SOFTWARE EVOLUTION AND MAINTENANCE

CHAPTER 9
FORWARD ENGINEERING

- Overview of Forward Engineering
- Refactoring
- Code Transformation
- Web-enabling
- Software Reengineering Strategies & Management

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

OVERVIEW OF FORWARD ENGINEERING

- **Forward engineering** is the traditional process of moving from high level abstractions and logical, implementation-independent designs to the physical implementation of a system.

Forward engineering VS Reverse Engineering

S.NO	Forward Engineering	Reverse Engineering
1.	In forward engineering, the application are developed with the given requirements.	In reverse engineering or backward engineering, the information are collected from the given application.
2.	Forward Engineering is a high proficiency skill.	Reverse Engineering or backward engineering is a low proficiency skill.
3.	Forward Engineering takes more time to develop an application.	While Reverse Engineering or backward engineering takes less time to develop an application.
4.	The nature of forward engineering is Prescriptive.	The nature of reverse engineering or backward engineering is Adaptive.
5.	In forward engineering, production is started with given requirements.	In reverse engineering, production is started by taking the products existing products.
6.	The example of forward engineering is the construction of electronic kit, construction of DC MOTOR , etc.	An example of backward engineering is research on Instruments etc.

REFACTORING

- Developers continuously modify, enhance and adapt software.
- As software evolves and strays away from its original design, three things happen.
  - Decreased understandability
  - Decreased reliability
  - Increased maintenance cost
- Decreased understandability is due to
  - Increased complexity of code
  - Out-of-date documentation
  - Code not conforming to standards
- Decrease the complexity of software by improving its internal quality by restructuring the software.
- Restructuring applied on object-oriented software is called refactoring.

REFACTORING

- A higher level goal of restructuring is to increase the software value
  - external software value: fewer faults in software is seen to be better by customers
  - internal software value: a well-structured system is less expensive to maintain
- Simple examples of restructuring
  - Pretty printing
  - Meaningful names for variables
  - One statement per line of source code

Activities in a Refactoring Process

- To restructure a software system, one follows a process with well defined activities.
  - Identify what to refactor.
  - Determine which refactorings to apply.
  - Ensure that refactoring preserves the software's behavior.
  - Apply the refactorings to the chosen entities.
  - Evaluate the impacts of the refactorings.
  - Maintain consistency.

Identify what to refactor

- The programmer identifies what to refactor from a set of high-level software artifacts.
  - source code;
  - design documents; and
  - requirements documents.
- Next, focus on specific portions of the chosen artifact for refactoring.
  - Specific modules, functions, classes, methods, and data can be identified for refactoring.

Identify what to refactor

- The concept of code smell is applied to source code to detect what should be refactored.
- A code smell is any symptom in source code that possibly indicates a deeper problem.
- Examples of code smell are:
  - duplicate code
  - long parameter list
  - long methods
  - large classes
  - message chain.

Identify what to refactor

- Entities to be refactored at the design level
  - software architecture
    - class diagram;
    - statechart diagram; and
    - activity diagrams;
  - global control flow; and
  - database schemas.

Determine which refactorings to apply

- Referring to Figure 7.1, some refactorings are
  - R1: Rename method *print* to *process* in class *PrintServer.*
  - R2: Rename method *print* to *process* in class *FileServer*. (R1 and R2 are to be done together.)
  - R3: Create a superclass *Server* from *PrintServer* and *FileServer*.
  - R4: Pull up method *accept* from *PrintServer* and *FileServer* to the superclass *Server*.
  - R5: Move method *accept* from *PrintServer* to class *Packet,* so that data packets themselves will decide what actions to take.
  - R6: Move method *accept* from *FileServer* to *Packet*.
  - R7: Encapsulate field receiver in *Packet* so that another class cannot directly access this field.
  - R8: Add parameter p of type *Packet* to method *print* in PrintServer to print the contents of a packet.
  - R9: Add parameter p of type *Packet* to method *save* in class *FileServer* so that the contents of a packet can be printed.

Determine which refactorings to apply

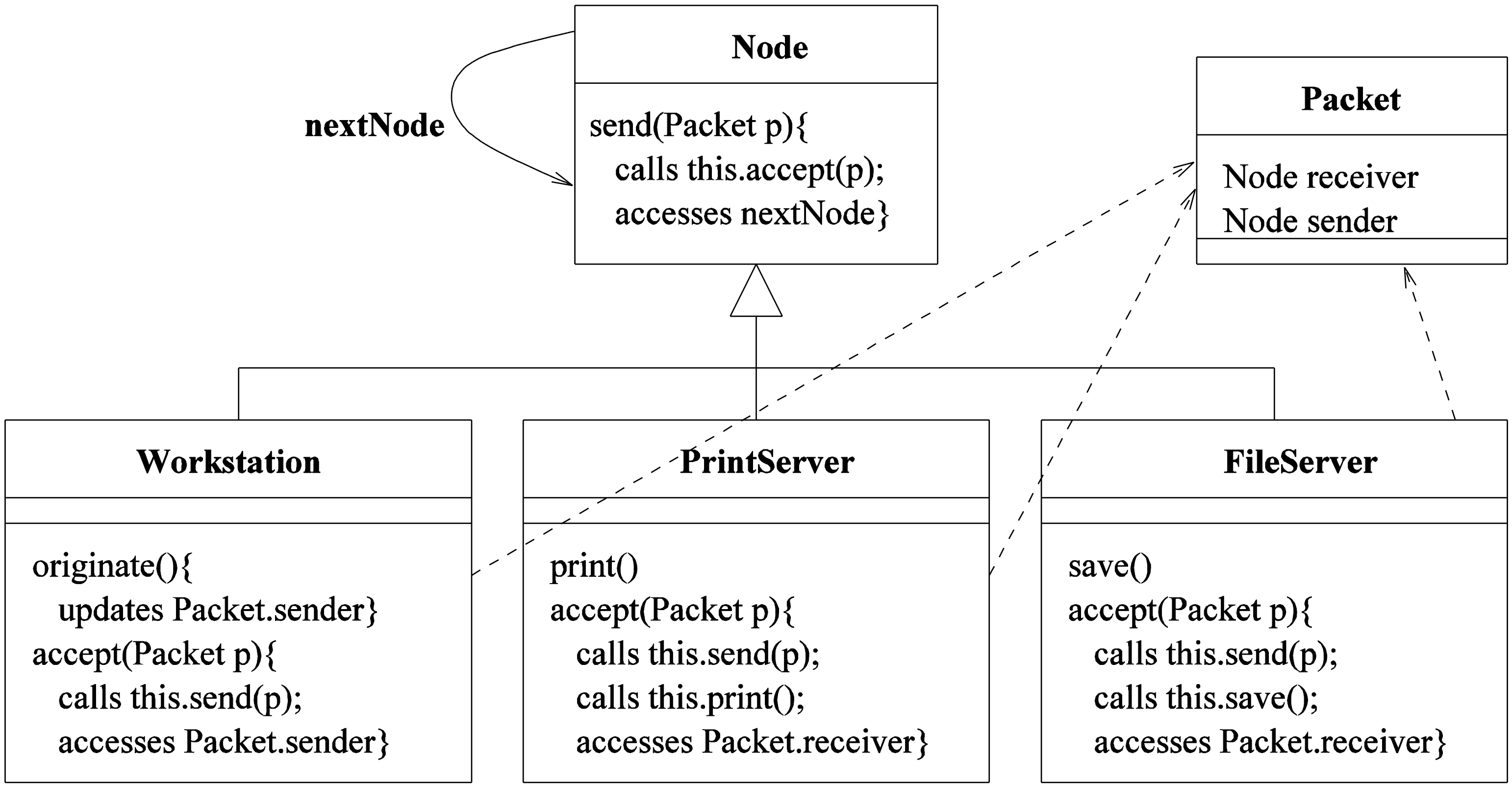

Figure 7.1: Class diagram of a Local Area Network (LAN) simulator [6] (©[2007] Springer).

Determine which refactorings to apply

- R1—R9 indicate that a large number of refactorings can be identified even for a small system.
- A subset of the entire set of refactorings need to be carefully chosen because of the following reasons.
  - Some refactorings must be applied together.
    - Example: R1 and R2 are to be applied together.
  - Some refactorings must be applied in certain orders.
    - Example: R1 and R2 must precede R3.
  - Some refactorings can be individually applied, but they must follow an order if applied together.
    - Example: R1 and R8 can be applied in isolation. However, if both of them are to be applied, then R1 must occur before R8.
  - Some refactorings are mutually exclusive.
    - Example: R4 and R6 are mutually exclusive.

Determine which refactorings to apply

- 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**
    - Given a set of refactorings, analyze each pair for conflicts. A pair is said to be conflicting if both of them cannot be applied together.
      - Example: R4 and R6 constitute a conflicting pair.
  - **Sequential dependency analysis**
    - In order to apply a refactoring, one or more refactorings must be applied before.
    - If one refactoring has already been applied, a mutually exclusive refactoring cannot be applied anymore.
      - Example: after applying R1, R2, and R3, R4 becomes applicable. Now, if R4 is applied, then R6 is not applicable anymore.

Ensure that refactoring preserves the software's behavior.

- Ideally, the input/output behavior of a program *after* refactoring is the same as the behavior *before* refactoring.
- In many applications, preservation of non-functional requirements is necessary.
- 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 certain order.
    - For real-time systems, refactorings should preserve temporal constraints.
  - Resource constraints: The software after refactoring does not demand more resources: memory, energy, communication bandwidth, and so on.
  - Safety constraints: It is important that the software does not lose its safety properties after refactoring.

Ensure that refactoring preserves the software's behavior.

- Two pragmatic ways of showing that refactoring preserves the software's behavior.

- Testing
  - Exhaustively test the software *before* and *after* applying refactorings, and compare the observed behavior on a test-by-test basis.

- Verification of preservation of call sequence
  - Ensure that the sequence(s) of method calls are preserved in the refactored program.

Apply the refactorings to chosen entities

- The class diagram of Fig. 7.2(a) has been obtained from Fig. 7.1 by
  - focusing on the classes *FileServer*, PrintServer, and Packet; and
  - applying refactorings R1, R2, and R3.

Apply the refactorings to chosen entities

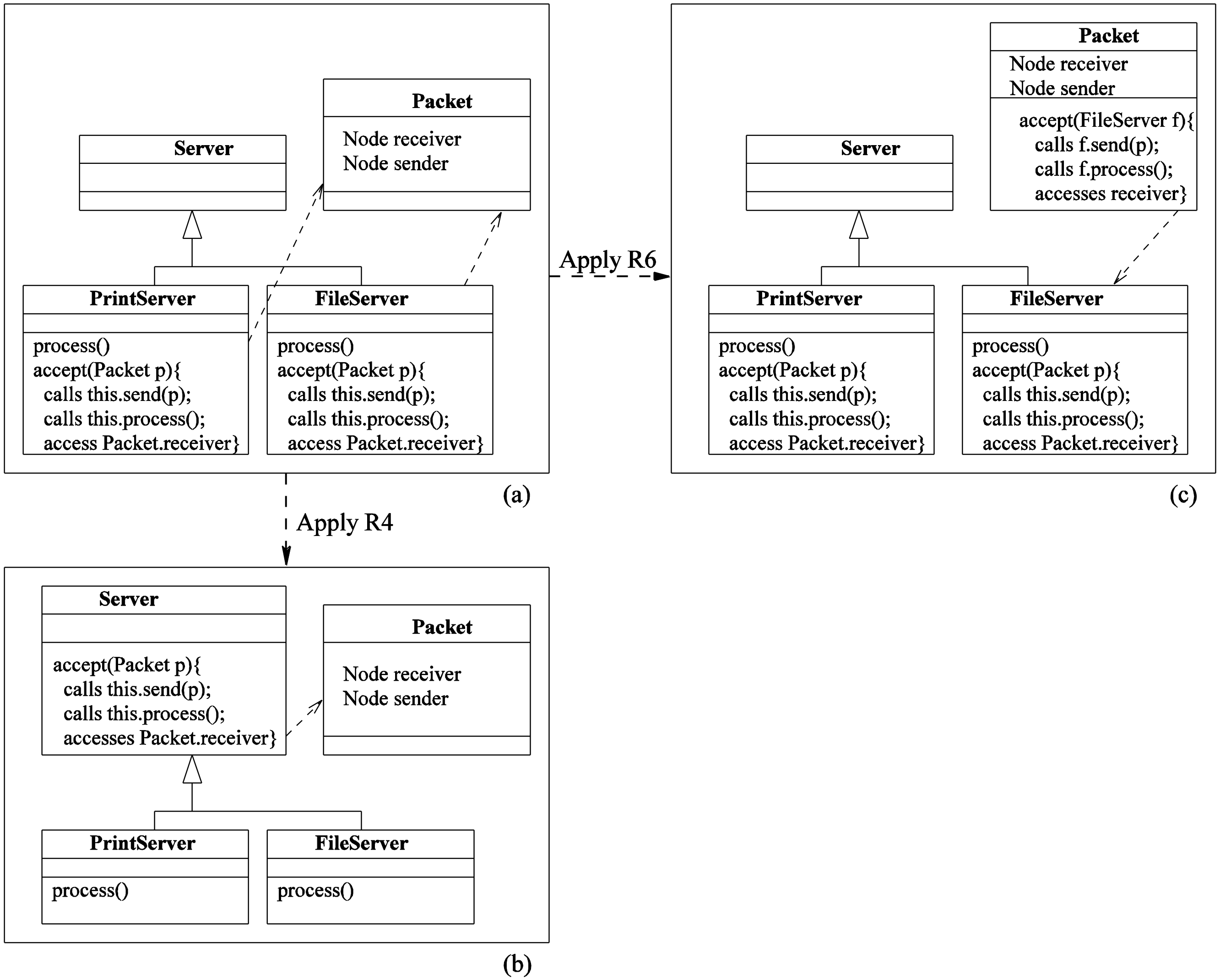

Figure 7.2: Applications of two refactorings [6] (© [2007] Springer).

Evaluate the impacts of the Refactorings on Quality

- Refactorings impact both *internal* and *external* qualities of software.

- Some examples of *internal* qualities of software are
  - size, complexity, coupling, cohesion, and testability
- Some examples of *external* qualities of software are
  - performance, reusability, maintainability, extensibility, robustness, and scalability

Evaluate the impacts of the Refactorings on Quality

- In general, refactoring techniques are highly specialized, with one technique improving a small number of quality attributes.

- For example,
  - some refactorings eliminate code duplication;
  - some raise reusability;
  - some improve performance; and
  - some improve maintainability.

Evaluate the impacts of the Refactorings on Quality

- By measuring the impacts of refactorings on internal qualities, their impacts on external qualities can be measured.
- Example of measuring external qualities
  - 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.
  - 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

- Rather than evaluate the impacts *after applying refactorings*, one selects refactorings such that the program *after* refactoring possesses better quality attributes.
- The concept of soft-goal graph help select refactorings.
- Exmple: A soft-goal graph for quality attribute (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.
- The leaf nodes represent refactoring transformations which contribute positively/negatively to soft-goals which appear above them in the hierarchy.
- …. continued on the following slides.

Maintain consistency

(Continued from the previous slide)

- A partial example of a soft goal graph with one leaf node, namely, *Move*, has been illustrated in Fig. 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.

Maintain consistency

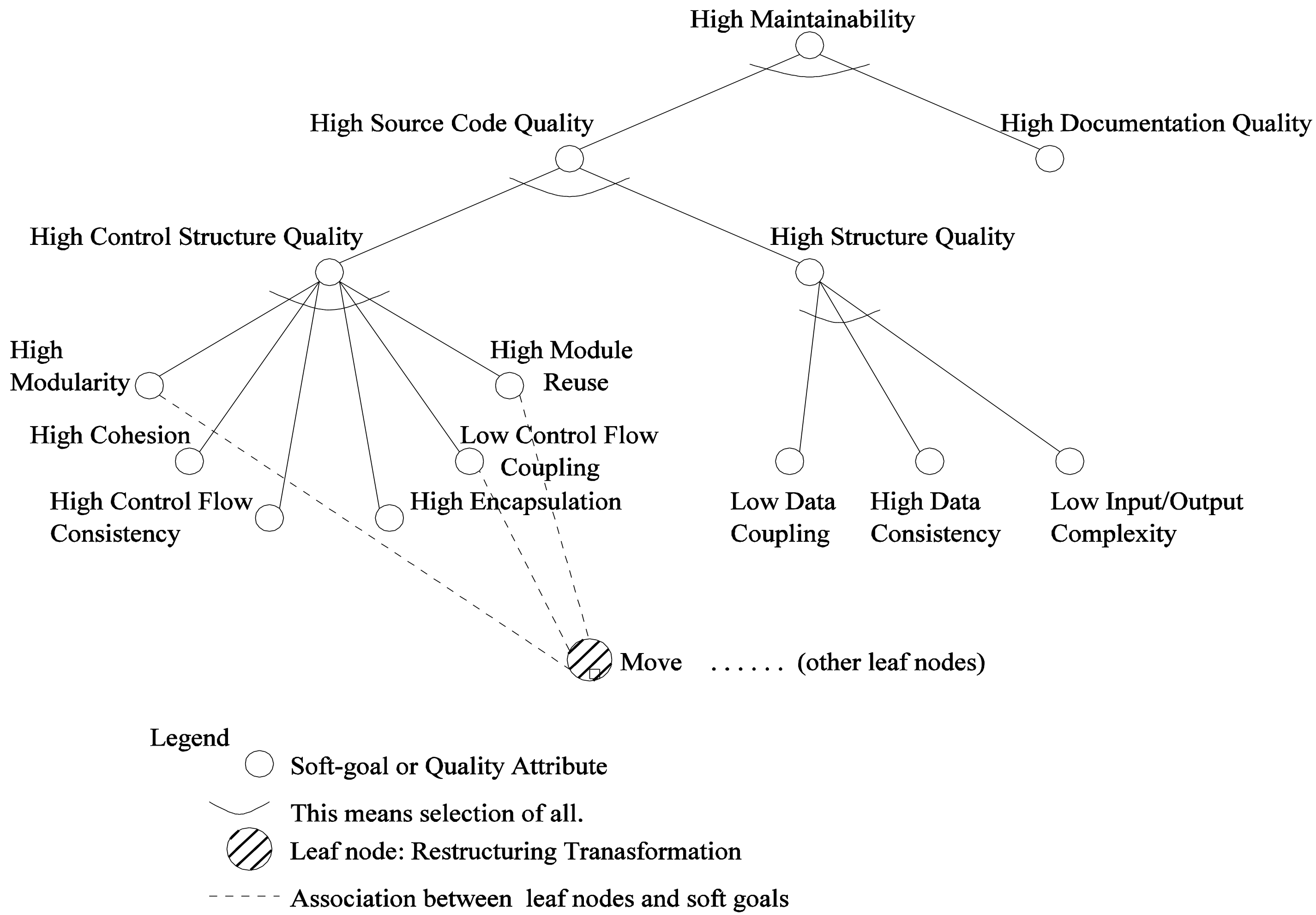

Figure 7.3: An example of a soft goal graph for maintainability, with one leaf node [11] (© [2002] IEEE).

Code Transformation

- The act of changing one program into another
  - from a source language to a target language

- This is possible because of a program's well-defined structure
  - But for validity, we have to be aware of the semantics of each structure

- Used in many areas of software engineering:
  - Compiler construction
  - Software visualization
  - Documentation generation
  - Automatic software renovation

Code Transformation

- Converting to a <u>new language dialect</u>
- Migrating from a <u>procedural language to an object-oriented one</u>, e.g. C to C++
- Adding <u>code comments</u>
- <u>Requirement upgrading</u>
  - e.g. using 4 digits for years instead of 2 (Y2K)
- Structural improvements
  - e.g. changing GOTOs to control structures
- Pretty printing

Code Transformation

- Modify all arithmetic expressions to reduce the number of parentheses using the formula: (a+b)*c = a*c + b*c

```
x := (2+5)*3
```

becomes

```
x := 2*3 + 5*3
```

Two types of transformations

- Translation
  - Source and target language are different
  - Semantics remain the same
- Rephrasing
  - Source and target language are the same
  - Goal is to improve some aspect of the program such as its understandability or performance
  - Semantics might change

Transformation tools

- There are many transformation tools
- Program-Transformation.org lists 90 of them
  - http://www.program-transformation.org/
  - TXL is one of the best
- Most are based on ‘term rewriting’
  - Other solutions use functional programming, lambda calculus, etc.

Web-enabling

- It is possible to take business applications that were designed for direct PC installation, and make them available via a web browser to users across the enterprise. This technique is called **web-enabling** the application
- A common product that allows for web-enabling applications is called **Citrix Metaframe**.
- In contrast to a web-based application, after a user logs into a web-enabled application, the browser no longer is the interface. So, there aren’t any traditional browser links for navigating in these applications, nor is there a “back” button like the browser has.

Web-enabling

Advantage

- The biggest **advantage** of web-enabling legacy business applications via Citrix is that specialized functionality embedded into the application can easily be extended to the enterprise via the web.
- Another **advantage** of web-enabled applications is performance. It is hard for a web-based application to match the snappy performance of a legacy windows client-server application that has been web-enabled via Citrix.

Web-enabling

Disadvantage

- The lack of consistency in the user interface compared to web-based apps. Users must learn the specialized application’s interface which will be different than the standard browser.

- The disadvantage to the IT department for a web-enabled application is increased cost and support since a front-end application like Citrix must be deployed to web-enable the application.

Software Reengineering Strategies

Three strategies that specify the basic steps of reengineering are **rewrite**, **rework**, and **replace**.

Rewrite strategy:

This strategy reflects the principle of alteration. By means of alteration, an operational system is transformed into a new system, while preserving the abstraction level of the original system. For example, the Fortran code of a system can be rewritten in the C language.

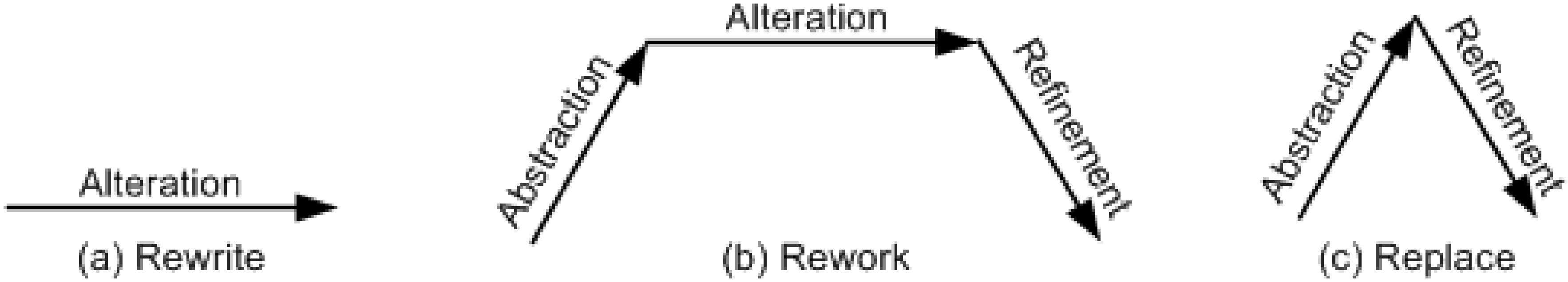

Figure 4.5 Conceptual basis for reengineering strategies © IEEE, 1992

Software Reengineering Strategies

Rework strategy:

- The rework strategy applies all the three principles.
- Let the goal of a reengineering project is to replace the unstructured control flow constructs, namely GOTOs, with more commonly used structured constructs, say, a “for” loop.
- A classical, rework strategy based approach is as follows:
  - Application of abstraction: By parsing the code, generate a control-flow graph (CFG) for the given system.
  - Application of alteration: Apply a restructuring algorithm to the control-flow graph to produce a structured control-flow graph.
  - Application of refinement: Translate the new, structured control-flow graph back into the original programming language.

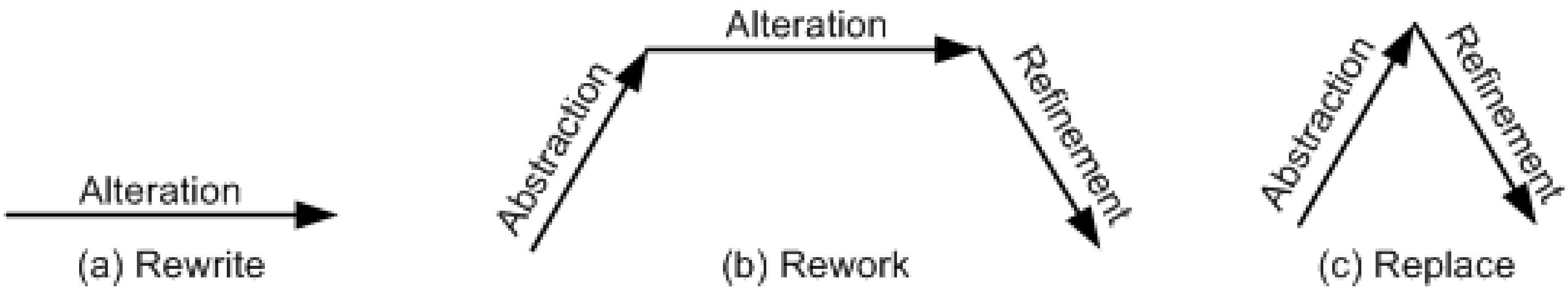

Figure 4.5 Conceptual basis for reengineering strategies © IEEE, 1992

Software Reengineering Strategies

Replace strategy:

- The replace strategy applies two principles, namely, abstraction and refinement.
- To change a certain characteristic of a system:
  - (i) the system is reconstructed at a higher level of abstraction by hiding the details of the characteristic; and
  - (ii) a suitable representation for the target system is generated at a lower level of abstraction by applying refinement.
- Let us reconsider the GOTO example. By means of abstraction, a program is represented at a higher level without using control flow concepts.
- Next, by means of refinement, the system is represented at a lower level of abstraction with a new structured control flow.

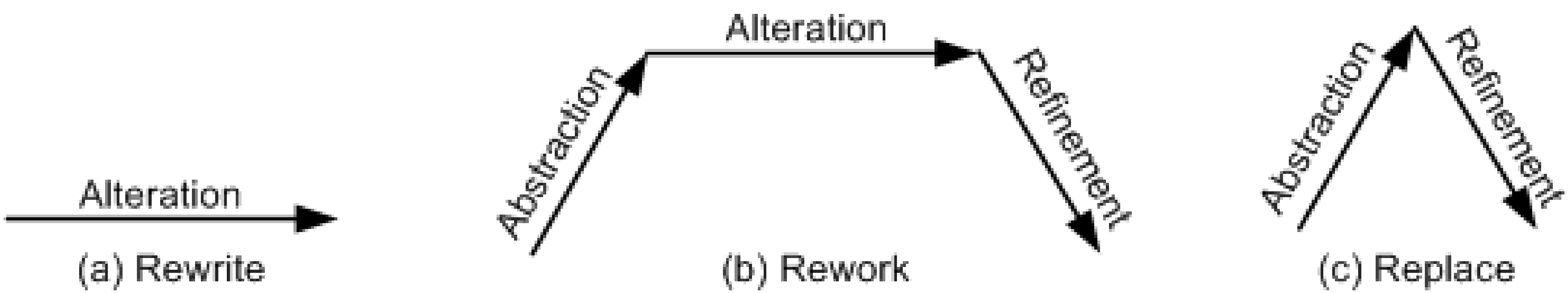

Figure 4.5 Conceptual basis for reengineering strategies © IEEE, 1992

References

- Alain April, Alain Abran (2008), Software Maintenance Management Evaluation and Continuous Improvement.
- Priyadarshi Tripathy, Kshirasagar, 2015, Naik, Software evolution and maintenance : a practitioner's approach.
- Penny Grub, Armstrong A Takang, Software Maintenance Concepts and Practice, 2^{nd} edition