7

REFACTORING

He who rejects change is the architect of decay. The only human institution which rejects progress is the cemetery.

—Harold Wilson

7.1 GENERAL IDEA

Developers continuously modify, enhance, and adapt software to new requirements and execution environments. The software evolves with time, and, most likely, it deviates from its intended design. As the software further evolves and strays too far away from its original design, three important things happen to the software:

- *Decreased understandability:* It becomes increasingly difficult to understand the software and it becomes less maintainable.
- *Decreased reliability:* The reliability of the software decreases. As the software deviates from its original design and as documentations become out-of-date, faults are inadvertently introduced into the software during maintenance.
- *Increased maintenance cost:* The cost of maintaining the software rises in the absence of preventive measures.

Software Evolution and Maintenance: A Practitioner's Approach, First Edition.
Priyadarshi Tripathy and Kshirasagar Naik.
© 2015 John Wiley & Sons, Inc. Published 2015 by John Wiley & Sons, Inc.

The difficulty in understanding the software is due to the:

- increased complexity of the code during the maintenance phase;
- out-of-date documentation;
- code not conforming to standards.

Therefore, there is a need to decrease the complexity of software by improving its internal quality. The internal quality of software is improved by means of *restructuring* the software. If restructuring is performed on an object-oriented software, then it is called *refactoring* [1]. Software should be continually restructured during and between other maintenance activities so that programmers find it easier and easier to work with it. If programmers find it easy to work with the software, new functions and features can be easily added to the system, and bugs can be easily fixed.

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 is informally stated as the modifications of software in order to make it [2]:

- easier to understand;
- easier to change;
- easier to update its internal and external documentations; and
- less susceptible to faults when changes are made in the future.

The term *software* in *software restructuring* is used in a broad sense to refer to both code and documentations—both internal and external. A higher level goal of software restructuring is to increase the *software value* [2] as explained in the following:

- *External software value:* This is the value of the software as perceived by the customers. For example, a software with faults may fail to satisfy the business needs of the customers. Such software may be seen by the customers to have less value.
- *Internal software value:* This represents the cost saving due to three factors: (i) maintenance cost saving due to a good structure of the software; (ii) cost saving due to potential reuse of components of a software with good structure; and (iii) cost saving due to extended use of a software.

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.

Developers and managers should be aware of software restructuring for the following reasons [2]:

- *Better understanding:* By reorganizing software with easily traceable structures, it becomes easier for programmers to understand it. Ease of understanding leads to easier documentation, easier testing, potentially greater programmer productivity, and reduced dependence of the maintenance group on a small number of programmers who alone understand the code.
- *Keep pace with new structures:* As time passes, new generations of programmers are taught new software structures. Therefore, transforming existing software into new software with structures that are close to what is taught to new programmers is beneficial because it will enhance the understandability of the software. This in turn will reduce the time that maintenance programmers take to become familiar with the software.
- *Better reliability:* It is easier to locate and fix bugs in well-structured, well-understood software than in poorly understood software. While fixing bugs in a poorly understood software, one may introduce more bugs due to side effects of code modification.
- *Longer lifetime:* The lifetimes of software can be increased by making them maintainable by means of improving their structures.
- *Automated analysis:* Programs with good structures are more amenable to automatic analysis than unstructured programs. For example, a test generation tool is likely to be more successful in generating an effective set of tests for a program with a good structure than for a program with complex dependencies among modules.

Before we move on to the next section, we summarize the characteristics of restructuring and refactoring as follows:

- The objective of restructuring and refactoring is to improve the internal and external values of software.

- When a subject program is transformed into a new program, the original program's external behavior is preserved by the new program. In other words, the two programs are functionally identical from the viewpoint of users.
- Restructuring does not normally involve code transformation to implement new requirements. Rather, restructuring can be performed without adding new requirements to the existing system.
- When a subject program is transformed into a new program, the relative level of abstraction is preserved. For example, a program in C is transformed into another C program, rather than a program in an assembly language. However, the concept of program restructuring can be applied to transform legacy code into a more structured form or migrate it to a different programming language. That is, restructuring and refactoring can be used in reengineering a system.

7.2 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.

7.2.1 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 artifact 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 [3].
- *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 superclasses.
- *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, statechart 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 [4]. To refactor a software architecture, Philipps and Rumpe [5] have proposed an approach where refactoring rules are based on the graphical representation of a system architecture.

7.2.2 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

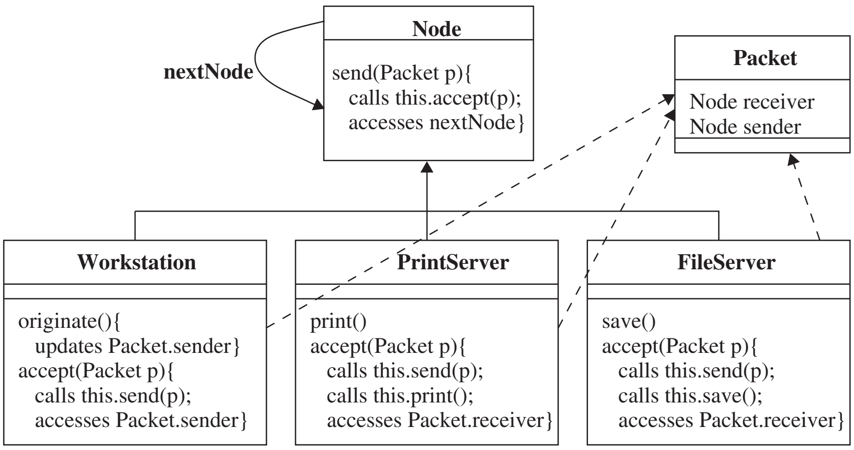


FIGURE 7.1 Class diagram of a local area network (LAN) simulator. From Reference 6. © 2007 Springer

the following we give some examples of refactorings in the context of a class diagram shown in Figure 7.1 [6]. 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 *print* to *process* in class *FileServer*. Perform this refactoring together with *R1*. In *R1* and *R2*, the new names of *print* 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 *R1* must occur before *R8*.
- *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.

7.2.3 Ensure that Refactoring Preserves the Behavior of the Software

Ideally, the behavior of a program after refactoring should be the same as the behavior before refactoring. In the original definition of behavior preservation proposed by Opdyke [1], program behavior 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 [7].

For programs written in languages with a simple and formally defined semantics, such as *Prolog*, one can prove for some kinds of refactorings that program semantics are preserved. On the other hand, for Java and C++ programs, much restrictions have to be put on language constructs and refactorings to show that program behavior is preserved.

7.2.4 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.

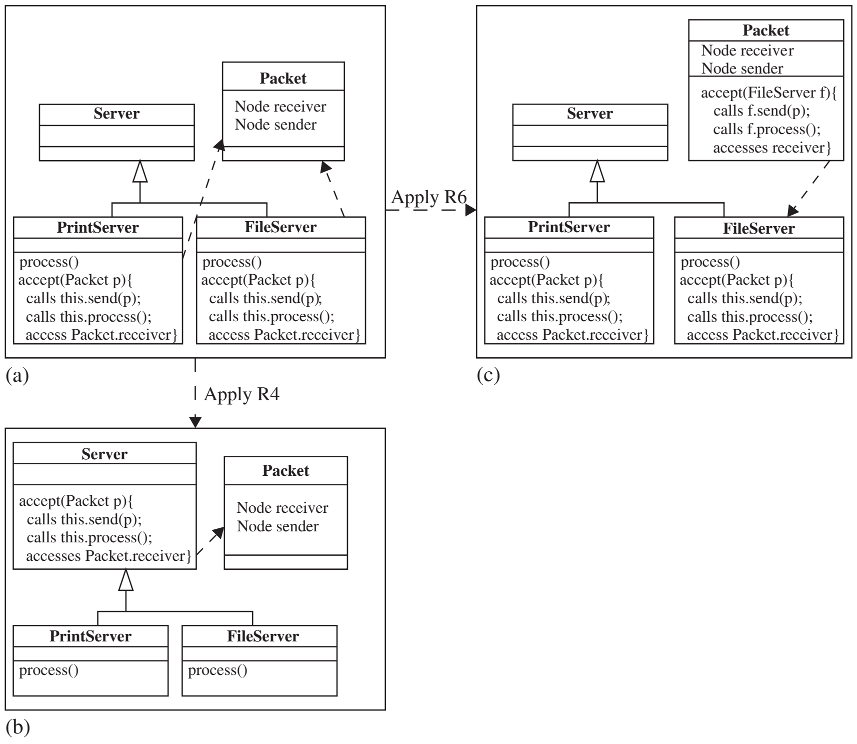


FIGURE 7.2 Applications of two refactorings. From Reference 6. © 2007 Springer

7.2.5 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. For example, some refactorings eliminate code duplications, some raise reusability, some improve performance, and some improve maintainability. 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. Kataoka et al. [8] have further investigated the idea of the coupling metrics by introducing three finer levels of coupling: *return value* coupling, *parameter* coupling, and *shared variable* coupling. Those couplings are explained as follows. If method *A* uses a return value from method *B*, there exists a return value coupling between the two methods. If method *B* receives *n* parameters from method *A*, there exists an *n* parameter coupling between the two methods. Similarly, if method *A* uses *n* class variables in common with method *B*, there exists *n* shared variable coupling between the two methods. The three kinds of coupling metrics can be combined to produce a single composite coupling metrics by taking their weighted sum.

Rather than evaluate the impacts *after* applying refactorings, one can select refactoring steps such that the program after refactoring possesses better quality attributes. Tahvildari and Kontogiannis [9] have used the concept of *soft-goal graph*—introduced in detail in the book by Chung, Nixon, Yu and Mylopoulos [10]—to help guide the application of refactorings. Intuitively, a soft-goal graph for a quality attribute, for example, maintainability, is a hierarchical graph rooted at the desired change in the attribute, for example, high maintainability. The internal nodes represent successive refinements of the attribute and are basically the soft goals. Finally, the leaf nodes represent refactoring transformations which fulfill or contribute positively/negatively to soft goals which appear above them in the hierarchy. A partial example of a soft-goal graph with one leaf node, namely, `Move`, has been illustrated in Figure 7.3. The dotted lines between the leaf node `Move` and three soft goals—`High Modularity`, `High Module Reuse`, and `Low Control Flow Coupling` imply that the `Move` transformation impacts those three soft goals.

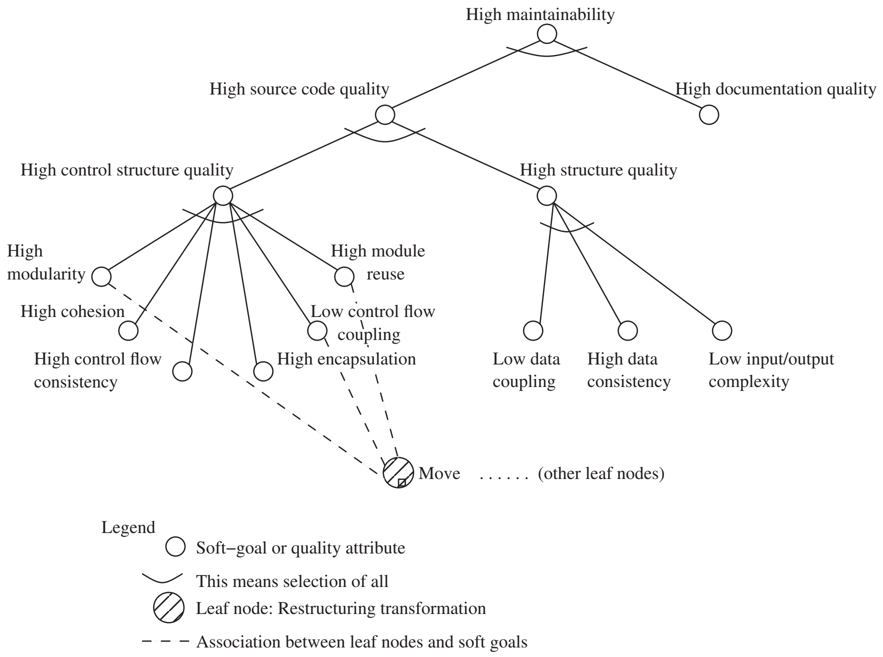

High maintainability
High source code quality
High documentation quality
High control structure quality
High structure quality
High modularity
High module reuse
High cohesion
Low control flow coupling
High control flow consistency
High encapsulation
Low data coupling
High data consistency
Low input/output complexity
Move
...... (other leaf nodes)
Legend
Soft–goal or quality attribute
This means selection of all
Leaf node: Restructuring transformation
Association between leaf nodes and soft goals

FIGURE 7.3 An example of a soft-goal graph for maintainability, with one leaf node. From Reference 11. © 2002 IEEE

7.2.6 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.

7.3 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.

7.3.1 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 *postconditions*. 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 postcondition 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 postconditions have been suggested by researchers to address the problem of behavior preservation. In the context of object-oriented database schemas, which have much similarities with UML class diagrams, Banerjee et al. [12] have shown that behavior preserving transformations can be applied to database schemas by using invariants. An example of invariant in the schema context 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, postconditions, and invariants.

7.3.2 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).

An example of a program graph has been shown in Figure 7.4. In a program graph, method bodies are represented as simplified trees with a root at a B-node, connected with E-nodes, where an E-nodes represent assignments, accessed variables, parameters, and function calls. A B-node with two E-nodes has been shown within the dotted box in Figure 7.4. The body of the originate method is contained in the Node superclass.

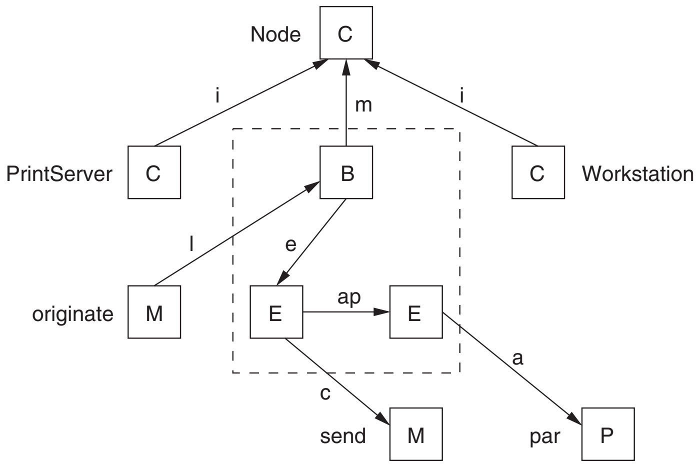


FIGURE 7.4 An example of a program graph. From Reference 13. © 2006 Elsevier

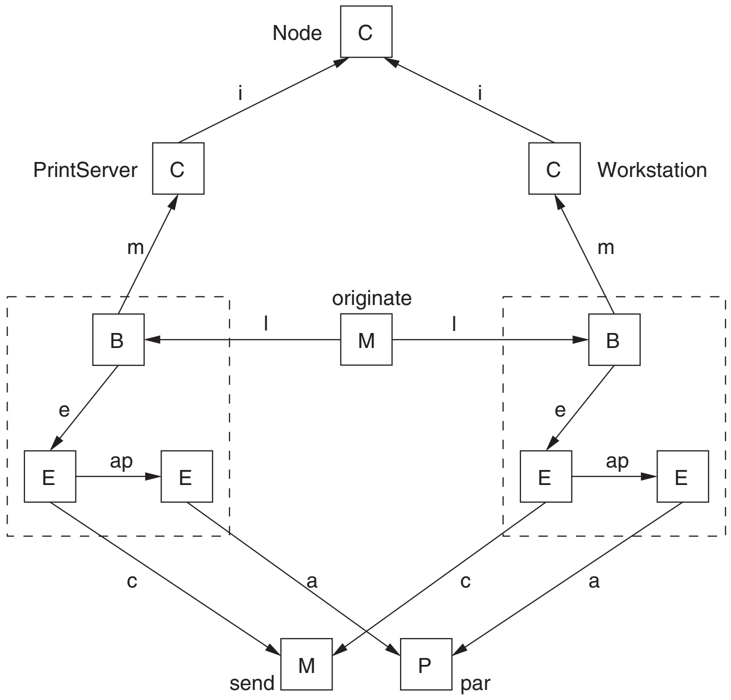


FIGURE 7.5 Program graph obtained after applying *push-down method* refactoring to the program graph of Figure 7.4. From Reference 13. © 2006 Elsevier

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`. The push-down method refactoring has been is applied to the method `originate` in the graph of Figure 7.4 to obtain a new graph shown in Figure 7.5.

7.3.3 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. [14] 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 $dist_B(x, y)$ is computed as follows:

$$dist_B(x, y) = 1 - \frac{|p(x) \cap p(y)|}{|p(x) \cup p(y)|} \tag{7.1}$$

where $p(x) = \{p_i \in B |\ x \text{ possesses property } p_i\}$.

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

$$B_f = \{f \cup \text{all methods directly used by} f \cup \text{all attributes used by } f\}. \tag{7.2}$$

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

$$B_g = \{g \cup \text{all methods using } g\}. \tag{7.3}$$

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 $B = B_{f_1} \cup B_{f_2}$. Similarly, if we are interested in the distance between a method f and an attribute g, then we have $B = B_f \cup B_g$.

Now, given a class C, let $M = \{m_1, \ldots, m_k\}$ be the set of methods and $A = \{a_1, \ldots, m_n\}$ 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 $M \cup A$, 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 [14]. 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.

Coupling Metrics The idea of coupling presented here was introduced by Kataoka, Imai, Andou, and Fukaya [15]. First, three basic kinds of couplings are explained:

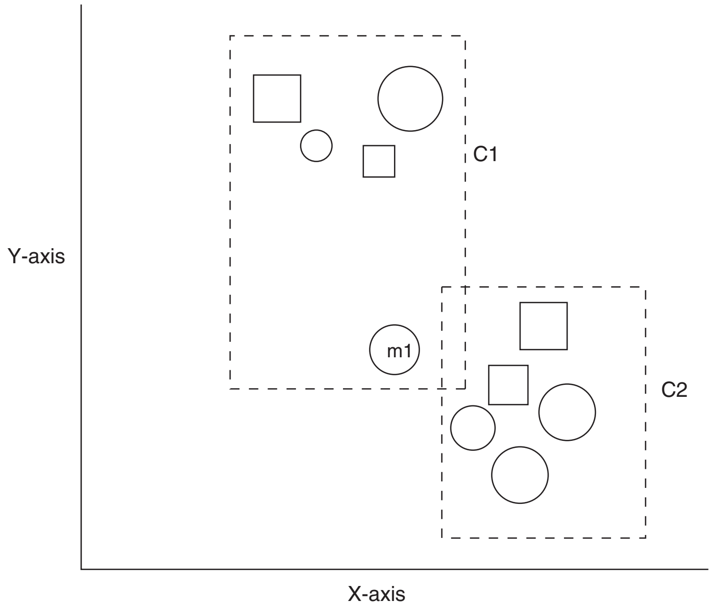


FIGURE 7.6 An example of a VRML diagram of two classes C1 and C2. Circles denote methods and squares denote attributes

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 $C_{rv}(A)$, with all the methods it calls and all the methods it is called from is computed as follows.

$$C_{rv}(A) = \sum_{m_\rho \in \rho(A)} K_{rv}(m_\rho) + \sum_{m_\sigma \in \sigma(A)} K_{rv}(m_\sigma), \tag{7.4}$$

where,

$\rho(A)$ is the set of methods which provide return values to A.

$\sigma(A)$ is the set of methods that use the return values provided by A.

$K_{rv}(m)$ is the return value coupling inter-class coefficient. $K_{rv}(m) = 1$ if both methods m and A belong to the same class. On the other hand, if m and A are in different classes, then $K_{rv}(m) = \kappa_{rv}$, where $\kappa_{rv} > 1$.

The idea behind $K_{rv}(m)$ 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 $C_{pp}(A)$, with all the methods it calls and all the methods it is called from is computed as follows.

$$C_{pp}(A) = \sum_{m_\zeta \in \zeta(A)} K_{pp}(m_\zeta) p_{m_\zeta} + \sum_{m_\xi \in \xi(A)} K_{pp}(m_\xi) p_{m_\xi}, \tag{7.5}$$

where,

$\zeta(A)$ is the set of methods that call A.

$\xi(A)$ is the set of methods that are called by A.

$K_{pp}(m)$ is the parameter coupling inter-class coefficient. $K_{pp}(m) = 1$ if both methods m and A are in the same class. On the other hand, if m and A are in different classes, then $K_{pp}(m) = \kappa_{pp}$, where $\kappa_{pp} > 1$.

p_{m_ζ} is the number of parameters received by method m_ζ when m_ζ is called by another method.

p_{m_ξ} is the number of parameters passed by method m_ξ when m_ξ calls another method.

The idea behind $K_{pp}(m)$ 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 $C_{sv}(A)$, with all the methods in the same class is computed as follows.

$$C_{sv}(A) = \sum_{m_\chi \in \chi(A)} K_{sv}(m_\chi) v_{m_\chi}, \tag{7.6}$$

where,

$\chi(A)$ is the set of methods that use class or instance variables in common with A.

v_m is the number of class or instance variables that are used both in m and A.

$K_{sv}(m)$ is the shared variable inter-class coefficient. $K_{sv}(m) = 1$ if both methods m and A are in the same class. On the other hand, if m and A are in different classes, then $K_{sv}(m) = \kappa_{sv}$, where $\kappa_{sv} > 1$.

The motivation for introducing $K_{sv}(m)$ is similar to the motivation for introducing $K_{rv}(m)$.

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 [15]. The concept of *weighted sum* is a simple strategy to combine the three coupling metrics. Three weighting factors, namely, W_{rv}, W_{pp}, and W_{sv} are introduced to represent the relative importance of $C_{rv}(A)$, $C_{pp}(A)$, and $C_{sv}(A)$, respectively, such that the following constraints, denoted by Eqs. 7.7 and 7.8, hold:

$$W_{rv} + W_{pp} + W_{sv} = 1. \tag{7.7}$$

$$0 < W_{rv} \leq W_{pp} \leq W_{sv}. \tag{7.8}$$

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

$$C_T(A) = W_{rv}C_{rv}(A) + W_{pp}C_{pp}(A) + W_{sv}C_{sv}(A). \tag{7.9}$$

In their experiments, Kataoka et al. [15] have used the following values of the different parameters and weighting factors:

$$\kappa_{rv} = 1.5, \kappa_{pp} = 2.0, \kappa_{sv} = 3.0$$
$$W_{rv} = 0.2, W_{pp} = 0.2, W_{sv} = 0.6.$$

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

7.4 MORE EXAMPLES OF REFACTORINGS

In Section 7.2.2, we explained some examples of refactoring by applying them to class diagrams. In this section, we intuitively explain additional examples of refactorings, without code or class diagrams.

- *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`. The refactored code segment has been shown in the following:

```
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.

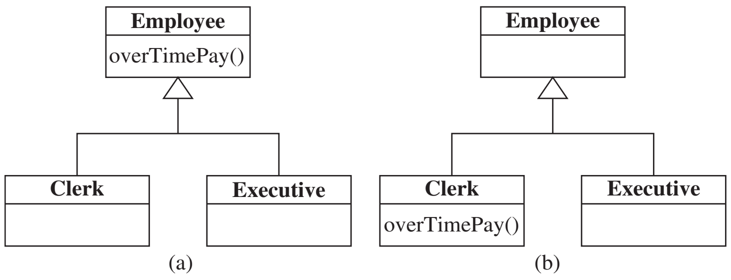


FIGURE 7.7 Illustration of the push-down method refactoring: (a) the class diagram before refactoring; (b) the class diagram after refactoring

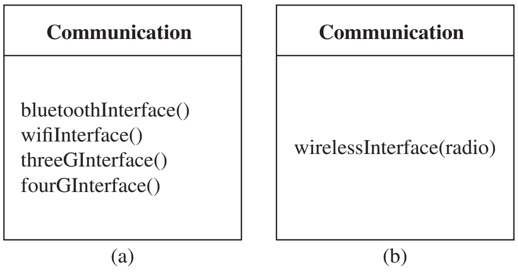


FIGURE 7.8 An example of parameterizing a method. There are four methods in (a), whereas there is one method in (b) with one parameter

- *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 invoked with different values of `radio` so that the `wirelessInterface` method can in turn invoke different radio interfaces.

7.5 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.

7.5.1 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, level, function, and module, play key roles in understanding code.

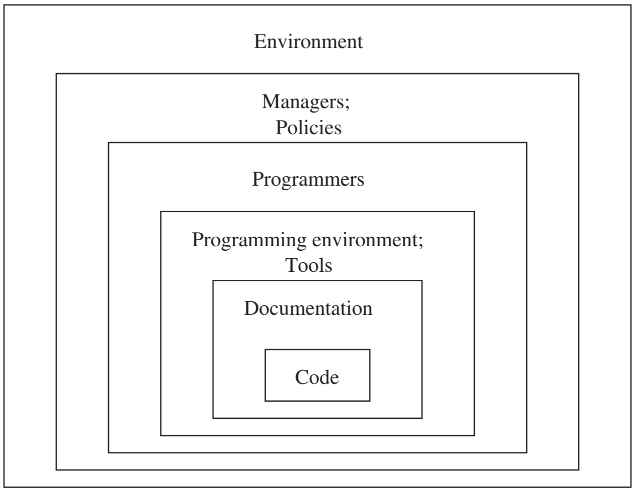


FIGURE 7.9 Factors which can influence software structure. From Reference 2. © 1989 IEEE

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.

7.5.2 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.

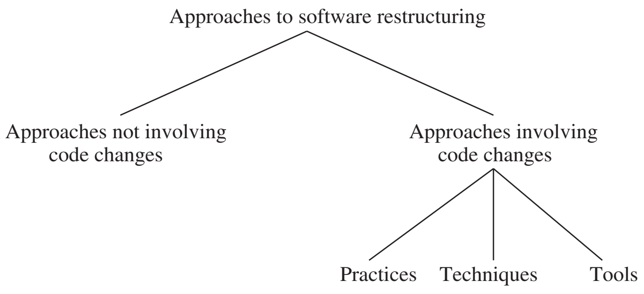

Approaches to software restructuring
Approaches not involving code changes
Approaches involving code changes
Practices
Techniques
Tools

FIGURE 7.10 Broad classification of approaches to software structuring

- *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 preprocessors; (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*.

7.5.3 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. Ashcroft and Manna [16] have 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.

Baker [17] has given an algorithm to transform a flowgraph 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. Chu and Patel [18] 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.

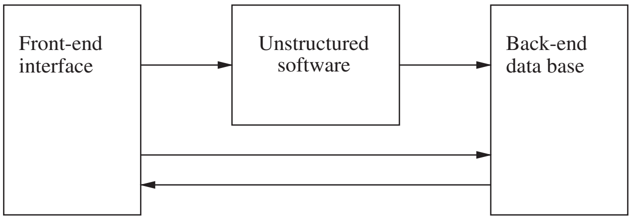


FIGURE 7.11 System sandwich approach to software restructuring. The arrows represent the flow of data and/or commands

- 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.

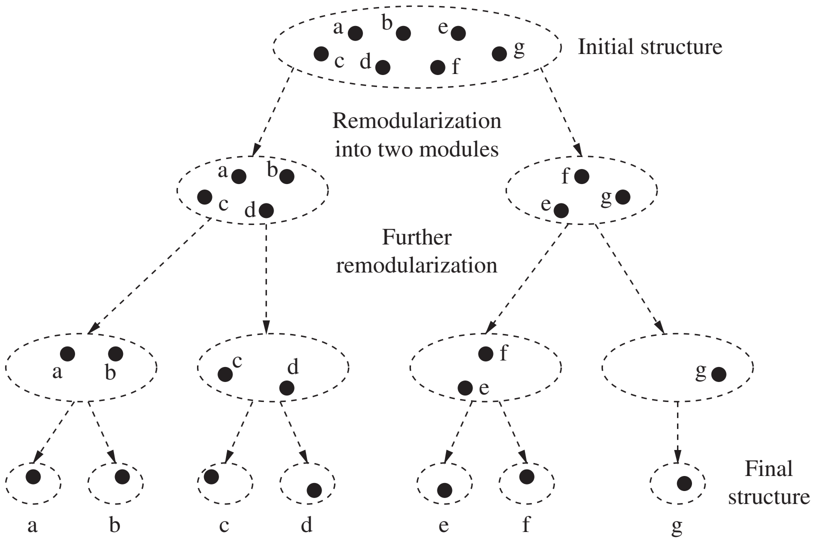
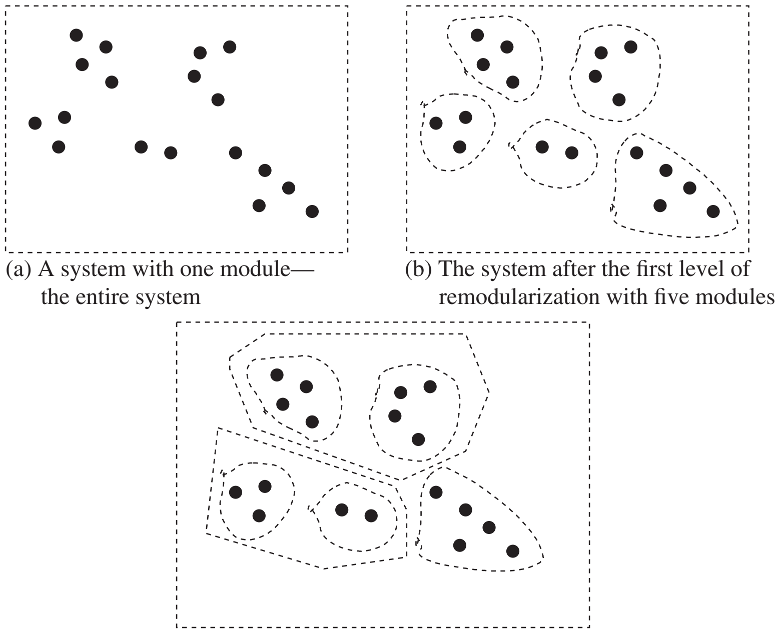


FIGURE 7.12 Illustration of system level remodularization. Bullets represent low level entities. Dotted shapes represent modules. Arrows represent progression from one level to the next


(a) A system with one module— the entire system
(b) The system after the first level of remodularization with five modules

(c) The system after the second level of remodularization with three upper level modules

FIGURE 7.13 Illustration of entity level remodularization. Bullets represent low level entities. Dotted shapes represent modules

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. A broad definition of clusters can be found in [19]: 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* [20]. 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 *dendogram*. The dendogram 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.

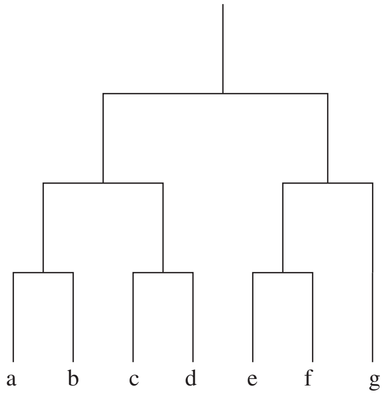


FIGURE 7.14 Dendogram representation of Figure 7.12

E. Program Slicing Approach The concept of program slicing, including forward program slicing and backward program slicing, has been explained in Chapter 4. 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 [21].

In addition, large functions can be decomposed into smaller functions by means of program slicing to restructure programs [22] and reuse code segments [23]. The idea of program slicing can be applied to object-oriented programs as well [24].

7.6 SUMMARY

Software refactoring and restructuring are topics of continued research for more than three decades. Several books have been written on the topic of software refactoring. Therefore, this chapter opened with an introduction to the idea of software refactoring and restructuring. We explained the negative impacts of software straying away from its original design: decreased understandability, decreased reliability, and increased maintenance cost. Therefore, a higher level goal of software refactoring is to increase the external software value and internal software value. In the second section, we explained the activities in a refactoring process: identifying what to refactor, determine which refactorings to apply, ensure that refactorings preserve the behavior of the software, apply the refactorings to the chosen entities, evaluate the impacts of the refactorings, and maintain consistency. We explained the concept of code smell by means of many examples, and showed what many refactorings look like. The ideas of critical pair analysis and sequential dependency analysis were explained toward finding a feasible subset of refactorings from a given set. In order to ensure that refactoring preserves the program's behavior, we explained two pragmatic ways: testing and verification of preservation of call sequence. We argued that preservation of input–output behavior of programs is not enough, and identified three other important behaviors that must be preserved: temporal constraints, resource constraints, and safety constraints. Applications of a few common refactorings were explained with class diagrams.

In the third section, we explained formalisms and techniques for refactoring: assertions, graph transformation, and software metrics. We explained two kinds of software metrics: cohesion and three kinds of coupling. The three kinds of coupling are: return value coupling, parameter coupling, and shared variable coupling. In the fourth section, we gave more examples of refactorings.

In the fifth section, we discussed the initial work on software restructuring: factors influencing software structure, classification of restructuring approaches, and restructuring techniques. The factors affecting software structures are code, documentation, tools, programmers, managers and policies, and environment. The two broad categories of approaches to restructuring are approaches not involving code changes and approaches involving code changes. The restructuring techniques that we explained are elimination-of-goto, system sandwich, localization and information hiding, and clustering approaches.

LITERATURE REVIEW

Many researchers have contributed to various facets of software restructuring and software refactoring for more than three decades, with hundreds of research publications and numerous tools. In addition, excellent books have been written and edited on the topic.

The main research results in software restructuring until the mid 1980s have been succinctly compiled in the form of an edited book by Arnold [25]: *Tutorial on Software Restructuring*. The book has been organized into eight parts. After a very good introduction in Part I, perspectives on software structure are provided in five research articles in Part II. In Part III, origins and effects of poor software structure are discussed in five articles. Strategies and tools for recognizing software structure are discussed in Part IV. Code level approaches and system level approaches to infusing software with structure are discussed in Part V and Part VI, respectively. Finally, restructuring criteria and rule-based restructuring are presented in Parts VII and VIII, respectively.

Seventy-two refactorings have been explained with class diagrams and code by Fowler in his book *Refactoring: Improving the Design of Existing Code* [26]. In the chapter *Refactoring Tools* by Roberts and Brant, they explain the technical criteria and the practical criteria for a refactoring tool. The technical criteria are as follows.

- Program database: For a tool to be useful, it is important that search and meaningful operations (e.g., substitution) on program entities can be performed, without the need for program testing.
- Parse trees: Parse trees are an important representation of programs, because a large number of refactorings operate at the method level and below.
- Accuracy: Accuracy means to what extent the program behavior has been preserved by the refactorings.

On the other hand, the practical criteria are as follows.

- Speed: It may be noted that tools in software development are created to support programmers. If tools do not fit the way programmers carry out their tasks, then tools will not be adopted. Therefore, speed of refactoring is important to programmers. If some refactorings take too long to achieve accuracy, then tool designers may not support those refactorings.
- Undo: An undo operation comes very handy in exploring the consequences of refactorings. If a programmer does not like the results of refactorings, an undo operation allows him to revert to the original program, without having to explicitly keep a copy of the program.
- Integration with tools: For refactoring tools to be widely used, it is important that such tools are made a part of integrated development environments (IDE).

In the book entitled *Refactoring, Reuse, and Reality* [26], the author has identified some reasons why programmers did not refactor their code.

- Since the benefits of refactorings are long term, a programmer may not be motivated to do it now.
- Since refactoring is an overhead, it does not contribute to day-to-day productivity.
- There is no guarantee that implementations of some refactorings do not introduce faults.

The chapter also discussed how to reduce the overhead of refactoring and the idea of refactoring safely. Intuitively, safe refactoring means refactoring a program without introducing faults into the program.

Ping and Kontogiannis [27] have proposed an approach to refactor web sites to the controller-centric architecture. Their approach is twofold: (i) a domain model is defined to represent dependencies among web pages in order to understand the structure of the current web site; and (ii) a system architecture is designed as a reference model for restructuring the existing web site into a controller-centric architecture based on the well-known concept of odel view controller (MVC). A link between two web pages is said to exist if one page contains a link to the other. For dependency analysis, links are grouped into two broad categories: reference link and conditional link. They consider four kinds of reference links as follows.

- Inner link: This means a web page has a link to itself.
- Handover link: This means that a link leads to a different page.
- Include link: This means that the target page is included in the source page when the link is activated.
- Invocation link: This represents a communication link between a source page and a target page.

The book *Refactoring HTML: Improving the Design of Existing Web Applications* by Harold [28] is solely about refactorings of HTML code. The book has a full chapter about smells of bad HTML code. Some example smells are:

- Illegible code: This code is difficult to understand.
- Slow page rendering: Though communication networks, databases, and servers can contribute to delay in page display, there is said to be a problem if page rendering from a locally saved file takes a long time.
- Pages look significantly different in different browsers: A problem is said to be present if the pages are illegible when browsed with common browsers, namely, Internet Explorer and Firefox.

Next, the book explains six broad categories of refactorings for HTML code as follows.

- Well-Formedness: Add End-Tag is an example of this category of refactorings.
- Validity: Remove All Non-Existent Tags is an example of the validity category.
- Layout: Add an ID Attribute is an example of layout refactorings.
- Accessibility: Introduce Skip Navigation is an example of accessibility.
- Web Applications: Prevent Caching is an example of web application refactorings.
- Content: Correct Spelling is an example of content refactorings.

In their book *Refactoring in Large Software Projects* [29], Lippert and Roock explain architecture smells, best practices for large refactorings, refactoring relational databases, refactoring application program interfaces (API), and tool-based detection and avoidance of architecture smells. Intuitively, large refactorings possess the following characteristics.

- Long time: Here, days and weeks mean long time.
- Significant change: This means changes to the more significant concepts, namely, software architecture, databases, and APIs.
- Large team: Almost all the people currently involved with the development and/or maintenance of the software are involved.

An excellent survey of software refactoring can be found in the article by Mens and Tourwe [30]. By means of a running example, they explain all the refactoring activities found in Section 7.2. Additional discussions of refactoring techniques and formalisms and types of software artifacts can be found in the article. Finally, the article presents insightful discussions of tool support for automation of refactoring activities. The desirable features of a tool for refactoring are reliability, configurability, coverage, scalability, and language independence.

REFERENCES

[1] W. F. Opdyke. 1992. Refactoring: A program restructuring aid in designing object-oriented application framework. PhD thesis, University of Illinois at Urbana-Champaign.

[2] R. S. Arnold. 1989. Software restructuring. *Proceedings of the IEEE*, 77(4), 607–616.

[3] S. Ducasse, M. Rieger, and S. Demeyer. 1999. *A Language Independent Approach for Detecting Duplicated Code*. Proceedings of the International Conference on Software Maintenance, 1999, IEEE Computer Society Press, Los Alamitos, CA, pp. 109–118.

[4] J. H. Jahnke and A. Zundorf. 1997. *Rewriting Poor Design Patterns by Good Design Patterns*. Proceedings of the ESE/FSE Workshop on Object-Oriented Reengineering, 1997, ACM, New York.

[5] J. Philipps and B. Rumpe. *Refinement of Information Flow Architecture*. Proceedings of the International Conference on Formal Engineering Methods, 1997, IEEE Computer Society Press, Los Alamitos, CA.

[6] T. Mens, G. Taentzer, and O. Runge, 2007. Analyzing refactoring dependencies using graph transformation. *Software and Systems Modeling*, 63, 269–285.

[7] F. Tip, A. Kiezun, and D. Baumer. 2003. *Refactoring for Generalization Using Type Constraints*. Proceedings of SIGPLAN Conference on Object-Oriented Programming, 2003, Systems, Languages, and Applications, ACM, New York, pp. 13–26.

[8] Y. Kataoka, M. D. Ernst, W. G. Griswold, and D. Notkin. 2001. *Automated Support for Program Refactoring using Invariants*. Proceedings of the International Conference on Software Maintenance, 2001, IEEE Computer Society Press, Los Alamitos, CA, pp. 736–743.

[9] L. Tahvildari and K. Kontogiannis. 2002. *A methodology for Developing Transformations Using the Maintainability Soft-goal Graph*. Proceedings of Working Conference on Reverse Engineering, 2002, IEEE Computer Society Press, Los Alamitos, CA, pp. 77–86.

[10] L. K. Chung, B. A. Nixon, E. Yu, and J. Mylopoulos. 2000. *Non-Functional Requirements in Software Engineering*. Kluwer Publishing.

[11] L. Tahvildari and K. Kontogiannis. 2002. *A Software Transformation Framework for Quality-driven Object-oriented Re-engineering*. Proceedings of the International Conference on Software Maintenance, 2002, IEEE Computer Society Press, Los Alamitos, CA, pp. 596–605.

[12] J. Banerjee, H. Kim W. Kim, and H. F. Korth. 1987. *Semantics and Implementation of Schema Evolution in Object-oriented Databases*. Proceedings of the ACM SIGMOD Conference, 1987, ACM, New York, pp. 311–322.

[13] B. Hoffmann, D. Janssens, and N. V. Eetvelde. 2006. Cloning and expanding graph transformation rules for refactoring. *Electronic Notes in Theoretical Computer Science*, 152, 53–67.

[14] F. Simon, F. Steinbruckner, and C. Lewerentz. 2001. *Metrics Based Refactoring*. Proceedings of the European Conference on Software Maintenance and Reengineering, 2001, IEEE Computer Society Press, Los Alamitos, CA, pp. 30–38.

[15] Y. Kataoka, T. Imai, H. Andou, and T. Fukaya. 2002. *A Quantitative Evaluation of Maintainability Enhancement by Refactoring*. Proceedings of the International Conference on Software Maintenance, 2002, IEEE Computer Society Press, Los Alamitos, CA, pp. 576–585.

[16] E. Ashcroft and Z. Manna. 1971. *The Translation of 'goto' Programs to 'while' Programs*. Proceedings of the 1971 IFIP Congress, 1971, pp. 250–260.

[17] B. S. Baker. 1977. An algorithm for structuring flowgraphs. *Journal of the ACM*, 24(1), 98–120.

[18] W. C. Chu and S. Patel. 1992. *Software Restructuring by Enforcing Localization and Information Hiding*. Proceedings of the Conference on Software Maintenance, 1992, IEEE Computer Society Press, Los Alamitos, CA, pp. 165–172.

[19] B. Everitt. 1974. *Cluster Analysis*. Heineman Educational Books, London.

[20] T. A. Wiggerts. 1997. *Using Clustering Algorithms in Legacy Systems Remodularization*. Proceedings of the Working Conference on Reverse Engineering, 1997, IEEE Computer Society Press, Los Alamitos, CA, pp. 33–43.

[21] H. S. Kim, Y. R. Kwon, and I. S. Chung. 1994. Restructuring programs through program slicing. *International Journal of Software Engineering and Knowledge Engineering*, 4(3), 349–368.

[22] A. Lakhotia and J.-C. Deprez. 1998. Restructuring programs by tucking statements into functions. *Information and Software Technology*, 40, 677–689.

[23] F. Lanubile and G. Ducasse. 1997. Extracting reusable functions by flow graph-based program slicing. *IEEE Transactions on Software Engineering*, 23(4), 246–258.

[24] L. Larsen and M. J. Harrold. 1996. *Slicing Object-oriented Software*. Proceedings of the International Conference on Software Engineering, 1996, IEEE Computer Society Press, Los Alamitos, CA, pp. 495–505.

[25] R. S. Arnold. 1986. *Tutorial on Software Restructuring*. IEEE Computer Society Press, Los Alamitos, CA.

[26] M. Fowler. 1999. *Refactoring: Improving the Design of Existing Programs*. Addison-Wesley.

[27] Y. Ping and K. Kontogiannis. 2004. *Refactoring Web Sites to the Controller-centric Architecture*. Proceedings of the 8th European Conference on Software Maintenance and Reengineering, 2004, IEEE Computer Society Press, Los Alamitos, CA, pp. 204–213.

[28] E. R. Harold. 2008. *Refactoring HTML: Improving the Design of Existing Web Applications*. Addison Wesley.

[29] M. Lippert and S. Roock. 2006. *Refactoring in Large Software Projects: Performing Complex Restructurings Successfully*. John Wiley.

[30] T. Mens and T. Tourwe. 2004. A survey of software refactoring. *IEEE Transactions on Software Engineering*, 30(2), 126–139.

EXERCISES

1. Briefly explain the need for restructuring software.

2. List the key activities in a software refactoring process.

3. How do programmers identify what to refactor?

4. How do you identify what refactorings to apply?

5. How do you select a feasible subset from a given set of refactorings?

6. Briefly explain the concept of preserving the software's behavior while refactoring.

7. Identify four key formalisms and techniques for refactoring.

8. Briefly explain the concept of assertions by means of examples.