Refactoring - General Idea and Process

Idaho State University | Computer Science

Isaac Griffith

CS 4423 and CS 5523
Department of Computer Science
Idaho State University

ROAR

Idaho State University
Computer Science

Outcomes

After today's lecture you will be able to:

- Understand and describe the general idea of refactoring
- Understand and describe the process of refactoring
- Understand and be able to evaluate the impacts of refactoring on quality

ROAR

Refactoring

CS 4423/5523

ROAR

General Idea

- 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  - Out-of-date documentation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

General Idea

- Decrease the complexity of software by improving its internal quality by restructuring the software.
- Restructuring applied on object-oriented software is called refactoring.
- Restructuring means reorganizing software (source code + documentation) to give it a different look, or structure.

General Idea

- Source code is restructured to improve some of its non-functional requirements:
  - Readability
  - Extensibility
  - Maintainability
  - Modularity
- Restructuring does not modify the software's functionalities.
- Restructuring can be performed while adding new features.

General Idea

- Software restructuring is informally stated as the modifications of software to make it
  - easier to understand;
  - easier to change;
  - easier to change its documentation;
  - less susceptible to faults when changes are made to it.

General Idea

- 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

General Idea

- Developers and managers need to be aware of restructuring for the following reasons
  - better understandability
  - keep pace with new structures
  - better reliability
  - longer lifetime
  - automated analysis

General Idea

- Characteristics of restructuring and refactoring
  - The objective of restructuring and refactoring is to improve the internal and external values of software.
  - Restructuring preserves the external behavior of the original program.
  - Restructuring can be performed without adding new requirements.
  - Restructuring generally produces a program in the same language.
    - Example: a C program is restructured into another C program.

Refactoring Process

CS 4423/5523

Refactoring Process Activities

- 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

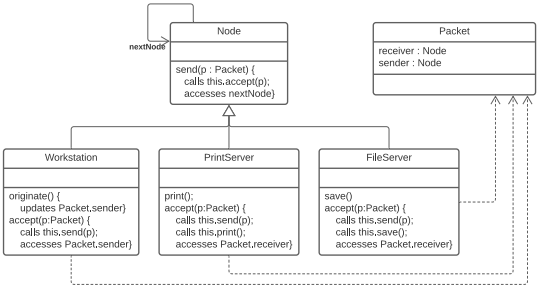
- The concept of code smell is applied to source code to detect what should be refactored.
- A code smell is any symptom in source code that possibly indicates a deeper problem.
- Examples of code smell are:
  - duplicate code;
  - long parameter list;
  - long methods;
  - large classes;
  - message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

Idaho State University
Computer Science

Determine refactorings to apply

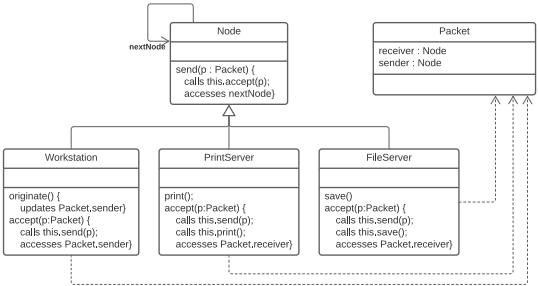


- R1: Rename method `print` to `process` in class `PrintServer`.
- R2: Rename method `save` to `process` in class `FileServer`. (R1 and R2 are to be done together.)
- R3: Create a superclass `Server` from `PrintServer` and `FileServer`.
- R4: Pull up method `accept` from `PrintServer` and `FileServer` to the superclass `Server`.
- R5: Move method `accept` from `PrintServer` to class `Packet`, so that data packets themselves will decide what actions to take.

ROAR

Idaho State University
Computer Science

Determine refactorings to apply



- 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 saved.

ROAR

Determine 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 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 preservation of 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.

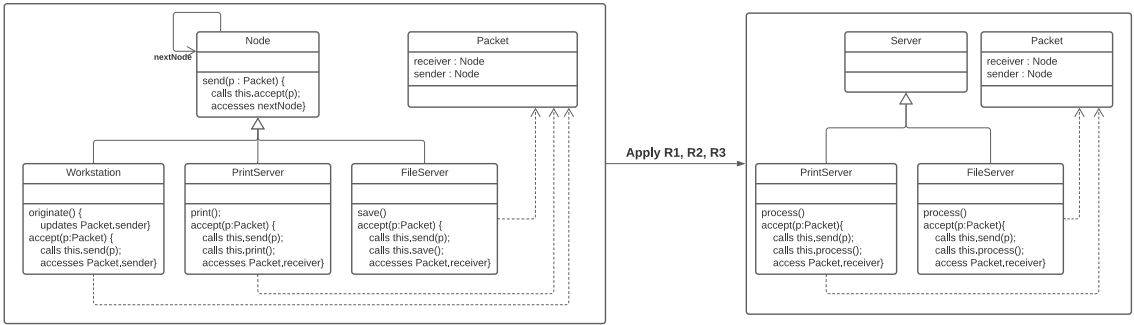
Ensure preservation of behavior

- 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.

Idaho State University
Computer Science

Apply the refactorings

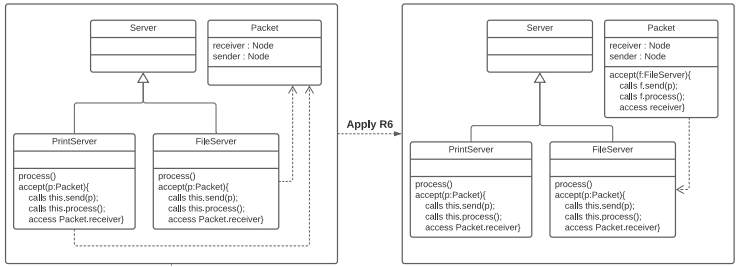
- The following transformation is obtained by:
  - focusing on the classes `FileServer`, `PrintServer`, and `Packet`
  - applying refactorings R1, R2, and R3.



- R1: Rename method `print` to `process` in class `PrintServer`.
- R2: Rename method `save` to `process` in class `FileServer`.
- R3: Create a superclass `Server` from `PrintServer` and `FileServer`.

ROAR

Apply the refactorings



- R4: Pull up method `accept` from `PrintServer` and `FileServer` to the superclass `Server`.
- R6: Move method `accept` from `FileServer` to `Packet`.

Evaluate impacts on quality

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

- **Internal** qualities of software include
  - size
  - complexity
  - coupling
  - cohesion
  - testability

- **External** qualities of software include:
  - performance
  - reusability
  - maintainability
  - extensibility
  - robustness
  - scalability

Evaluate impacts 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 impacts on Quality

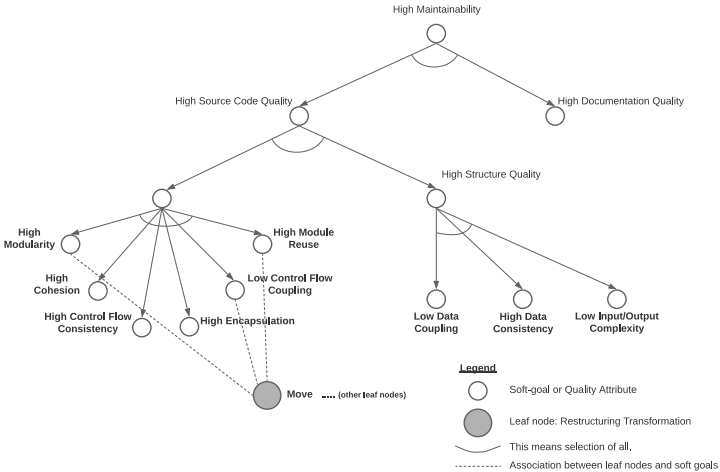
- By measuring the impacts of refactorings on internal qualities, their impacts on external qualities can be measured.
- Example of measuring external qualities
  - Some examples of software metrics are coupling, cohesion, and size.
  - Decreased coupling, increased cohesion, and decreased size are likely 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 a **soft-goal graph** help select refactorings.

Example

- A soft-goal graph for quality attribute (maintainability) rooted at high maintainability (desired change).

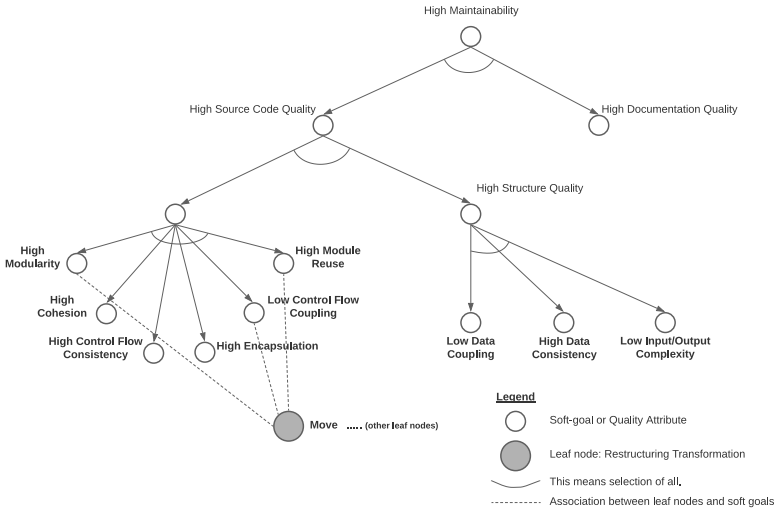


- The internal nodes represent successive refinements of the attribute and are basically the soft goals.
- The leaf nodes represent refactoring transformations which contribute positively/negatively to soft-goals which appear above them in the hierarchy.

Idaho State University
Computer Science

Example

- A partial example of a soft goal graph with one leaf node, **Move**

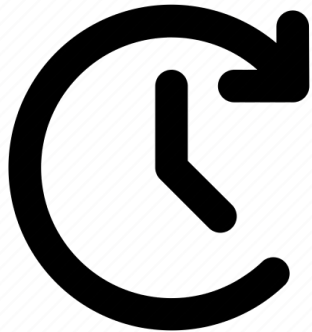


- The dotted lines between **Move** and three soft goals
  - High Modularity
  - High Module Reuse
  - Low Control Flow Coupling
- Implies that the Move transformation impacts those three soft goals

ROAR

For Next Time

- Review EVO Chapter 7.1 - 7.2
- Read EVO Chapter 7.3 - 7.4
- Watch Lecture 18



Are there any questions?