SOFTWARE RE-ENGINEERING

LEGACY SYSTEM

- A legacy system is an old system which is valuable for the company which often developed and owns it.

- It is the phase out stage of the software
- Bennett used the following definition:
  "large software systems that we don't know how to cope with but that are vital to our organization."

- Similarly, Brodie and Stonebraker: define a legacy system as
  "any information system that significantly resists modification and evolution to meet new and constantly changing business requirements."

LEGACY SYSTEM

- There are a number of options available to manage legacy systems. Typical solution include:
  - Freeze: The organization decides no further work on the legacy system should be performed.
  - Outsource: An organization may decide that supporting software is not core business, and may outsource it to a specialist organization offering this service.
  - Carry on maintenance: Despite all the problems of support, the organization decides to carry on maintenance for another period.
  - Discard and redevelop: Throw all the software away and redevelop the application once again from scratch.
  - Wrap: It is a black-box modernization technique – surrounds the legacy system with a software layer that hides the unwanted complexity of the existing data, individual programs, application systems, and interfaces with the new interfaces.
  - Migrate: Legacy system migration basically moves an existing, operational system to a new platform, retaining the legacy system’s functionality and causing minimal disruption to the existing operational business environment as possible.

IMPACT ANALYSIS

- Impact analysis is the task of estimating the parts of the software that can be affected if a proposed change request is made.

- Impact analysis techniques can be partitioned into two classes:
  - Traceability analysis In this approach the high-level artifacts such as requirements, design, code and test cases related to the feature to be changed are identified.
  - A model of inter-artifacts such that each artifact in one level links to other artifacts is constructed, which helps to locate a pieces of design, code and test cases that need to be maintained.

IMPACT ANALYSIS

- Dependency (or source-code) analysis Dependency analysis attempt to assess the affects of change on semantic dependencies between program entities.
- This is achieved by identifying the syntactic dependencies that may signal the presence of such semantic dependencies.
    - The two dependency-based impact analysis techniques are:
    - call graph based analysis and
    - dependency graph based analysis.

IMPACT ANALYSIS

- Two additional notions related to impact analysis are very common among practitioners:
  - Ripple effect.
  - Change propagation.
- Ripple effect analysis measures the impact, or how likely it is that a change to a particular module may cause problems in the rest of the program.
- Measuring ripple effect can provide knowledge about the system as a whole through its evolution:
  - how much its complexity has increased or decreased since the previous version,
  - how complex individual parts of a system are in relation to other parts of the system, and
  - to look at the effect that a new module has on the complexity of a system as a whole when it is added.
- The change propagation activity ensures that a change made in one component is propagated properly throughout the entire system.

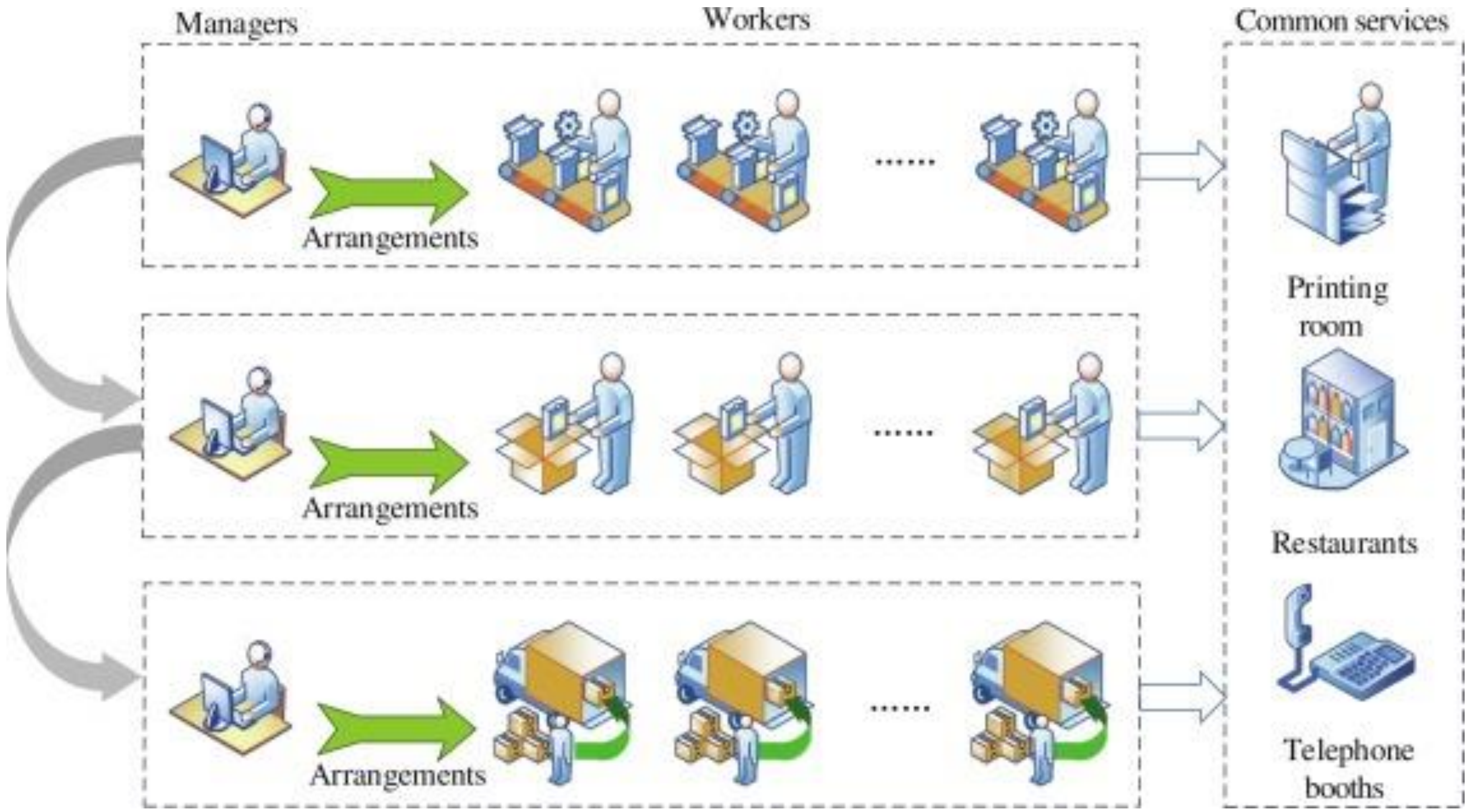
PROGRAM COMPREHENSION

- Program understanding or comprehension is

  "the task of building mental models of an underlying software system at various abstraction levels, ranging from models of the code itself to ones of the underlying application domain, for software maintenance, evolution and re-engineering purposes."

- A mental model describes a programmer's mental representation of the program to be comprehended.
- Program comprehension deals with the cognitive processes involved in constructing a mental model of the program.

ABSTRACT MODEL



PROGRAM COMPREHENSION

- A common element of such cognitive models is generating hypotheses and investigating whether they hold or must be rejected.
  - Hypotheses are a way to understand code in an incremental manner.
  - After some understanding of the code, the programmer forms a hypothesis and verifies it by reading code.
  - By continuously formulating new hypotheses and verifying them, the programmer understands more and more code and in increasing details.

PROGRAM COMPREHENSION

- Several strategies can be used to arrive at relevant hypotheses such as:
  - bottom up (starting from the code).
  - top down (starting from high-level goal).
  - opportunistic combinations of the two.
- A strategy is formulated by identifying actions to achieve a goal.

PROGRAM COMPREHENSION

- Strategies guide two mechanisms, namely chunking and cross-referencing to produce higher-level abstraction structures.
  - Chunking creates new, higher level abstraction structures from lower level structures.
  - Cross-referencing means being able to make links elements of different abstraction levels.
- Chunking and cross-referencing helps in building mental model of the program under study at different levels of abstractions.

REENGINEERING CONCEPTS

- **Principle of Abstraction:** enables software engineer to reduce the complexity of understanding a system by:
  - (i) focusing on the more significant information about the system; and
  - (ii) Hiding the irrelevant details at the moment.

- The level of abstraction of the representation of a system can be gradually increased by successively replacing the details with abstract information.

- By means of abstraction one can produce a view that focuses on selected system characteristics by hiding information about other characteristics.

- **Principle of refinement:**
- Refinement is the reverse of abstraction

- The level of abstraction of the representation of the system is gradually decreased by successively replacing some aspects of the system with more details.

REENGINEERING CONCEPTS

- **Reverse engineering** of software systems is a process comprising the following steps:

  (i) analyze the software to determine its components and the relationships among the components,

  (ii) represent the system at a higher level of abstraction or in another form.

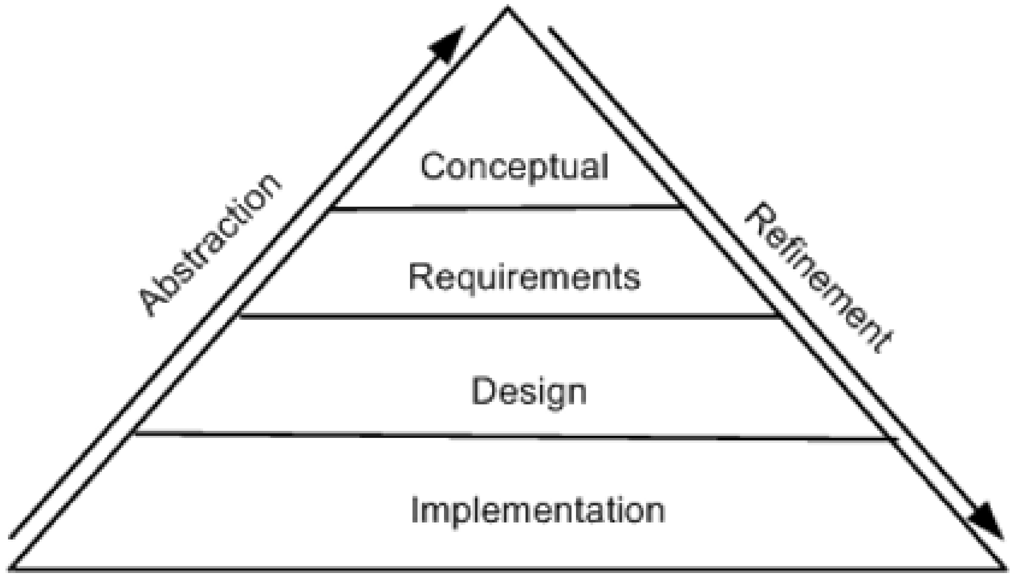
- **Decompilation** is an example of Reverse Engineering, in which object code is translated into a high-level program.

REENGINEERING CONCEPTS

- The concepts of abstraction and refinement are used to create models of software development as sequences of phases, where the phases map to specific levels of **abstraction** or **refinement**, as shown in Figure.

The abstraction process:

how? !
what & how? !
what? !
why?



levels of abstraction and refinement

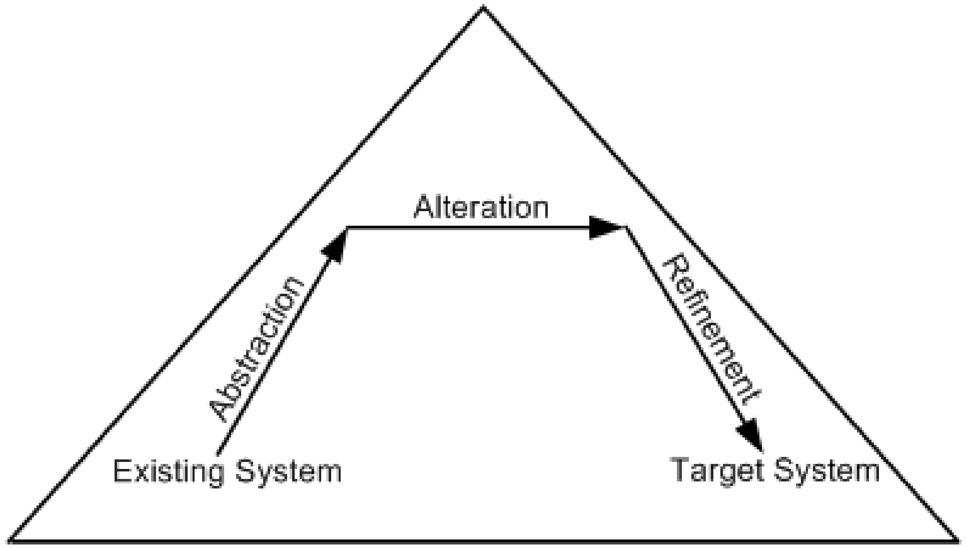
The refinement process:

why? !
what? !
what & how? !
how?

REENGINEERING CONCEPTS

- An optional principle called **alteration** underlies many reengineering methods.
- **Principle of alteration:** The making of some changes to a system representation is known as alteration. Alteration does not involve any change to the degree of abstraction, and it does not involve modification, deletion, and addition of information.

• **Reengineering principles** are represented by means of arrows. Abstraction is represented by an up-arrow, alteration is represented by a horizontal arrow, and refinement by a down-arrow.
• The arrows depicting refinement and abstraction are slanted, thereby indicating the increase and decrease, respectively, of system information.
• It may be noted that alteration is non-essential for reengineering



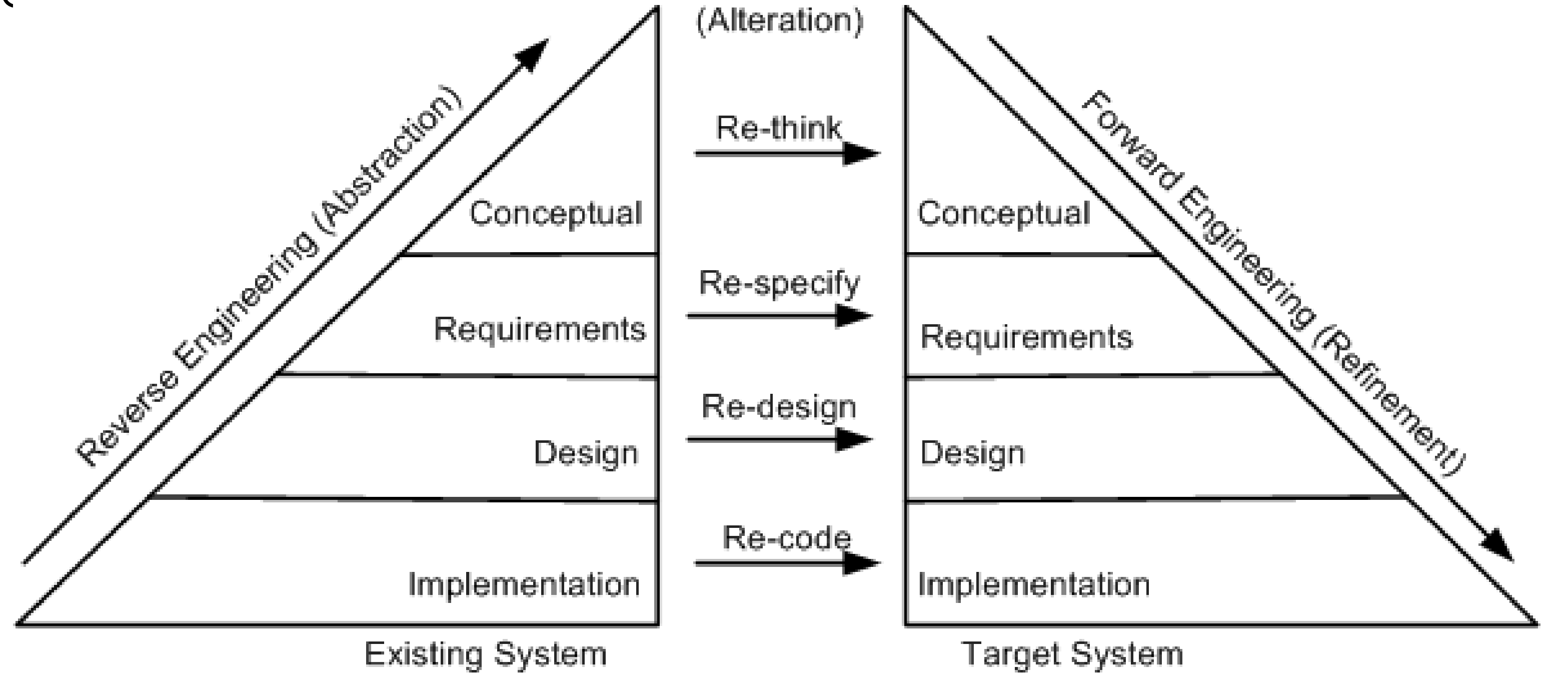
A GENERAL MODEL FOR SOFTWARE REENGINEERING

- The reengineering process accepts as input the existing code of a system and produces the code of the renovated system.
- The reengineering process may be as straightforward as translating with a tool the source code from the given language to source code in another language.
- For example, a program written in C can be translated into a new program in C++/JAVA.

- The reengineering process may be very complex as explained below:
  - recreate a design from the existing source code.
  - find the requirements of the system being reengineered.
  - compare the existing requirements with the new ones.
  - remove those requirements that are not needed in the renovated system.
  - make a new design of the desired system.
  - code the new system.

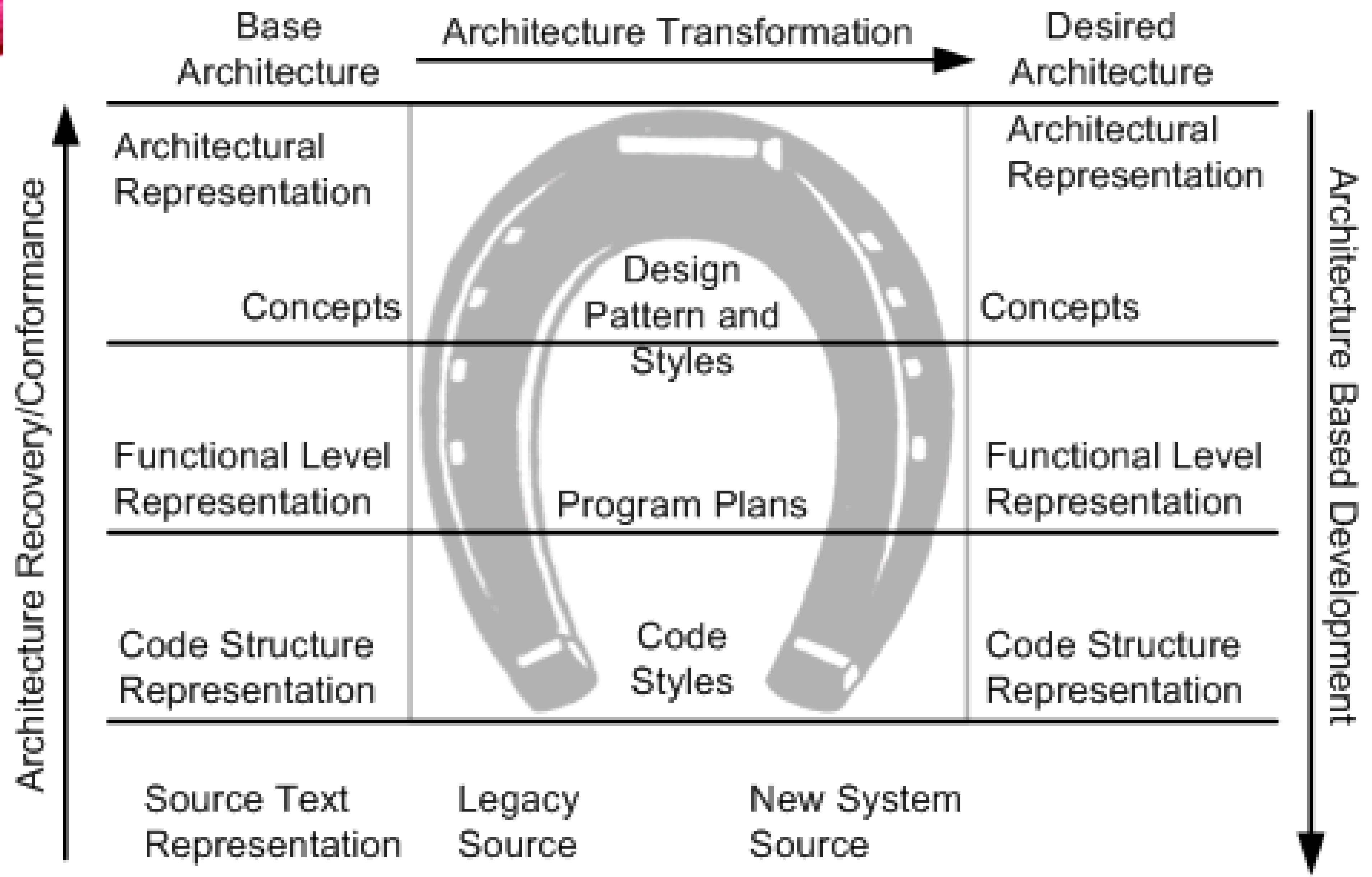
GENERAL MODEL OF SOFTWARE REENGINEERING

- The model in Figure, proposed by Eric J. Byrne suggests that reengineering is a sequence of three activities:
  - **reverse engineering, re-design, and forward engineering**
  - strongly founded in three principles, namely, abstraction, alteration, and refinement



HORSESHOE

- A visual metaphor called horseshoe, was developed by Kazman et al. to describe a three-step architectural reengineering process.
- Three distinct segments of the horseshoe are the left side, the top part, and the right side. Those three parts denote the three steps of the reengineering process.


Base Architecture
Architecture Transformation
Desired Architecture
Architecture Recovery/Conformance
Architecture Based Development
Architectural Representation
Concepts
Design Pattern and Styles
Architectural Representation
Concepts
Functional Level Representation
Program Plans
Functional Level Representation
Code Structure Representation
Code Styles
Code Structure Representation
Source Text Representation
Legacy Source
New System Source

TYPES OF CHANGE

i) Recode: Implementation characteristics of the source program are changed by re-coding it.

- **Rehosting:** reengineering of source code without addition or reduction of features. It is most effective when the user is satisfied with the system's functionality, but looks for better qualities of the system.

- **Rephrasing:** Changes are done by keeping the program in the same language (**normalization**, **optimization**, **refactoring**, and **renovation)**

- **Translation:** a program is transformed into a program in a different language (**compilation**, **decompilation**, and **migration)**

ii) Redesign: The design characteristics of the software are altered by re-designing the system. Common changes to the software design include:

- Restructuring the architecture;
- Modifying the data model of the system; and
- Replacing a procedure or an algorithm with a more efficient one.

iii) Respecify: This means changing the requirement characteristics of the system in two ways:

- Change the form of the requirements, and
- Change the scope of the requirements.

iv) Rethink

- Manipulating the concepts embodied in an existing system to create a system that operates in a different problem domain.
- It involves changing the conceptual characteristics of the system, and it can lead to the system being changed in a fundamental way.
- Moving from the development of an ordinary cellular phone to the development of smartphone system is an example of Re-think.