City University of New York (CUNY)
CUNY Academic Works

Open Educational Resources Hunter College

2023

Reengineering and Refactoring

Raffi T. Khatchadourian
CUNY Hunter College

How does access to this work benefit you? Let us know!

More information about this work at: https://academicworks.cuny.edu/hc_oers/39
Discover additional works at: https://academicworks.cuny.edu

This work is made publicly available by the City University of New York (CUNY).
Contact: AcademicWorks@cuny.edu

Programming Languages/Software Engineering

Reengineering and Refactoring

Credits: Slide content may contain repurposed material originally by Ian Sommerville.

Inaugural Work on Refactoring

- William F. Opdyke. 1992. Refactoring Object-Oriented Frameworks. Ph.D. Dissertation. University of Illinois at Urbana-Champaign, Champaign, IL, USA.
- 1999. Refactoring: Improving the Design of Existing Code. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.
- Joshua Kerievsky. 2004. Refactoring to Patterns. Pearson Higher Education.

Type Inference-based Refactoring

- Maria Anna G. Gaitani, Vassilis E. Zafeiris, N.A. Diamantidis, and E.A. Giakoumakis. 2015. Automated refactoring to the Null Object design pattern. Inf. Softw. Technol. 59, C (March 2015), 33-52.
- Raffi Khatchadourian. 2017. Automated refactoring of legacy Java software to enumerated types. Automated Software Engg. 24, 4 (December 2017), 757-787.

Constraint-based Refactoring

- Frank Tip, Robert M. Fuhrer, Adam Kieżun, Michael D. Ernst, Ittai Balaban, and Bjorn De Sutter. 2011. Refactoring using type constraints. ACM Trans. Program. Lang. Syst. 33, 3, Article 9 (May 2011), 47 pages.
- Friedrich Steimann. 2018. Constraint-Based Refactoring. ACM Trans. Program. Lang. Syst. 40, 1, Article 2 (January 2018), 40 pages.
- Raffi Khatchadourian and Hidehiko Masuhara. 2017. Automated refactoring of legacy Java software to default methods. In Proceedings of the 39th International Conference on Software Engineering (ICSE '17). IEEE Press, Piscataway, NJ, USA, 82-93.

Software reengineering

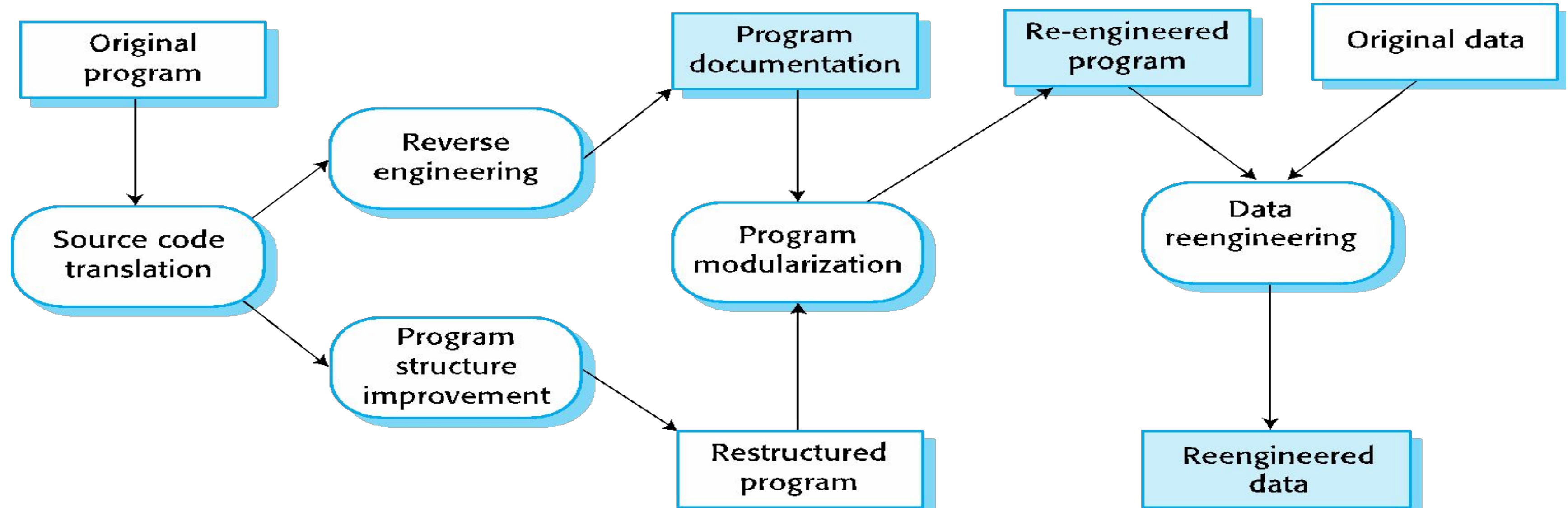
- Restructuring or rewriting part or all of a legacy system without changing its functionality.
- Applicable where some but not all subsystems of a larger system require frequent maintenance.
- Reengineering involves adding effort to make them easier to maintain. The system may be re-structured and re-documented.

Advantages of reengineering

- Reduced risk
  - There is a high risk in new software development. There may be development problems, staffing problems and specification problems.
- Reduced cost
  - The cost of reengineering is often significantly less than the costs of developing new software.

Credits: Slide content may contain repurposed material originally by Ian Sommerville.

The reengineering process

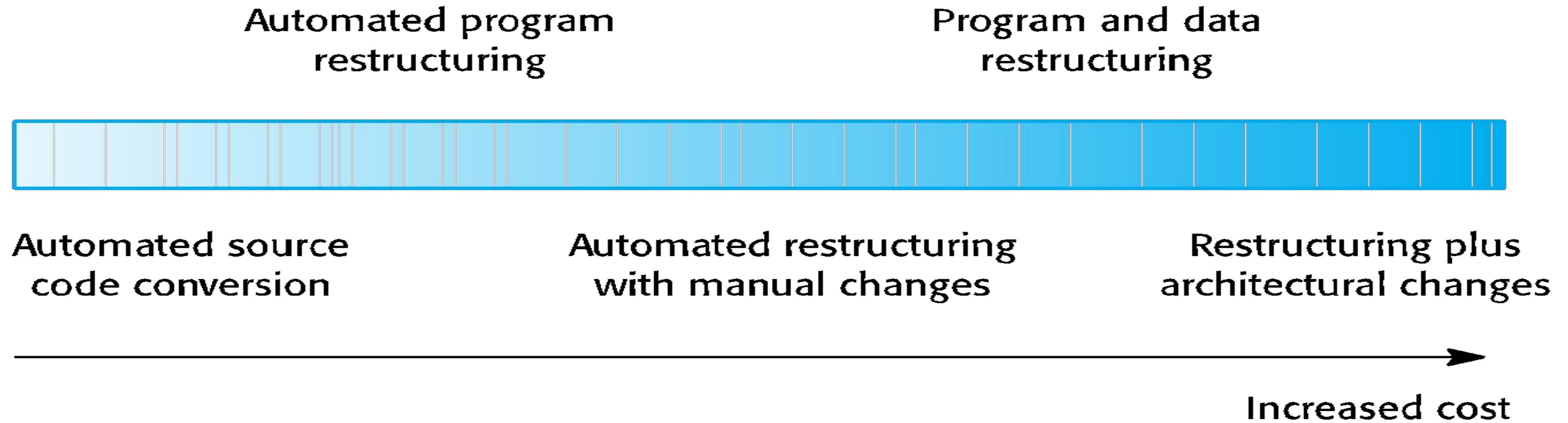


Reengineering process activities

- Source code translation
  - Convert code to a new language.
- Reverse engineering
  - Analyse the program to understand it;
- Program structure improvement
  - Restructure automatically for understandability;
- Program modularisation
  - Reorganise the program structure;

Credits: Slide content may contain repurposed material originally by Ian Sommerville.

Reengineering approaches



Reengineering cost factors

- ✧ The quality of the software to be reengineered.
- ✧ The tool support available for re-engineering.
- ✧ The extent of the data conversion which is required.
- ✧ The availability of expert staff for reengineering.
  - This can be a problem with old systems based on technology that is no longer widely used.

Credits: Slide content may contain repurposed material originally by Ian Sommerville.

Refactoring

- Refactoring is the process of making improvements to a program to slow down degradation through change.
- You can think of refactoring as ‘preventative maintenance’ that reduces the problems of future change.
- Refactoring involves modifying a program to improve its structure, reduce its complexity or make it easier to understand.
- When you refactor a program, you should not add functionality but rather concentrate on program improvement.

Refactoring and reengineering

- Re-engineering takes place after a system has been maintained for some time and maintenance costs are increasing. You use automated tools to process and reengineer a legacy system to create a new system that is more maintainable.
- Refactoring is a continuous process of improvement throughout the development and evolution process. It is intended to avoid the structure and code degradation that increases the costs and difficulties of maintaining a system.

Research on Refactoring

- Two parts: Preconditions and Transformations.
- Preconditions are conditions that must be met in order to **guarantee** semantics preservation.
- Transformation is done after preconditions are met.
- Preconditions are usually the most challenging but transforming in a way that is similar to what a human would do can also be non-trivial.

Type Inference-based Refactoring

- Jens Palsberg and Michael I. Schwartzbach. 1994. Object-Oriented Type Systems. John Wiley and Sons Ltd., Chichester, UK.
- Use a type-checking approach, which is a form of type inference.

Credits: Slide content may contain repurposed material originally by Ian Sommerville.

$\mathcal{P}$	original program
$\phi(\mathcal{P})$	$\{f \mid f$ is a static final field of primitive type in $\mathcal{P}\}$
$\mu(\mathcal{P})$	$\{m \mid m$ is a method in $\mathcal{P}\}$
$\upsilon(\mathcal{P})$	$\{l \mid l$ is a variable in $\mathcal{P}\}$
α	variable, field, method
α_{ctxt}	*context* in which α may occur

Identifiers.

function $EA(\alpha,\mathrm{ID})$

1: **return** $EA(\alpha, Parent(\mathrm{ID}))$

Credits: Slide content may contain repurposed material originally by Ian Sommerville.

Equality expressions

function $EA(\alpha, \text{EXP}_1 == \text{EXP}_2)$

1: **return** $ED(\text{EXP}_1) \wedge ED(\text{EXP}_2)$

Array access/creation expressions

function $EA(\alpha, \text{EXP}_1[\text{EXP}_2])$

1: **if** $contains(\text{EXP}_2, \alpha)$ **then**
2: **return false**
3: **else**
4: **return** $EA(\alpha, Parent(\text{EXP}_1))$
5: **end if**

Conditional expressions

function $EA(\alpha, \textsc{Exp}_1 \; ? \; \textsc{Exp}_2 : \textsc{Exp}_3)$

1: **if** $contains(\textsc{Exp}_2, \alpha) \vee contains(\textsc{Exp}_3, \alpha)$ **then**
2:   **return** $EA(\alpha, Parent(\textsc{Exp}_1 \; ? \; \textsc{Exp}_2 : \textsc{Exp}_3))$
3: **else**
4:   **return** **true**
5: **end if**

Credits: Slide content may contain repurposed material originally by Ian Sommerville.

Integer literals

function $ED(\text{IL})$

1: **return false**

Identifiers

function $ED(\text{ID})$

1: **return true**

Parenthesized expressions

function $ED((\text{E}\textsc{xp}))$

1: **return** $ED(\text{E}\textsc{xp})$

Assignment expressions

function $ED(\text{Exp}_1 = \text{Exp}_2)$

1: **return** $ED(\text{Exp}_1) \wedge ED(\text{Exp}_2)$

Infix addition expressions

function $ED(\textsc{Exp}_1 + \textsc{Exp}_2)$

1: **return false**

function $Enumerizable(C)$

1: $W \leftarrow C$ */* seed the worklist with the input constants */*
2: $N \leftarrow \emptyset$ */* the non-enumerizable set list, initially empty */*
3: **for all** $c \in C$ **do**
4: $MakeSet(c)$ */* init the union-find data structure */*
5: **end for**
6: **while** $W \neq \emptyset$ **do**
7: */* remove an element from the worklist */*
8: $\alpha \leftarrow e \mid e \in W$
9: $W \leftarrow W \setminus \{\alpha\}$
10: **for all** $\alpha_{ctxt} \in Contexts(\alpha, \mathcal{P})$ **do**
11: **if** $\neg isEnumerizableContext(\alpha, \alpha_{ctxt})$ **then**
12: */* add to the non-enumerizable list */*
13: $N \leftarrow N \cup \{\alpha\}$
14: **break**
15: **end if**
16: */* extract entities to be enumerized due to* α **/*
17: **for all** $\hat{\alpha} \in Extract(\alpha, \alpha_{ctxt})$ **do**
18: **if** $Find(\hat{\alpha}) = \emptyset$ **then**
19: $MakeSet(\hat{\alpha})$
20: $W \leftarrow W \cup \{\hat{\alpha}\}$
21: **end if**
22: $Union(Find(\alpha), Find(\hat{\alpha}))$
23: **end for**
24: **end for**
25: **end while**
26: $F \leftarrow AllSets()$ */* the sets to be returned */*
27: **for all** $\alpha' \in N$ **do**
28: $F \leftarrow F \setminus Find(\alpha')$ */* remove nonenum sets */*
29: **end for**
30: **return** F */* all sets minus the non-enumerizable sets */*

Worklist-based Algorithm

Credits: Slide content may contain repurposed material originally by Ian Sommerville.

Constraint-Based Refactoring

- Use constrain “templates” that encode the semantics of the language in some sense.
- Type constraints express type relationships between various entities in the program.
- Generate constraints using templates on an actual program.
- Receive a *system* of constraints.
- Can solve the system for useful information.
- Can use the constraints to check that a transformation is semantics-preserving (i.e., no violations).