.
* *R3:* Create a superclass called *Server* from *PrintServer* and *FileServer*, because their behaviors are very similar.
* *R4:* Pull up method *accept* from classes *PrintServer* and *FileServer* to the superclass *Server* created with *R3*. Applications of *R1* and *R2* essentially makes the two original versions of *accept* in *PrintServer* and *FileServer* identical.
* *R5:* Move method *accept* from class *PrintServer* to class *Packet*, because method *accept* directly accesses the field *receiver* in class *Packet*. An advantage of moving *accept* from *PrintServer* to class *Packet* is that data packets themselves will decide what actions to take.
* *R6:* Move method *accept* from class *FileServer* to class *Packet* for the same reason given for *R5*.
* *R7:* Encapsulate field *receiver* in class *Packet*, so that another class cannot directly access this field. The advantages are: (i) there is increased modularity of the system; and (ii) internal representation of data packets can be modified without modifying the classes that use data packets.
* *R8:* Add parameter *p* of type *Packet* to method *print* in class *PrintServer* so that the contents of a packet can be printed.
* *R9:* Add parameter *p* of type *Packet* to method *save* in class *FileServer* so that the contents of a packet can be saved.

From the class diagram shown in Figure 7.1 and the nine refactorings *R1 – R9* it is apparent that one can design a large number of refactorings even for a small system. A subset of the entire set of refactorings must be carefully chosen, because of the following reasons.

* *Some refactorings must be applied together.* For example, refactorings *R1* and *R2* are applied together. It is of no use to apply only one of them. If both of them are not applied together, then *R4* cannot be applied, because applying just one of them or not applying them at all will not make method *accept* identical in both classes *FileServer* and *PrintServer*.
* *Some refactorings must be applied in certain orders.* For example, refactorings *R1* and *R2* must precede *R3*. One can apply *R3* only after applying *R1* and *R2*, because applications of *R1* and *R2* make the methods *accept* in classes *FileServer* and *PrintServer* identical. In other words, *R3* cannot be applied if *R1* and *R2* have not yet been applied.
* *Some refactorings can be individually applied, but they must follow an order if applied together.* For example, refactorings *R1* and *R8* can be applied in isolation. However, if a programmer chooses to apply both, then *R8* must occur before *R1*.
* *Some refactorings are mutually exclusive.* For example, refactorings *R4* and *R6* are mutually exclusive. Refactoring *R4* pulls up methods *accept* from classes *FileServer* and *PrintServer* into the superclass *Server*, whereas *R6* moves the method *accept* to class *Packet*. It is clear that one cannot apply both the refactorings together.

For large sets of refactorings, tool support is needed to identify a feasible subset of refactorings. The following two techniques can be used to analyze a set of refactorings to select a feasible subset.

* *Critical pair analysis:* Here the idea is to identify pairs of mutually exclusive refactorings. Given a set of refactorings, analyze each pair of refactorings for conflicts. A pair of refactorings is said to be conflicting if both of them cannot be applied together. For example, *R4* and *R6* constitute a conflicting set, which means that one cannot be applied after applying the other.
* *Sequential dependency analysis:* Sequential dependency of refactorings means that: (i) in order to apply a refactoring, one or more refactorings must have been applied before; and (ii) if one refactoring has already been applied, a mutually exclusive refactoring cannot be applied anymore. For example, after applying *R1*, *R2*, and *R3*, refactoring *R4* becomes applicable. On the contrary, if *R4* is applied, then *R6* is not applicable anymore.

## Ensure that Refactoring Preserves the Behavior of the Software

Ideally, the behavior of a program after refactoring should be the same as the behaviour before refactoring. Program behavior originally simply referred to input–output behavior. In other words, for the same set of input values, the programs before refactoring and after refactoring were desired to produce the same output values. However, in many applications preservation of input–output behavior alone is not enough, because preservation of temporal constraints and non-functional requirements of the program may be key to the success of refactoring. A non-exclusive list of such non-functional requirements is as follows:

* *Temporal constraints:* A temporal constraint over a sequence of operations is that the operations occur in a desired order. For real-time applications, refactoring should preserve temporal constraints.
* *Resource constraints:* Memory, energy, and communication bandwidth are some examples of critical resources on some computers. Therefore, it is important that the software after refactoring does not demand more of those resources than what the software before refactoring demanded.
* *Safety constraints:* It is important that a software does not lose its safety properties after it is refactored.

Therefore showing that a refactored program behaves the same ways as the program before refactoring is a difficult task. Two pragmatic ways of showing that a refactored program behaves the same way as the original program are as follows:

* *Testing:* By means of extensive testing, observe the behavior of the program before and after refactoring to determine whether or not there is behavior preservation. However, it may be noted that refactoring may invalidate some tests that were designed based on the structure of the program.
* *Verification of preservation of call sequence:* The concept of *call preservation* means that all method calls are preserved in the refactored program. This is a slightly more formal, but still weak, way of showing that refactoring preserves behavior. In a further limited way, the type correctness of a sequence of calls can be preserved by using type constraints to verify the preconditions of refactorings and determining what source code to modify.

## Apply the Refactorings to the Chosen Entities

This means executing the steps of the refactorings chosen before. The class diagram of Figure 7.2a has been obtained from Figure 7.1 by **focusing** on the classes *FileServer*, *PrintServer*, and *Packet*, and applying refactorings *R1*, *R2*, and R3. Next, the class diagram of Figure 7.2b has been obtained by applying *R4* to the class diagram of Figure 7.2a. Similarly, the class diagram of Figure 7.2c has been obtained by applying *R6* to the class diagram of Figure 7.2a.





From Softw Syst Model (2007) 6:269–285, DOI 10.1007/s10270-006-0044-6, Analysing refactoring dependencies using graph transformation, T. Mens, G. Taentzer, O. Runge

## Evaluate the Impacts of the Refactorings on Quality

Both *internal* qualities and *external* qualities are impacted by refactorings. Some examples of internal qualities are *size*, *complexity*, *coupling*, *cohesion*, and *testability*. Similarly, some examples of external qualities are *performance*, *reusability*, *maintainability*, *extensibility*, *robustness*, and *scalability*. In general, refactoring techniques are highly specialized, which means that one technique is intended to improve a small number—generally one—of quality attributes of the program. It is important to note that, refactorings directly impact internal qualities. Therefore, by measuring the impact of refactorings on internal qualities, their impacts on external qualities can be estimated. Some examples of software metrics are *coupling*, *cohesion*, and *size*. Decreased coupling, increased cohesion, and decreased size are likely to make a software system more maintainable. Therefore, to assess the impact of a refactoring technique for better maintainability, one can evaluate the metrics before refactoring and after refactoring, and compare them.

## Maintain Consistency of Software Artifacts

A software system is described by many artifacts at different levels of abstractions. Those artifacts include requirements documents, design documents, source code, and test suites. If one kind of artifact is changed, then it is important to change some or all of the other artifacts so that consistency is maintained across the artifacts. For example, changes in source code may require changes in the design documents and the test suites. Therefore, the concept of *change propagation* is used to cope with inconsistencies across different software artifacts. The concept of change propagation has been explained in Chapter 6 Impact analysis.

# FORMALISMS FOR REFACTORING

In this section, we explain three key formalisms for refactoring: assertions, graph transformation, and metrics. Assertions are useful in verifying the assumptions made by programmers. The concept of graph transformation is useful in viewing refactorings as applications of transformation rules. The concept of metrics is useful in quantifying to what extent the internal and external properties of software entities have changed as a result of applying refactorings.

## Assertions

Programmers make assumptions about the behavior of programs at specific points of their interests, and those assumptions can be tested by means of assertions. Thus, an *assertion* is specified as a Boolean expression which evaluates to *true* or *false*. When an assertion is put at a certain point of execution in a program, the programmer thinks that the assertion always evaluates to true at that point. That is, a programmer can use an assertion to test their assumptions about the program at the point where the assertion occurs. If the assertion evaluates to true, normal execution of the program continues. On the other hand, if the assertion evaluates to false, then it is an indication of something gone wrong in the computation process. Different execution semantics can be associated with assertions, when they evaluate to false. For example, when the assertion fails, program execution is halted and a detailed message can be displayed. There are three kinds of commonly understood assertions, namely, *invariants*, *preconditions*, and *post-conditions*. An invariant is a Boolean expression that the programmer always expects to evaluate to true. In other words, an invariant evaluates to true wherever in the program it is invoked. The concept of a *class invariant* is a special case of the general concept of invariant. A class invariant is a condition that all instances of that class must satisfy. A precondition is a condition that must be satisfied *before* a computation, whereas a post-condition is a condition that must be satisfied *after* a computation. Behavior preservation is a key requirement of refactoring and restructuring techniques. That is, the input–output behavior of a program must remain unchanged even after changes in a program’s structure due to the applications of refactoring or restructuring techniques. Invariants, preconditions, and post-conditions have been suggested by researchers to address the problem of behavior preservation. An example of invariant is: *All instance variables of a class, whether defined or inherited, have distinct names. Similarly, all methods of a class, whether defined or inherited, must have distinct names.* An obvious problem with the use of assertions in testing the behavior preserving property of refactoring and restructuring techniques is the computationally expensive static checking of preconditions, post-conditions, and invariants.

## Graph Transformation

Programs and design diagrams, namely, class diagrams and statecharts, can be viewed as *graphs*, and refactorings can be viewed as graph production rules. Therefore, applying refactorings to software can be viewed as applying graph transformations. Software entities, namely, classes (C), method signatures (M), block structures (B), variables (V), parameters (P), and expressions (E) are represented by *typed nodes* in a graph. The possible relationships among the nodes are: method lookup (l), inheritance (i), membership (m), (sub)type (t), expression (e), actual parameter (ap), formal parameter (fp), cascaded expression (∙), call (c), variable access (a), and update (u).

From *B. Hoffmann et al. / Electronic Notes in Theoretical Computer Science 152 (2006) 53–67,* Cloning and Expanding Graph Transformation Rules for Refactoring

If the transformation is local, i.e. it has an effect on only a small part of the program graph, then it makes sense to express it as a rule. This is typically the case for refactorings. A concrete instance of the push-down-method refactoring is shown in Fig. 2. The body of method originate is copied from its containing class (Node) to its subclasses (Workstation and PrintServer). The method body, containing a call to a method send, is represented by a simplified syntax tree on the left-hand side of the rule, consisting of the node of type **B** and two nodes of type **E**. This subgraph occurs twice on the right-hand side. The concrete class and method names identify the nodes in the rule’s interface. When applied, the left hand side of this rule is matched against the program graph, and then the rule removes the syntax tree in the left hand side (the shaded part, which is not in the interface) and adds two copies of it as part of the two subclasses.



The *push-down method* refactoring is explained as follows. Assume that there is a superclass A and two subclasses X and Y designed to inherit A. Also assume that there is a method m defined in A. If m is not further redefined in X and Y, then method m is available in both X and Y. Note that if there is no need to use method m in subclass Y, there is no need to push it down to Y. Another way of accessing m in X and Y is to push m down to X and Y.

## Software Metrics

Software metrics can be used to quantitatively represent the internal and external qualities of software. In this subsection, we present the details of calculating two kinds of metrics, namely, cohesion and coupling. In general, a module consists of several components, with each component providing a defined functionality used by components within the same module and components within other modules. Therefore, it is useful to measure the strength of togetherness of components within a module so that one can decide whether or not some components should stay in the same module. The aforementioned concept of the strength of togetherness of components in the same module is expressed by means of cohesion metrics. On the other hand, the strength of dependency between modules is expressed by means of coupling metrics.

***Cohesion Metrics*** Simon et al., 2001, *Metrics Based Refactoring*. have introduced the concept of a *distance based* metric to express design cohesion, where cohesion refers to the degree to which module components belong together. The distance-based metric is explained in what follows. Let *B* be a set of considered properties for a special *similarity viewpoint*. Also, let *x* and *y* denote two entities (e.g., methods and attributes) of a “module” (e.g., a class) for which we are interested in finding its cohesion. Then, the *distance* between *x* and *y* with respect to the considered property set *B*, denoted by *distB*(*x*, *y*) is computed as follows:

*distB*(*x*, *y*) = 1 −|*p*(*x*) ∩ *p*(*y*)|/|*p*(*x*) ∪ *p*(*y*)| (7.1)

where *p*(*x*) = {*pi* ∈ *B*| *x* possesses property *pi*}.

For a method *f*, the set of its properties, denoted by *Bf* , is given as follows:

*Bf* = {*f* ∪ all methods directly used by*f* ∪ all attributes directly used by *f* }*.* (7.2)

Similarly, for an attribute *g*, the set of its properties, denoted by *Bg*, is given as follows:

*Bg* = {*g* ∪ all methods using *g*}*.* (7.3)

For the calculation of distance between two entities, the needed *B* is given by the union of the two corresponding sets of attributes. For example, if we are interested in calculating the distance between two methods *f*1 and *f*2, we have *B* = *Bf*1 ∪ *Bf*2.

Similarly, if we are interested in the distance between a method *f* and an attribute *g*, then we have *B* = *Bf* ∪ *Bg*.

Now, given a class *C*, let *M* = {*m*1,…,*mk*} be the set of methods and *A* = {*a*1,…,*mn*} be the set of attributes of *C*. Using the distance-based metric of Eq. 7.1, one can calculate the distance between all pairs of entities in the set *M* ∪ *A*, and plot *C* in a graphical manner such that each attribute is represented by a square and each method by a circle such that the Euclidean distance between two entities is equal to their distance calculated using Eq. 7.1. Such a notation to represent a class is called a Virtual Reality Modeling Language (Simon et al.,2001). In Figure 7.6, we show the VRML diagram of two classes C1 and C2. Though method m1 is a part of class C1, it is closer to the methods and attributes of class C2 than to the methods and attributes of class C1. Therefore, the *Move*-method refactoring can be applied to method m1 so that it becomes a part of class C2. The idea of *Move*-method refactoring has been explained by means of refactoring *R5* in Section 7.2.2.



From Simon, 2001, *Metrics Based Refactoring,* doi: 10.1109/CSMR.2001.914965

Consider the following code of two classes:



The location of the method methodB1( ) has a very “bad smell” because it uses features of class-A only. Thus, the *move method* refactoring should be applied to this method. To automate the identification of this problem the distances between each two entities (six methods and four attributes) are calculated. The resulting distances can be displayed in a so called *distance matrix*

**

$$B\_{mA1}=\left\{mA1,mA2,aA1\right\}, B\_{mA2}=\left\{mA2,aA1,aA2\right\}, dist\left(mA1,mA2\right)=1-\frac{\left|B\_{mA1}∩B\_{mA2}\right|}{\left|B\_{mA1}∪B\_{mA2}\right|}=1-\frac{\left|\left\{aA1,mA2\right\}\right|}{\left|\left\{mA1,mA2,aA1,aA2\right\}\right|}=1-\frac{2}{4}=0.5$$

$$B\_{aB2}=\left\{aB2,mB2\right\}, B\_{aB1}=\left\{aB1,mB2,mB3\right\}, dist\left(aB2,aB1\right)=1-\frac{\left|B\_{aB2}∩B\_{aB1}\right|}{\left|B\_{aB2}∪B\_{aB1}\right|}=1-\frac{\left|\left\{mB2\right\}\right|}{\left|\left\{aB1,aB2,mB2,mB3\right\}\right|}=1-\frac{1}{4}=0.75$$

The visualisation in Figure 1 strongly indicates in a ery simple and understandable way the recommendation to move methodB1 (indicated by the arrow) from class-B to class-A. Only in very few cases it would make sense to keep such a construction unchanged. Because the developer is the last authority, this still would be possible. This visualisation only should support the analysis of the system.

**

***CouplingMetrics*** The idea of coupling presented here was introduced by Kataoka, Imai, Andou, and Fukaya Y. (Kataoka, T. Imai, H. Andou and T. Fukaya, "A quantitative evaluation of maintainability enhancement by refactoring," International Conference on Software Maintenance, 2002. Proceedings., Montreal, QC, Canada, 2002, pp. 576-585, doi: 10.1109/ICSM.2002.1167822). First, three basic kinds of couplings are explained: *return value coupling*, *parameter coupling*, and *shared variable coupling*. Next, the three kinds of couplings are combined to obtain an overall, composite coupling metric.

* *Return value coupling:* Let there be two methods *A* and *B*. There exists a return value coupling between *A* and *B* if *A calls B* and *B returns* a value to *A*. Similarly, if *B* calls *A* and *A* provides a return value to *B*, then there also exists return value coupling between *A* and *B*. The quantity of return value coupling of a method *A*, denoted by *Crv*(*A*), with all the methods it calls and all the methods it is called from is computed as follows.



The idea behind *Krv*(*m*) is to consider the likelihood that inter-class coupling by means of value passing can lead to higher maintenance cost than intra-class coupling.

* *Parameter coupling:* Parameter coupling is about the number of parameters passed when a method calls another method. Method *A* has *n parameter coupling* with method *B* if method *A* receives *n* parameters from *B*. Similarly, if *A* passes *n* parameters to method *C*, then there exists *n parameter coupling* with *C*. The quantity of parameter coupling of a method *A*, denoted by *Cpp*(*A*), with all the methods it calls and all the methods it is called from is computed as follows.

*Cpp*(*A*) =Σ*m𝜁*∈*𝜁*(*A*)*Kpp*(*m𝜁* )*pm𝜁* +Σ*m𝜉*∈*𝜉*(*A*)*Kpp*(*m𝜉* )*pm𝜉*, (7.5)

where,

*𝜁* (*A*) is the set of methods that call *A*.

*𝜉* (*A*) is the set of methods that are called by *A*.

|  |
| --- |
| **Greek alphabet** |
|

|  |  |  |  |
| --- | --- | --- | --- |
| Αα | [Alpha](https://en.wikipedia.org/wiki/Alpha) | Νν | [Nu](https://en.wikipedia.org/wiki/Nu_%28letter%29) |
| Ββ | [Beta](https://en.wikipedia.org/wiki/Beta) | Ξξ | [Xi](https://en.wikipedia.org/wiki/Xi_%28letter%29) |
| Γγ | [Gamma](https://en.wikipedia.org/wiki/Gamma) | Οο | [Omicron](https://en.wikipedia.org/wiki/Omicron) |
| Δδ | [Delta](https://en.wikipedia.org/wiki/Delta_%28letter%29) | Ππ | [Pi](https://en.wikipedia.org/wiki/Pi_%28letter%29) |
| Εε | [Epsilon](https://en.wikipedia.org/wiki/Epsilon) | Ρρ | [Rho](https://en.wikipedia.org/wiki/Rho) |
| Ζζ | [Zeta](https://en.wikipedia.org/wiki/Zeta) | Σσς | [Sigma](https://en.wikipedia.org/wiki/Sigma) |
| Ηη | [Eta](https://en.wikipedia.org/wiki/Eta) | Ττ | [Tau](https://en.wikipedia.org/wiki/Tau) |
| Θθ | [Theta](https://en.wikipedia.org/wiki/Theta) | Υυ | [Upsilon](https://en.wikipedia.org/wiki/Upsilon) |
| Ιι | [Iota](https://en.wikipedia.org/wiki/Iota) | Φφ | [Phi](https://en.wikipedia.org/wiki/Phi) |
| Κκ | [Kappa](https://en.wikipedia.org/wiki/Kappa) | Χχ | [Chi](https://en.wikipedia.org/wiki/Chi_%28letter%29) |
| Λλ | [Lambda](https://en.wikipedia.org/wiki/Lambda) | Ψψ | [Psi](https://en.wikipedia.org/wiki/Psi_%28Greek%29) |
| Μμ | [Mu](https://en.wikipedia.org/wiki/Mu_%28letter%29) | Ωω | [Omega](https://en.wikipedia.org/wiki/Omega) |

 |

*Kpp*(*m*) is the parameter coupling inter-class coefficient. *Kpp*(*m*) = 1 if both methods *m* and *A* are in the same class. On the other hand, if *m* and *A* are in different classes, then *Kpp*(*m*) = *𝜅pp*, where *𝜅pp >* 1.

*Pm𝜁* is the number of parameters received by method *m𝜁* when *m𝜁* is called by another method. *Pm𝜉* is the number of parameters passed by method *m𝜉* when *m𝜉* calls another method.

The idea behind *Kpp*(*m*) is to consider the likelihood that inter-class coupling by means of parameter passing can lead to higher maintenance cost than intra-class coupling.

* *Shared variable coupling:* Two methods *A* and *B* belonging in the same class are said to have *n shared variable coupling* if they use *n* class or instance variables. The quantity of shared variable coupling of method *A*, denoted by *Csv*(*A*), with all the methods in the same class is computed as follows.

*Csv*(*A*) =Σ*m𝜒*∈*𝜒*(*A*)*Ksv*(*m𝜒* )*vm𝜒*, (7.6)

where, *𝜒* (*A*) is the set of methods that use class or instance variables in common with *A*. *vm* is the number of class or instance variables that are used both in *m* and *A*.

*Ksv*(*m*) is the shared variable inter-class coefficient. *Ksv*(*m*) = 1 if both methods *m* and *A* are in the same class. On the other hand, if *m* and *A* are in different classes, then *Ksv*(*m*) = *𝜅sv*, where *𝜅sv >* 1. The motivation for introducing *Ksv*(*m*) is similar to the motivation for introducing *Krv*(*m*).

Making decisions about the maintainability of a program from three different coupling perspectives is a difficult task. Therefore, it is useful to combine the three coupling metrics to obtain a single, composite coupling metric. While combining the three coupling metrics, it is useful to note that all the three metrics are not equally significant. For example, from the standpoint of maintainability of a program, the influence of accessing an instance variable from another class is arguably more severe than the influence of receiving a parameter from a method within the same class.

The concept of *weighted sum* is a simple strategy to combine the three coupling metrics. Three weighting factors, namely, *Wrv*, *Wpp*, and *Wsv* are introduced to represent the relative importance of *Crv*(*A*), *Cpp*(*A*), and *Csv*(*A*), respectively, such that the following constraints, denoted by Eqs. 7.7 and 7.8, hold:

*Wrv* + *Wpp* + *Wsv* = 1*.* (7.7)

0 *< Wrv* ≤ *Wpp* ≤ *Wsv.* (7.8)

Now, the total coupling represented by a single metric can be given by Eq. 7.9:

*CT* (*A*) = *WrvCrv*(*A*) + *WppCpp*(*A*) + *WsvCsv*(*A*)*.* (7.9)

In their experiments, Kataoka et al. have used the following values of the different parameters and weighting factors:

*𝜅rv* = 1*.*5, *𝜅pp* = 2*.*0, *𝜅sv* = 3*.*0

*Wrv* = 0*.*2,*Wpp* = 0*.*2,*Wsv* = 0*.*6*.*

It is important to note that applications of good refactorings should lead to the reduction in the value of *CT* (*A*).

From Y. Kataoka, T. Imai, H. Andou and T. Fukaya, "A quantitative evaluation of maintainability enhancement by refactoring," International Conference on Software Maintenance, 2002. Proceedings., Montreal, QC, Canada, 2002, pp. 576-585, doi: 10.1109/ICSM.2002.1167822.

We will show the example of refactoring effect quantification using a refactoring called “Move Method.” “Move Method” is supposed to be applied when “a method is, or will be, using or used by more features of another class than the class on which it is defined.” Figs. 2 and 3 show program codes before and after the “Move Method” respectively.





These programs are for the bank account management. Class AccountType deals with various types of bank account such as a regular account, a premium account and some others. In this case we assume that there are going to be several new account types, each of which has its own rule for calculating the overdraft charge. Therefore we plan to move overdraftCharge() method to the Account-Type class from Account class.

We can expect that the coupling of overdraftCharge()decreases after the refactoring. Let *n* be a number of account types dealt with AccountType class. Before the refactoring, overdraftCharge()provides its return value to bankCharge() and uses the return values of *n* account type decision methods such as AccountType.isPremium(). In other words, the coupling of overdraftCharge() consists of one intra–class return value coupling and *n* inter–class return value couplings. Therefore the whole coupling of overdraftCharge() is *Wrv*(1 + *n\*κrv*).

After the refactoring, on the other hand, overdraftCharge() provides its return value to Account.bankCharge(), receives a parameter from Account. bankCharge(), and uses the return values of *n* account type decision methods such as isPremium(). In other words, the coupling of overdraftCharge() consists of one inter–class return value coupling, *n* intra–class return value couplings, and one intra–class parameter coupling. Therefore the whole coupling of overdraftCharge()is *Wrv* (*κrv* + *n*) + *Wpp* .

The refactoring effect is then calculated as follows:

*Wrv* (1 + *n\*κrv*) *− {Wrv* (*κrv* + *n*) + *Wpp}*

= *Wrv* (*κrv −* 1)(*n −* 1) *− Wpp ,*

and this value is positive under the following condition:

(*κrv −* 1)(*n −* 1) *>Wpp/Wrv*($\geq $1)*.*

Suppose *κrv >* 1*.*5, which means that it is at least 1.5 times more costly to maintain inter–class coupling than intra–class coupling. If *κrv >* 1*.*5, then the refactoring effect is positive when *n ≥* 3, which means this refactoring is effective if there are more than two account types involved and that is what we have assumed in this example.

# MORE EXAMPLES OF REFACTORINGS

* *Substitute Algorithm:* A programmer may decide to replace an existing algorithm X in the code with a new algorithm Y for various reasons: (i) implementation of algorithm Y is clearer (i.e., it takes less time to understand) than the implementation of algorithm X; (ii) algorithm Y performs better than algorithm X; and (iii) standardization bodies have recommended to replace algorithm X with algorithm Y. Algorithm substitution can be easily applied if both the algorithms have the same input–output behaviors.
* *Replace Parameter with Method:* Consider the following code segment, where the method bodyMassIndex has two formal parameters.

int person;

:

// person is initialized here;

:

int bodyMass = getMass(person);

int height = getHeight(person);

int BMI = bodyMassIndex(bodyMass, height);

:

The above code segment can be rewritten such that the new bodyMassIndex method accepts one formal parameter, namely, person, and internally computes the values of bodyMass and height:

int person;

:

// person is initialized here;

:

int BMI = bodyMassIndex(person);

:

The advantage of this refactoring is that it reduces the number of parameters passed to methods. Such reduction is important because one can easily make errors while passing long parameter lists.

* *Push-down Method:* Assume that Executive and Clerk are two subclasses of the superclass Employee as shown in Figure 7.7a. Method overTimePay has been defined in class Employee. However, if the overTimePay method is used in the Clerk class, but not in the Executive class, then the programmer can push down the overTimePay method to the Clerk class as illustrated in Figure 7.7b. 
* *Parameterize Methods:* Sometimes programmers may find multiple methods performing the same computations on different input data sets. Those methods can be replaced with a new method with additional formal parameters, as illustrated in Figure 7.8. In Figure 7.8a, we have the Communication class with four methods: bluetoothInterface, wifiInterface, threeGInterface, and fourGInterface. In Figure 7.8b, we have the Communication class with just one method, namely, wirelessInterface with one parameter, namely, radio. The method wirelessInterface can be invokedwith different values of radio so that the wirelessInterface method can in turn invoke different radio interfaces.
* 

# INITIAL WORK ON SOFTWARE RESTRUCTURING

The concept of software restructuring dates back to the mid 1960s, almost as soon as programs were written in Fortran. In this section, we explain the factors that influence software structure, classification of early restructuring approaches, and some widely studied early restructuring techniques.

## Factors Influencing Software Structure

Before we discuss restructuring approaches, it is useful to have a broad understanding of *software structure.* Software structure is a set of attributes of the software such that the programmer gets a good understanding of software. Therefore, any *factor* that can influence the state of software or the programmer’s perception might influence software structure. One view of the factors that influence software structure has been shown in Figure 7.9.



In the following, we explain the factors one by one.

* *Code:* Undoubtedly, the source code has the biggest influence on software structure. Code quality and style at all levels of details, namely, variables, constants, statement, function, and module, play key roles in understanding code. For example, adherence to coding standards significantly increases the readability of code. Similarly, adoption of common architectural styles enhances code understanding.
* *Documentation:* There are two kinds of documentations associated with source code: (i) in-line documentation of source code; and (ii) external documentations, namely, requirements documents, design documents, user manuals, and test cases. Often the in-line documentations determine the programmer’s perception of code structure. Programmers’ perception of software structure is influenced by clearly written, easily referenced, complete, accurate, and up-to-date documentations.
* *Tools—Programming environment:* Development tools can help programmers better understand source code. Tools can assist programmers trace through the source code to understand dynamic behavior, animation can help in understanding the dynamic strategy used in an algorithm, and cross-referencing of global variables reveal the interactions among modules. In addition, tools can reformat code for better readability via pretty printing, highlight keywords, and take advantage of color coding of source code. For example, comments can be displayed in one color and executable code in a different color.
* *Programmers:* Qualities of programmers influence their perception of software structure. Examples of programmer qualities are individual capabilities, education, experience, training, and aptitude. Happenings in their personal lives too can influence their perception of software structure.
* *Managers and policies:* Management can play an influencing role in having a good initial structure and sustain, or even improve, the initial structure by means of designing policies and allocating resources. Management can design general policies about means, such as adhering to standards, of achieving software qualities. Similarly, managers can influence the practice of achieving good software structures by tying the annual performance review of programmers with their adherence to those standards.
* *Environment:* This factor refers to the general working environment of programmers, including the physical facilities and availability of resources when needed.

All the factors shown in Figure 7.9 influence software structure, to varying degrees. For example, source code has more influence on restructuring than working environment of programmers. Consequently, approaches that influence any factor in Figure 7.9 can be applied to software restructuring. In the following section, we present some well-known structuring approaches found in the literature.

## 5.2. Classification of Restructuring Approaches

A broad classification of software restructuring approaches has been shown in Figure 7.10, and explained in the following.

* + *Approaches not involving code changes:* There are several software restructuring approaches that do not involve making changes to the existing software. These approaches are as follows:
	+ Train programmers: Programmers may be trained in structured programming and software engineering, including software architectural styles and modularization techniques
	+ Upgrade documentation: In-line comments in source code can be made more accurate and readable. Cryptic comments can be expanded to explain the rationale behind decisions, and what alternatives had been explored. Comments can be updated to reflect changes in the code. Similarly, external documentations can be updated to make them consistent with the code, accurate, and complete. Incomplete and inconsistent documentations constantly frustrate programmers.



* *Approaches involving code changes:* A large majority of the approaches involve making changes to source code in the form of writing new code, making changes to existing code, deleting code, reformatting code, and moving code chunks within and between modules. These approaches can be further divided into three major categories as follows:
* Practices: Some examples of software restructuring practices are: (i) restructuring code with pre-processors; (ii) making code understandable by means of inspection and walkthroughs; (iii) formatting code by means of adhering to programming standards and style guidelines; and (iv) restructuring code for reusability.
* Techniques: Some software restructuring approaches are based on defined techniques. Some examples of software restructuring techniques are: (i) incremental restructuring; (ii) goto-less approach; (iii) case-statement approach; (iv) Boolean flag approach; and (v) clustering approach.
* Tools: Many tools have been designed for software restructuring since the late 70s. Some example tools are *Eclipse IDE* (Integrated Development Environment), *IntelliJ IDEA*, *jFactor*, *Refactorit*, and *Clone Doctor*. Some early day restructuring tools were *Refactoring Browser* for *Smalltalk*, *Moose Refactoring Engine*, *CStructure* for the *C* language, a prototype tool for *Oberon*, and Griswold’s tool for *Scheme*.

## Restructuring Techniques

In this section, we explain several restructuring techniques developed in the mid-1970s, before the time of object-oriented programming. The techniques are applied at different levels of abstractions: reorganization and rewriting of source code to eliminate goto statements, application of the concept of information hiding to C programs, wrapping of a highly unstructured system with newly written front ends

and back ends, and remodularization of software with clustering.

***A. Elimination-of-*goto *Approach*** An important feature of structured programming is that it puts emphasis on the following control constructs: *for*, *while*, *until*, and *if-then-else*. It is easy to understand code with such constructs, because those constructs make occurrences of loop and branching of control clear. Before the onset of structured programming in the 70s, much source code had been written with *goto* statements. A *goto* statement is an unconditional jump statement that can be found in several high level programming languages, namely, Fortran, Cobol, and C. It is difficult to understand the control flow in programs with many *goto* statements in them. It was shown that every flowchart program with *goto* statements can be transformed into a functionally equivalent *goto*-less program by using *while* statements. They introduce new Boolean variables to keep track of information about the sequence of the computation. The resulting program has the same order of efficiency as the original program. Though it is easy to understand a flowchart program, it is implemented with *goto* statements to represent the unconditional jumps in the flowchart. There was developed an algorithm to transform a flow-graph into a program with *repeat* (*do-forever*), *if-then-else*, *break*, and *next* statements. The *break* statement causes a jump out of the enclosing *repeat* statement, whereas the *next* statement causes a jump to the next iteration of an enclosing *repeat*. The goal of this algorithm is to produce understandable programs, rather than completely eliminate the use of *goto* statements. The design of the algorithm is based on the idea of *properly nested* programs, where *repeat* statements reflect iteration in the program and *if-then-else* statements reflect branching and merging of control flow. The restructured program contains *goto* statements if no other available control construct describes the flow of control. The algorithm is central to the implementation of a tool called *STRUCT*, which translates Fortran programs into programs in *RATFOR*. *RATFOR* is an extended Fortran language that includes *while*, *if-then-else*, *break*, and *next*. The programs produced by *RATFOR* are more readable than the original Fortran programs.

***B. Localization and Information Hiding Approach*** *Localization* and *information* hiding are well-known software engineering principles that can be applied to design good quality software. As the name suggests, localization is the process of collecting the logically related computational resources in one physical module. Functions, procedures, operations, and data types are examples of computational resources in an imperative programming language, such as C. As a result of localizing computational

resources into separate modules, programmers can restructure a program into a loosely coupled system of sufficiently independent modules. In the C and Fortran language, localization is difficult to achieve for the following reasons.

* A variable of a function may be referred by the *extern* and *include* statements and imported and exported to other program modules.
* Data sharing and relations among functions are not explicitly represented in source code.

By means of information hiding, one can suppress (i.e., hide) the details of implementations of computational resources, thereby enabling programmers to focus on high level concepts, which make it easier to understand programs. For example, a queue is a high level concept, which can be implemented by means of a variety of low level data structures, namely, singly linked list, doubly linked list, and even arrays. Therefore, a programmer can design a function by using enqueue and dequeue calls without any concern for their actual implementations. In other words, a programmer

can design a program by using abstract data types without waiting for their actual implementations. In the C and Fortran languages, there are no constructs that support the principle of information hiding. There was developed a tool to localize variables and functions and support information hiding as follows.

* Localization of variables: Organize global variables and functions which refer to those global variables into package-like groups. This is achieved by applying the concept of *closure* of functions to a set of global variables. This step leads to groups of functions and the global variables referred to by those functions.
* Localization of functions: Group locally called functions and the calling function in the same group.
* Information hiding and hierarchical structuring: Organize groups of functions into hierarchical package structures based on the visibility of functions within groups. Those functions and variables which are only externally referable and visible to other packages constitute the package specification, whereas the functions and variables in a package body are hidden from other packages and only visible to functions in the same package

The restructuring steps explained above offer a framework for translating a program in imperative languages, such as C and Fortran, into other languages that support modularity and hierarchical structure.

***C. System Sandwich Approach*** For badly structured programs that need to be retained for their output and which cannot be restructured with any hope, a sandwich approach can be applied, as illustrated in Figure 7.11.



The idea is to write a new front-end interface and a new back-end data base so that it is easy to interface with the program and the program’s outputs are recorded in a more structured way. The front-end and the back-end communicate for report generation purpose. The old system is used just for producing outputs.

***D. Clustering Approach*** Software modularization is an important design step in which a larger system is partitioned into smaller-sized cohesive chunks, called modules. During maintenance, a program can be remodularized in two broad ways as follows:

* System level remodularization: A program is remodularized at the system level by partitioning the program into smaller modules. This is a top-down approach to remodularize a program. The concept of system level remodularization has been illustrated in Figure 7.12.



* Entity level remodularization: At this level, a program is remodularized by grouping the entities to form larger modules. This is a bottom-up approach to remodularizing a program. The concept of entity level remodularization has been illustrated in Figure 7.13.



The concept of *clustering* plays a key role in modularization. Modularization is defined as the clustering of large amounts of entities in groups in such a way that the entities in one group are more closely related, based on some *similarity* metrics, than entities in different groups. Such groups are called clusters. Clusters are defined as continuous regions of space containing a relatively high density of points, separated from other such regions by regions containing a relatively low density of points. Such a broad definition can be easily applied to software systems, as illustrated in the VRML diagram of Figure 7.6. While applying the idea of clustering to a set of entities, two basic questions need to be answered.

* *Similarity metrics:* Clustering algorithms group similar entities together. Therefore, there is a need for a metric to capture the idea of similarity. For example, in a software system, entity *a* is more similar to *b* than to *c*. A similarity metric always yields a value between 0 and 1, where 1 means highly similar. A number of metrics to measure similarity are found in the literature: *distance measure*, *association coefficients*, *correlation coefficients*, and *probabilistic measures*. In the following, we explain the first two measures.
* Distance measure: The most common distance measures are the squared Euclidean distance and the Manhattan distance.
* Association coefficients: This measure is also known as the *simple matching coefficient*. The association coefficient for two entities *x* and *y* are expressed in terms of the number of features which are present for both the entities. Let *a* be the number of features present for both *x* and *y*, *b* be the number of features present for *x* but not for *y*, *c* be the number of features present for *y* but not *x*, and *d* be the features *not* present for both *x* and *y*. The *simple matching coefficient* is defined as *simple*(*x*, *y*) = (*a* + *d*)∕(*a* + *b* + *c* + *d*). The *Jaccard coefficient* is defined as *Jaccard*(*x*, *y*) = *a*∕(*a* + *b* + *c*).
* *Selection of clustering algorithms:* Clustering algorithms have been developed in diverse application areas, namely, image processing, pattern recognition, biology, software testing, information retrieval, graph theory, and information architecture. The techniques used in those algorithms can broadly be grouped into the following four categories.
* Graph theoretical algorithms: These algorithms work on graph representations of systems to be clustered. The nodes of those graphs represent entities, and edges represent relationships between nodes. The graph algorithms try to find subgraphs where each subgraph is a cluster. Many graph theoretical clustering algorithms use the concepts of minimum-spanning trees, graph reduction, aggregate node, and *k*-components.
* Construction algorithms: Clustering algorithms in this group assign the entities to clusters in one pass. The resulting clusters are either predetermined or identified by the algorithms. Many algorithms in this category use the following common techniques: *geographic technique* and *density search technique*. In the algorithms which are based on geographic techniques, entities are represented on a two-dimensional plane; the algorithm divides the plane into two halves; and the entities lying on the same side of the dividing line are said to belong to the same cluster. The algorithms based on density search work as follows: (i) find regions containing a relatively high density of entities; each of those regions is a member of the set of initial clusters; (ii) merge clusters to find lager clusters;members of clusters may be moved to neighboring clusters while merging them.
* Optimization algorithms: Optimization algorithms are also called improvement algorithms or iterative algorithms. The basic structure of those algorithms has been illustrated in the following:

1. Find an initial partition of k clusters.

2. REPEAT

Determine the seed point of each cluster.

Move each entity to the cluster with the seed point having the highest level of similarity with the entity.

 UNTIL no entities can be moved from one cluster to another.

The *centroid* of a cluster is taken as the cluster’s seed point. The centroid of a cluster is interpreted to be the “average” of the cluster. Selection of the initial set of *k* clusters can have an impact on the final clustering.

* Hierarchical algorithms: Hierarchical algorithms build a hierarchy of clustering. There are two broad kinds of hierarchical algorithms: *agglomerative algorithms* and *divisive algorithms*. The working of an agglomerative algorithm has been illustrated in Figure 7.13, whereas Figure 7.12 illustrates the working of a divisive algorithm. The clustering of a hierarchical algorithm can be visualized in a *dendrogram*. The dendrogram representation of the hierarchy in Figure 7.12 has been shown in Figure 7.14.



The general structure of an agglomerative algorithm is as follows:

1. IF there are N entities, begin with N clusters such that each cluster contains a unique entity. Compute the similarities between the clusters.

2. WHILE there is more than a cluster DO

Find the most similar pair of clusters and merge them into a single cluster.

Recompute the similarities between the clusters.

 END

Divisive clustering algorithms work in a top-down manner as follows. (i) in the beginning, all the *N* entities belong to one cluster; and (ii) in each step, a cluster is partitioned into two clusters. Finally, after *N* steps, there are *N* clusters with one entity in each cluster.

***E. Program Slicing Approach*** The concept of program slicing, including forward program slicing and backward program slicing, has been explained in Chapter 4 Reengineering. Informally, the set of statements that can affect the value of a variable at some point of interest in a program is called a backward program slice. Similarly, the set of statements that are likely to be affected by the value of a variable at some point of interest in a program is called a forward program slice. The key idea in program

slicing is to identify and extract a cohesive subset of statements from a program. Therefore, if a module supports multiple functionalities, a portion of the code can be extracted to form a new module. For example, let us consider a module *A* supporting functionalities *{f*1, *f*2, *f*3, *f*4*}*. If subsets *A*′ = *{f*1, *f*2*}* and *A*′′ = *{f*3, *f*4*}* are more cohesive, while the degree of cohesion between members of the two subsets is weak, then module *A* can be split up into two modules *A*′ and *A*′′ by applying the idea of program slicing. In addition, large functions can be decomposed into smaller functions by means of program slicing to restructure programs and reuse code segments. The idea of program slicing can be applied to object-oriented programs as well.