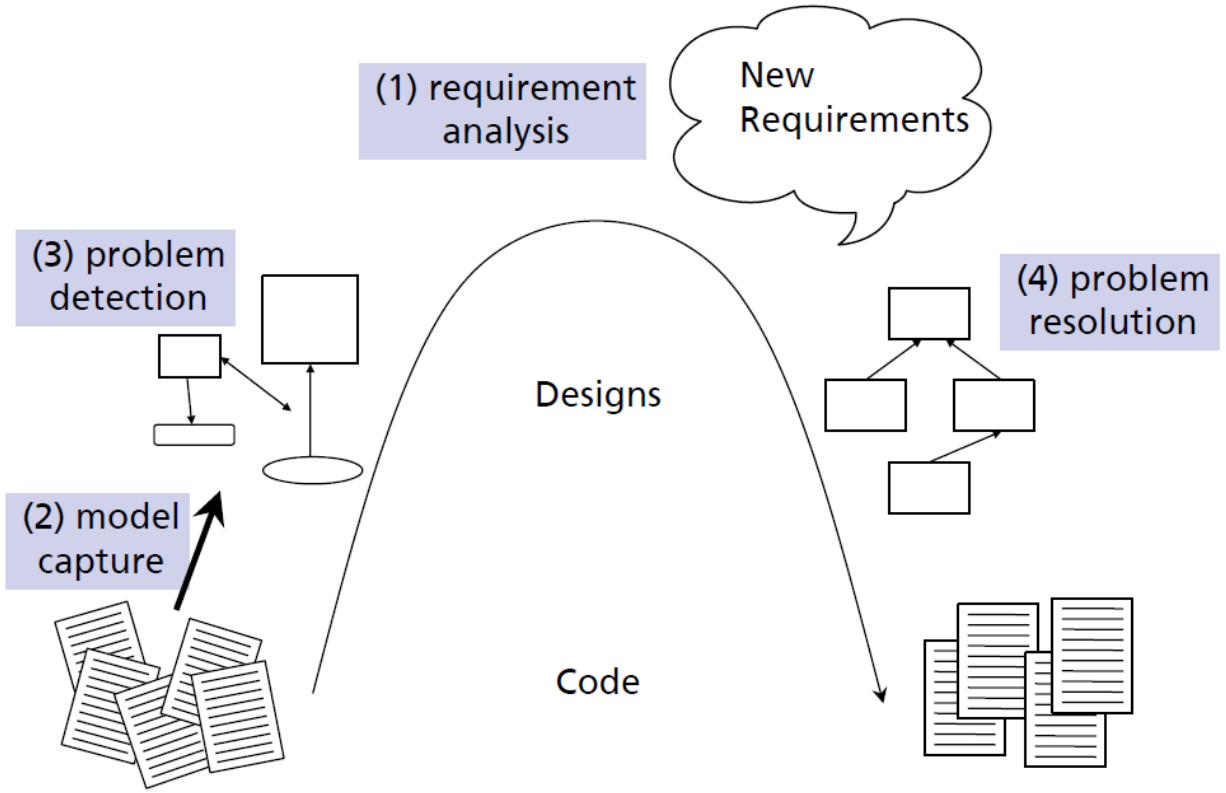
SOFTWARE RE-ENGINEERING

RE-ENGINEERING CYCLE



REENGINEERING

- **Reengineering is a mix of:**
- coarsegrained,
- architectural problems,
- fine-grained,
- design problems.

THE MOST COMMON FINE-GRAIN PROBLEMS IN OO SOFTWARE :

- Misuse of inheritance: for composition, code reuse rather than polymorphism
- Missing inheritance: duplicated code, and case statements to select behavior
- Misplaced operations: unexploited cohesion—operations outside instead

of inside classes

- Violation of encapsulation: explicit type-casting, C++ “friends” .
- Class abuse: lack of cohesion—classes as namespaces

- you will be preparing the code base for the reengineering activity
- by developing exhaustive test cases for all the parts of the system that you plan to change or replace.
- Repaired Vs replaced decision.

REPAIRED VS REPLACED

- According to Chikofsky and Cross:

"Restructuring is the transformation from one representation form to another at the same relative abstraction level, while Preserving the system's external behavior."

"Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure."

WHY PATTERNS?

- A pattern is a recurring motif
- Event
- Structure
- Design
- Document
- Practice
- Language Implementation

SETTING DIRECTION

- What are the goals of the project?
- Find Go/No-Go decision

FIRST CONTACT

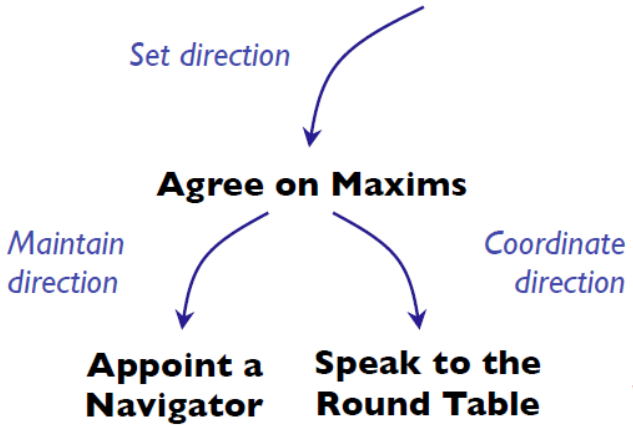
- You are facing a system that is completely new to you and within hours/days you should determine:

- Whether the software is still viable

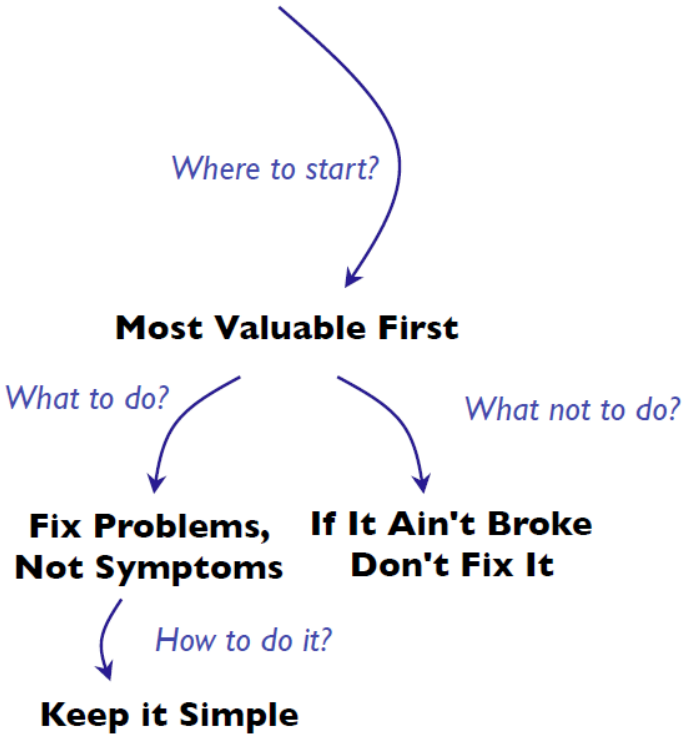
- A plan of work

- A cost-estimation

SETTING DIRECTION PATTERNS



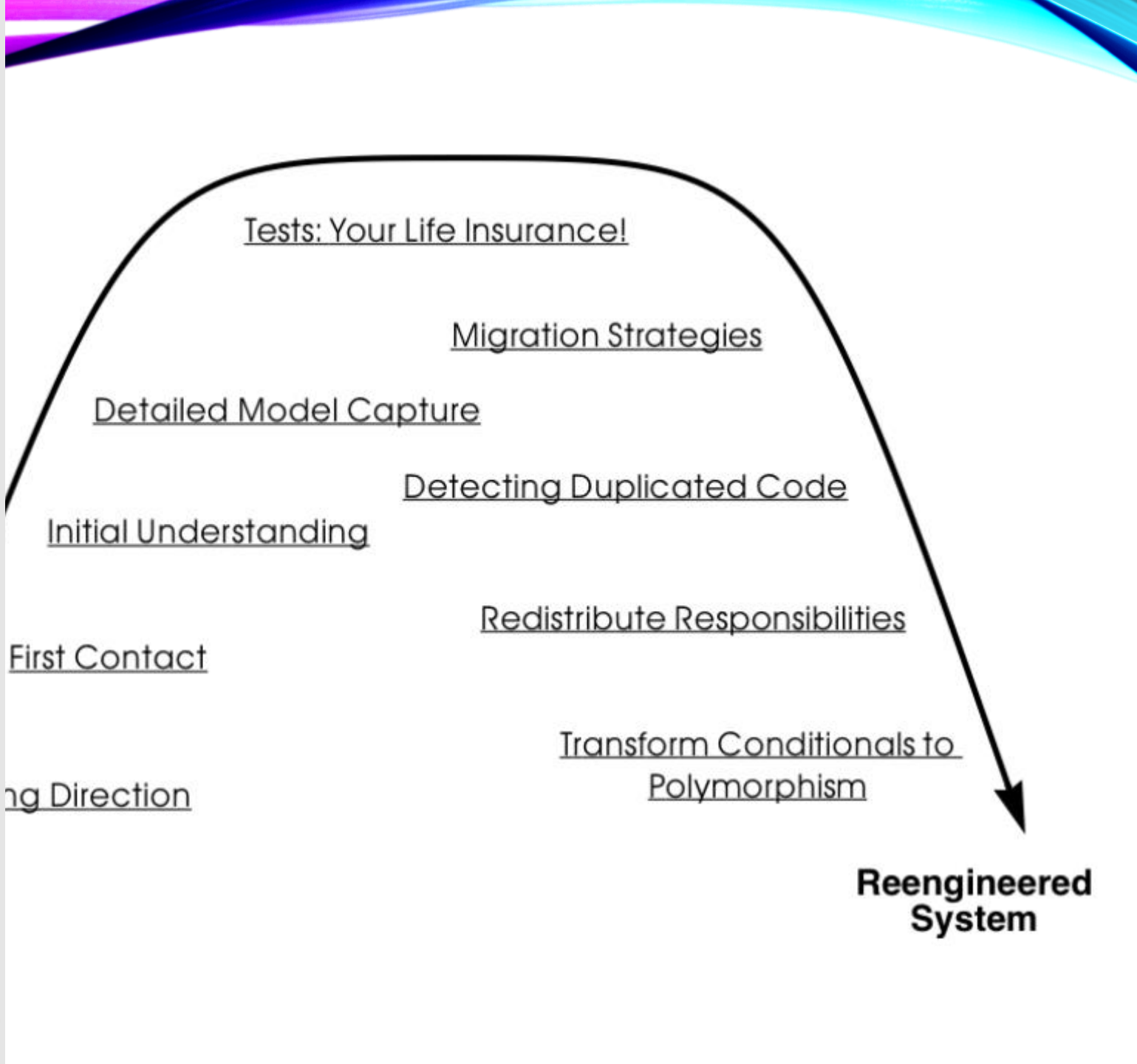
Principles & guidelines for software project management are *especially* relevant for reengineering projects



MOST VALUABLE FIRST

- Problem: Which problems should you address first?

```
while (true) {
    Expression res;
    int c = StreamTokenizer.TT_EOL;
    String varName = null;

    System.out.println("Enter an expression...");
    try {
        while (true) {
            c = st.nextToken();
            if (c == StreamTokenizer.TT_EOF) {
                System.exit(1);
            } else if (c == StreamTokenizer.TT_EOL) {
                continue;
            } else if (c == StreamTokenizer.TT_WORD) {
                if (st.sval.compareTo("dump") == 0) {
                    dumpVariables(variables);
                    continue;
                } else if (st.sval.compareTo("clear") == 0) {
                    variables = new Hashtable();
                    continue;
                } else if (st.sval.compareTo("quit") == 0) {
                    System.exit(0);
                } else if (st.sval.compareTo("exit") == 0) {
                    System.exit(0);
                } else if (st.sval.compareTo("help") == 0) {
                    help();
                    continue;
                }
                varName = st.sval;
                c = st.nextToken();
            }
            break;
        }
        if (c != '=') {
            throw new SyntaxError("missing initial '=' sign.");
        }
```



If It Ain't Broke, Don't Fix It

The name is usually an action phrase.

Intent: Save your reengineering effort for the parts of the system that will make a difference.

The intent should capture the essence of the pattern

Problem

Which parts of a legacy system should you reengineer?

The problem is phrased as a simple question. Sometimes the context is explicitly described.

This problem is difficult because:

- Legacy software systems can be large and complex.
- Rewriting everything is expensive and risky.

Yet, solving this problem is feasible because:

- Reengineering is always driven by some concrete goals.

Next we discuss the forces! They tell us why the problem is difficult and interesting. We also pinpoint the key to solving the problem.

Solution

Only fix the parts that are "broken" — that can no longer be adapted to planned changes.

The solution sometimes includes a recipe of steps to apply the pattern.

Tradeoffs

Pros You don't waste your time fixing things that are not only your critical path.

Each pattern entails some positive and negative tradeoffs.

Cons Delaying repairs that do not seem critical may cost you more in the long run.

Difficulties It can be hard to determine what is "broken".

There may follow a realistic example of applying the pattern.

Rationale

There may well be parts of the legacy system that are ugly, but work well and do not pose any significant maintenance effort. If these components can be isolated and wrapped, it may never be necessary to replace them.

We explain why the solution makes sense.

Known Uses

Alan M. Davis discusses this in his book, *201 Principles of Software Development* .

We list some well documented instances of the pattern.

Related Patterns

Be sure to Fix Problems, Not Symptoms.

Related patterns may suggest alternative actions. Other patterns may suggest logical followup action.

What Next

Consider starting with the Most Valuable First.

EVOLUTION VS RE-ENGG

- Reengineering implies a single cycle of taking an existing system and generating from it a new system
- Reengineering is done to transform an existing “lesser or simpler” system into a new “better” system

JACOBSON AND LINDSTORM EQUATION

Reengineering = Reverse engineering +Δ+ Forward engineering

JACOBSON EQUATION

- Reverse engineering is the activity of defining a more abstract, and easier to understand, representation of the system
- The core of reverse engineering is the **process of examination**, **not a process of change**, therefore it does not involve changing the software under examination.

- The third element "forward engineering," is the traditional process of moving from **high-level abstraction and logical**, implementation-independent designs to the physical implementation of the system.

- The second element  captures alteration that is change of the system.

BENEDUSI REPEATABLE PARADIGM

- **Goals.** In this phase, one analyzes the motivations for setting up the process to identify the **information needs** of the process and the **abstractions to be produced.**

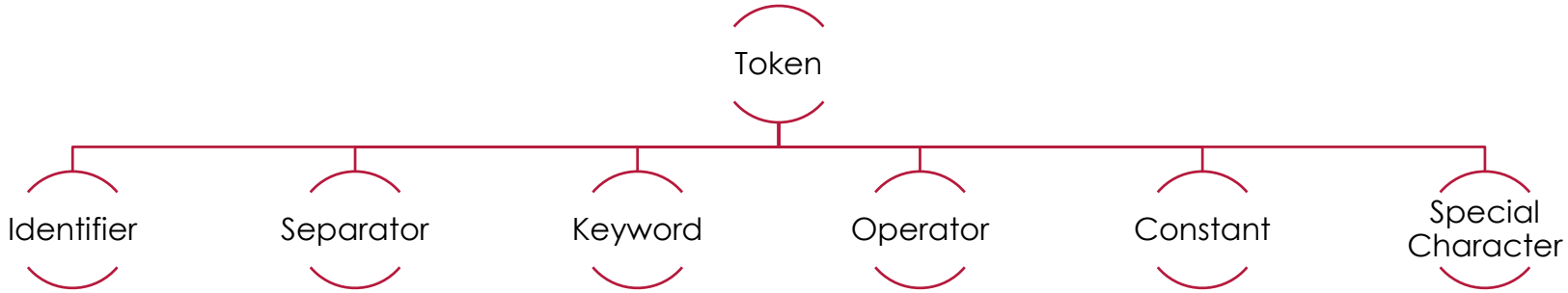
- **Models.** In this phase, one analyzes the abstractions to construct **representation models** that capture the information needed for their production.

- **Tools.** In this phase, software tools are defined, acquired, enhanced, integrated, or constructed to: (i) execute the Models phase and (ii) transform the program models into the abstractions identified in the Goals phase.

- In order to extract information that is not explicitly available in source code, automated analysis techniques,
- such as lexical analysis,
- syntactic analysis,
- control flow analysis,
- data flow analysis, and
- program slicing are used to facilitate reverse engineering.

LEXICAL ANALYSIS

- *Process of converting a sequence of characters in a source code file into a sequence of tokens*
- *processed by a compiler or interpreter*
- *followed by syntax analysis and semantic analysis.*



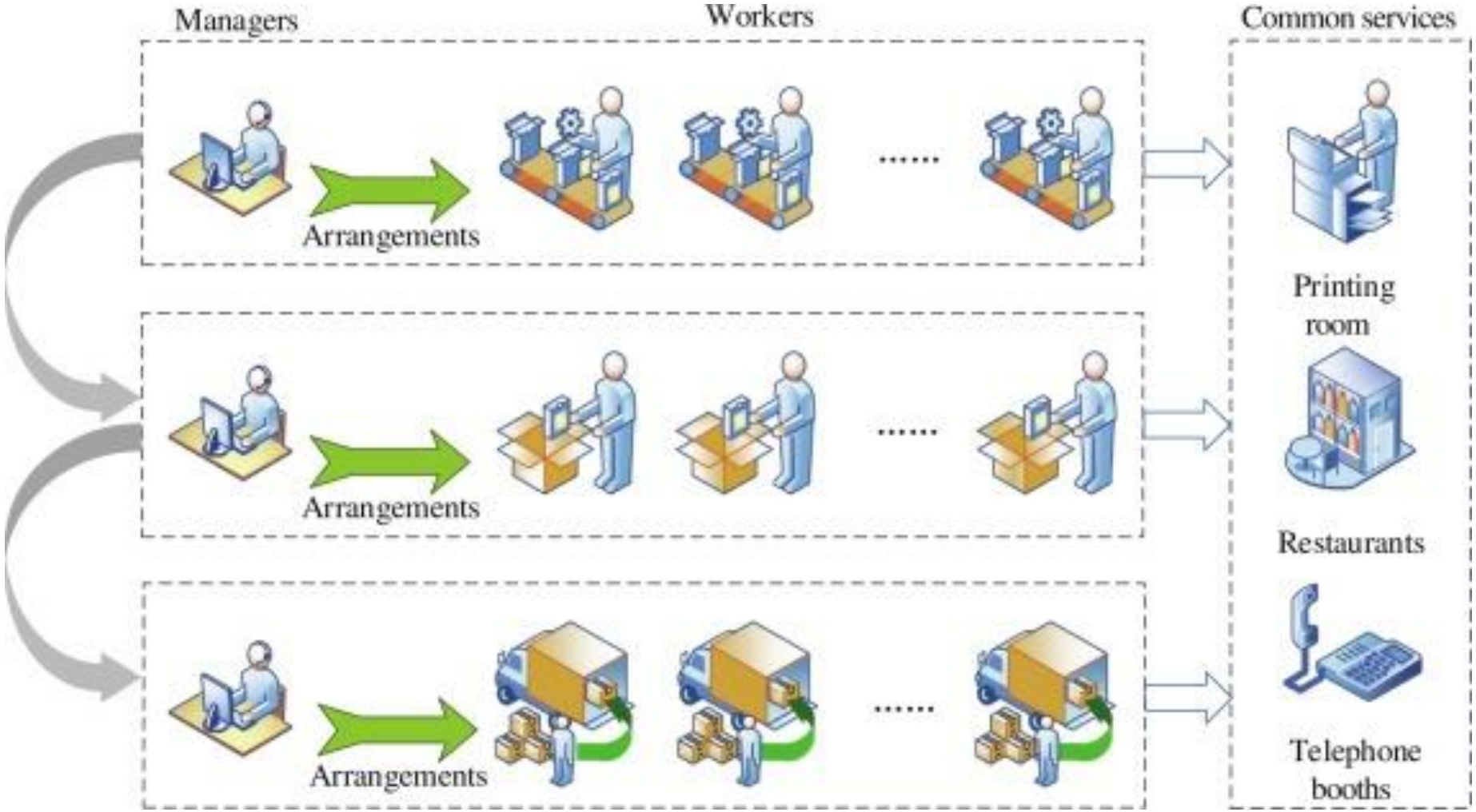
```
while (true) {
    Expression res;
    int c = StreamTokenizer.TT_EOL;
    String varName = null;

    System.out.println("Enter an expression...");
    try {
        while (true) {
            c = st.nextToken();
            if (c == StreamTokenizer.TT_EOF) {
                System.exit(1);
            } else if (c == StreamTokenizer.TT_EOL) {
                continue;
            } else if (c == StreamTokenizer.TT_WORD) {
                if (st.sval.compareTo("dump") == 0) {
                    dumpVariables(variables);
                    continue;
                } else if (st.sval.compareTo("clear") == 0) {
                    variables = new Hashtable();
                    continue;
                } else if (st.sval.compareTo("quit") == 0) {
                    System.exit(0);
                } else if (st.sval.compareTo("exit") == 0) {
                    System.exit(0);
                } else if (st.sval.compareTo("help") == 0) {
                    help();
                    continue;
                }
                varName = st.sval;
                c = st.nextToken();
            }
            break;
        }
        if (c != '=') {
            throw new SyntaxError("missing initial '=' sign.");
        }
```

PROGRAM SLICING

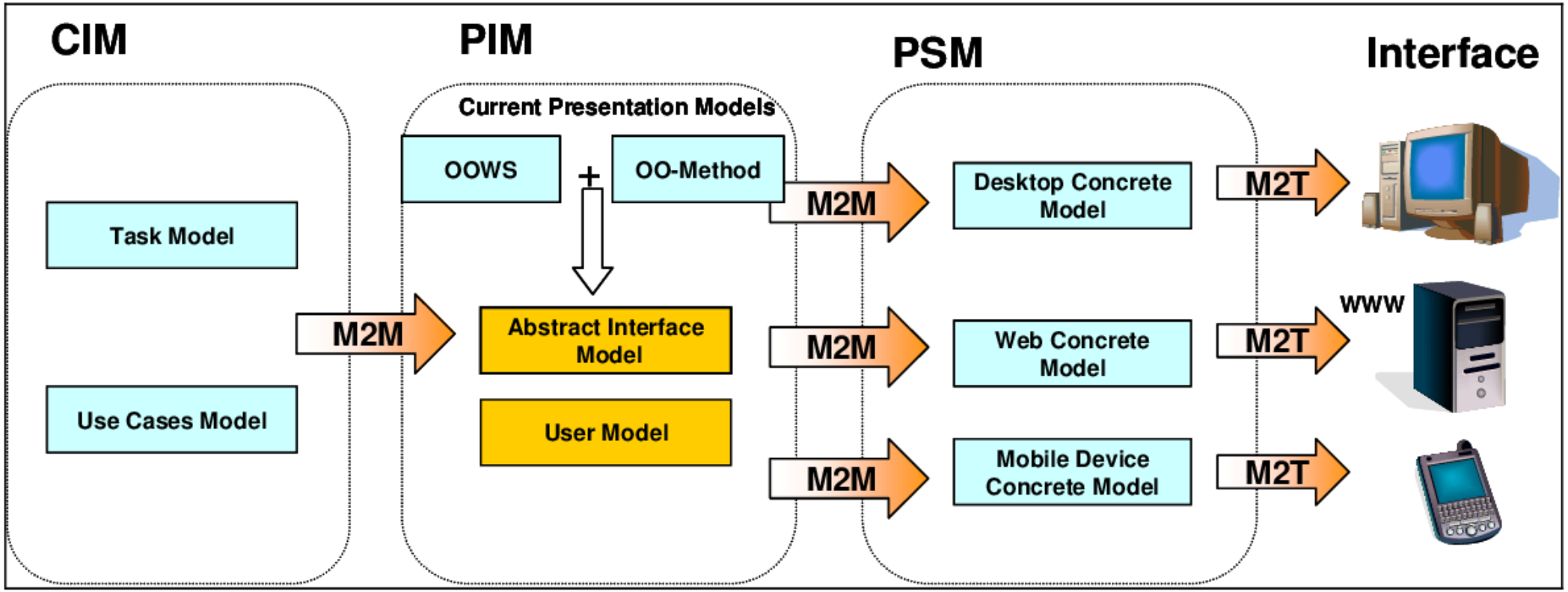
- Take a piece of code and analysis the behavior

```
int z = 10;
int n;
cin >> n;
int sum = 0;
if (n > 10)
    sum = sum + n;
else
    sum = sum - n;
cout << "Hey";
```

ABSTRACT MODEL



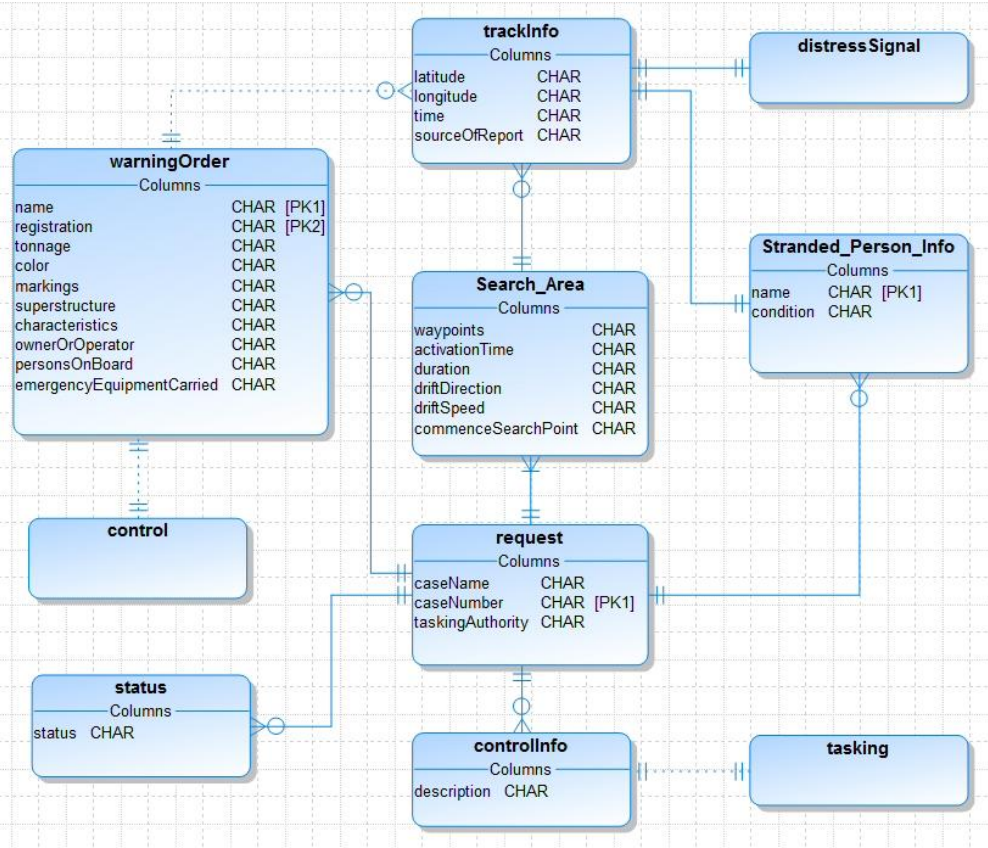
LOGICAL MODEL

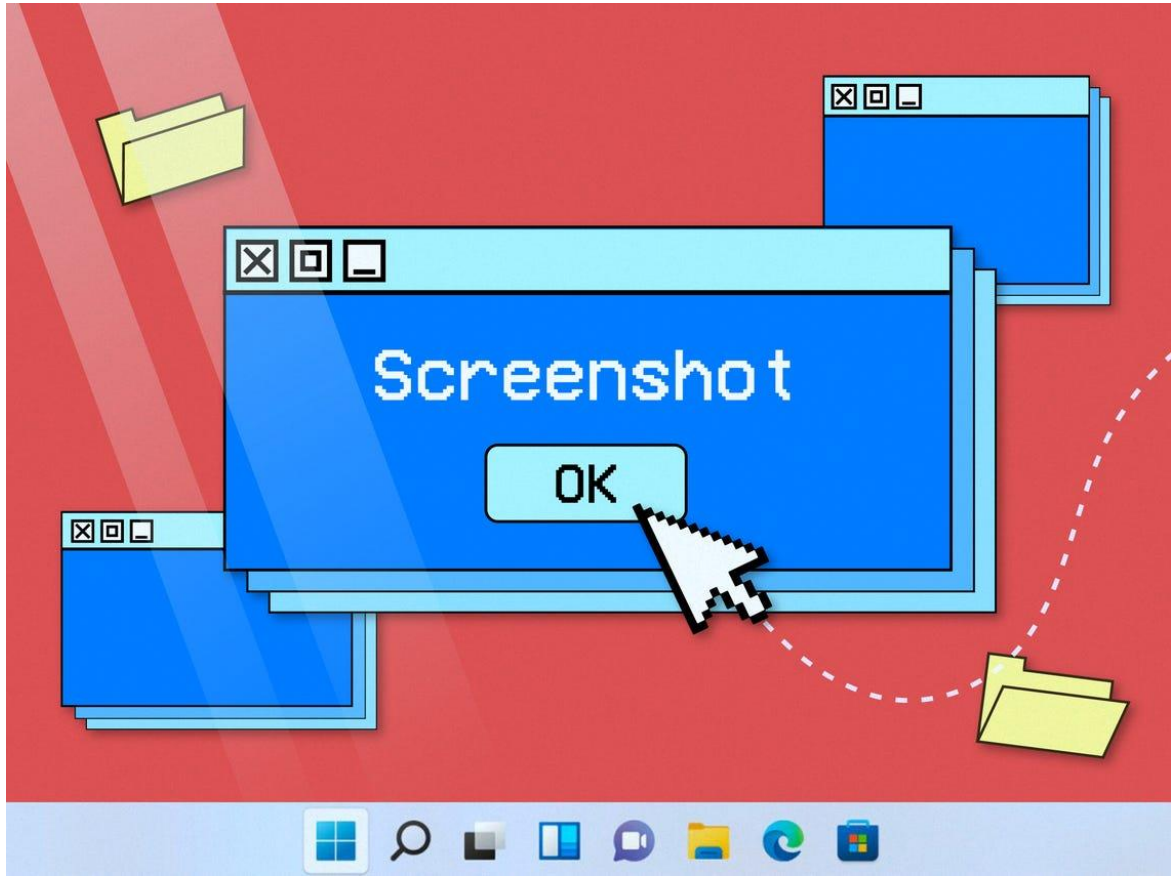


LOGICAL MODEL

- Class Diagrams
- Usecase
- Sequence
- Activity etc.

PHYSICAL MODEL





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.

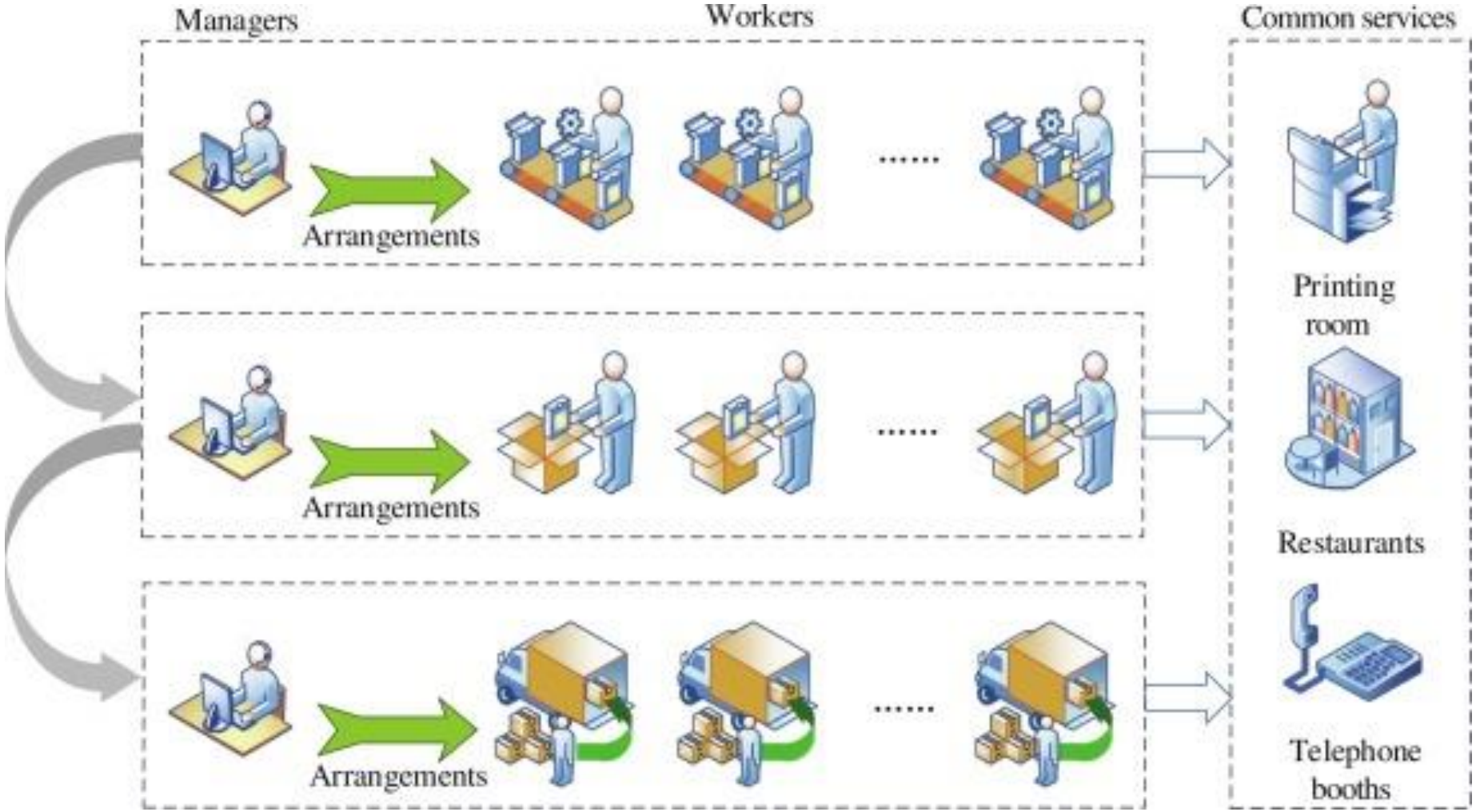
PROGRAM COMPREHENSION

- 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,

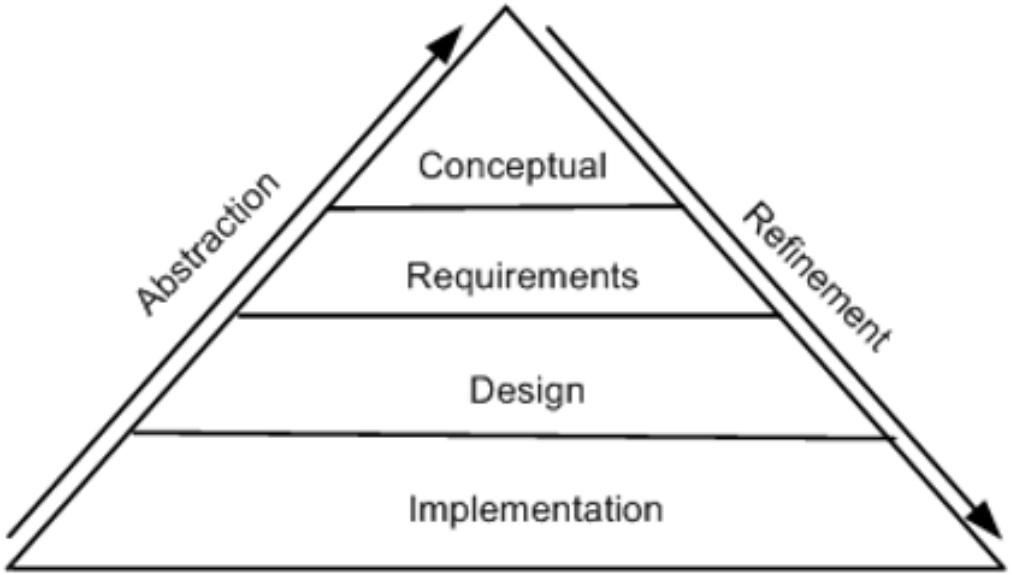
  (ii) represent the system at a higher level of abstraction or in another form.

- **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?



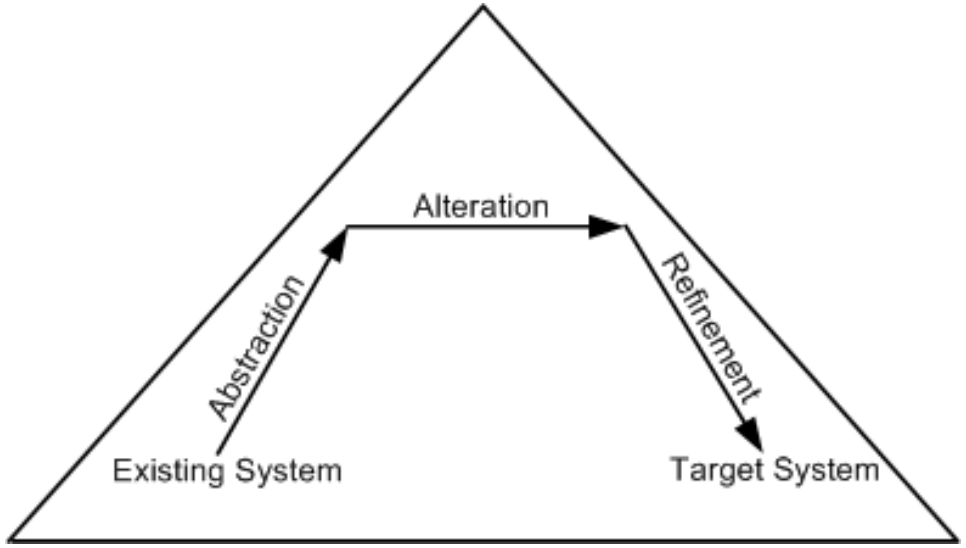
levels of abstraction and refinement

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



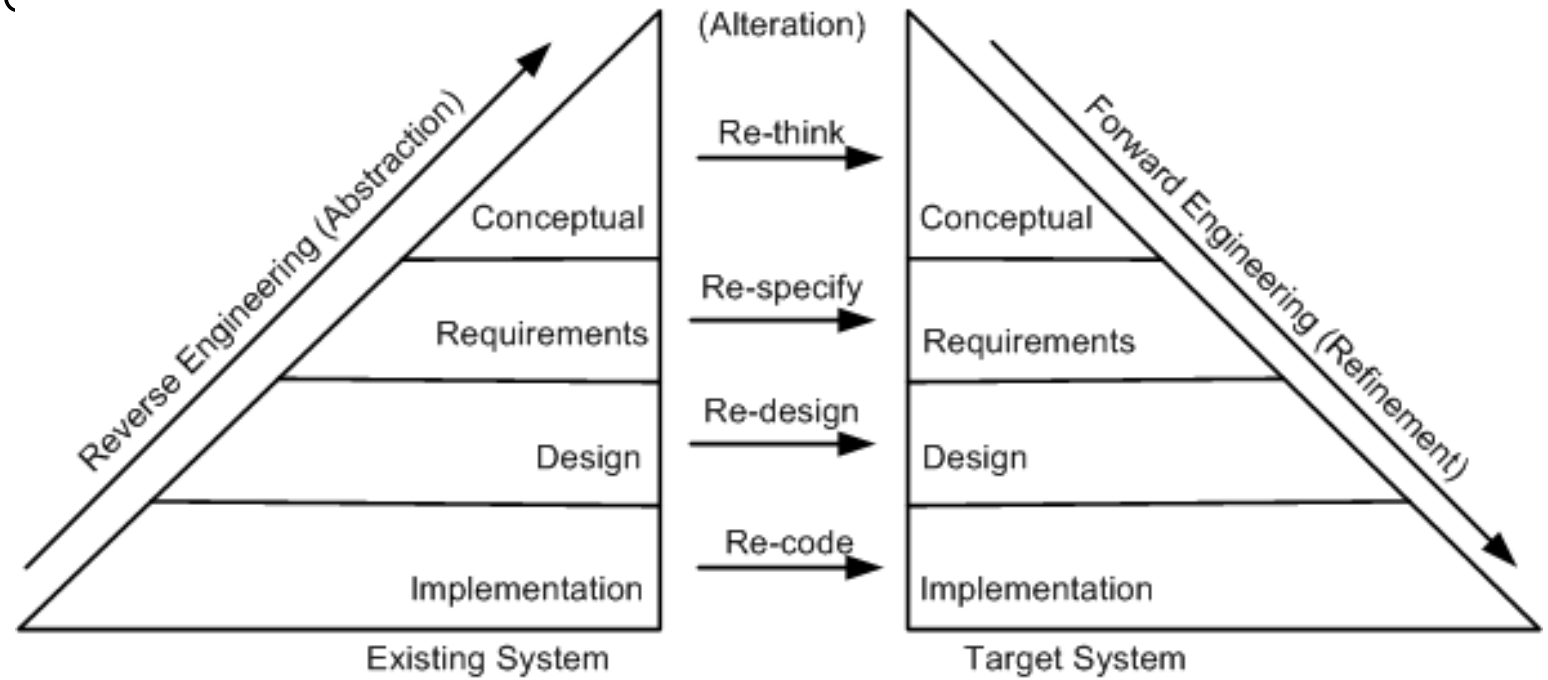
A GENERAL MODEL FOR SOFTWARE REENGINEERING

- 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

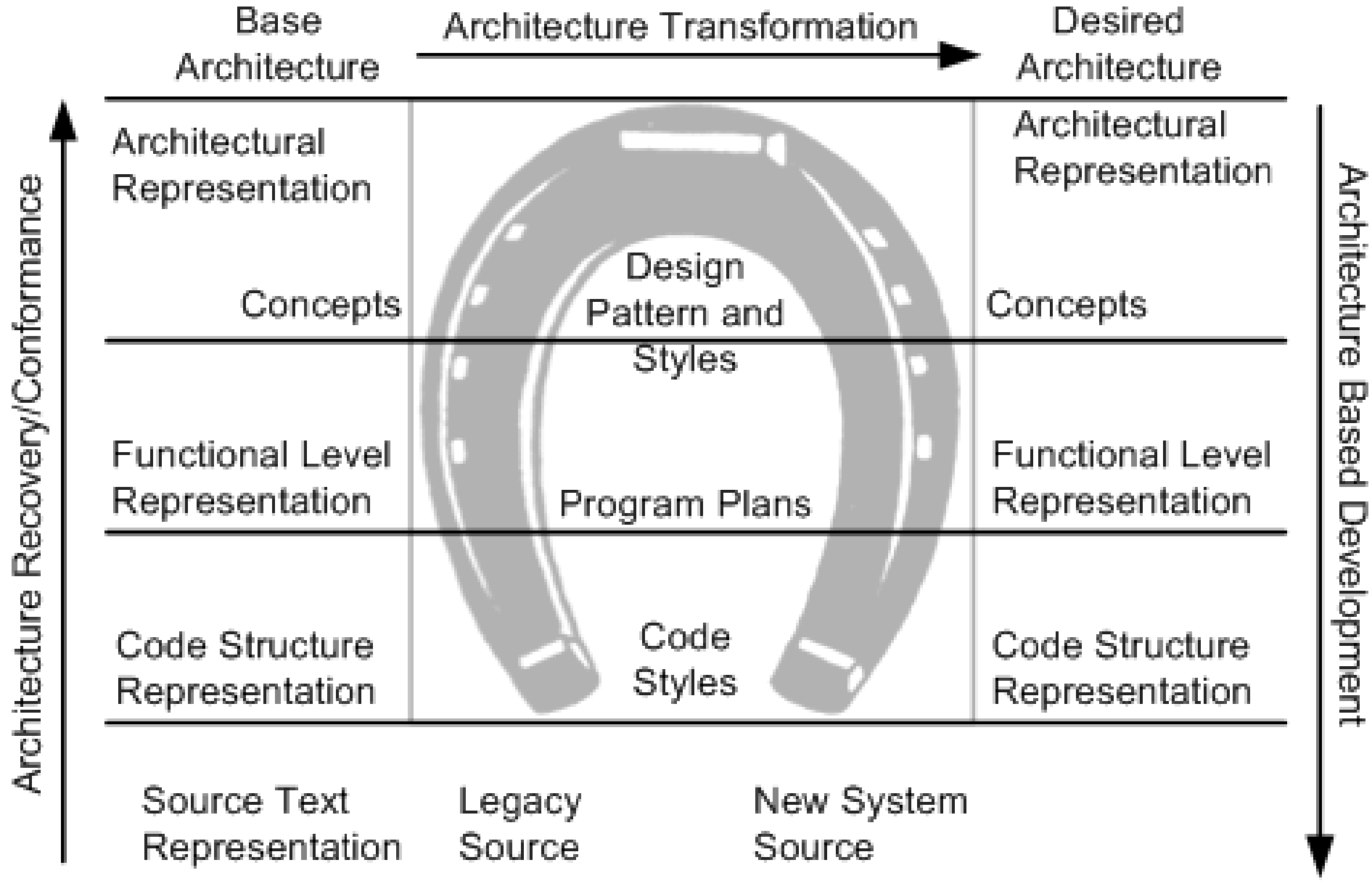
- The model in Figure, proposed by Eric J. Byrne suggests that reengineering is a sequence of three activities:
  - **reverse engineering, re-design, and forward engineering**
  - strongly founded in three principles, namely, abstraction, alteration, and refinement



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
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
Architecture Recovery/Conformance
Architecture Based Development

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.

SOFTWARE RE-ENGINEERING

Unit 4

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.
- 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.
- Internal schema, business logic, flow, dependencies, internal structures are not changed.
- May have some cosmetic changes/additions to improve look and feel of the system.

(a) Rewrite

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 new flow
- Remove expensive codes (complex)
- Remove any extra iterative and conditional flows
- Make function calls effective.

REWORK STRATEGY:

- A classical, rework strategy based approach is as follows:

Step 1

- Application of abstraction: By parsing the code, generate a control-flow graph (CFG) for the given system.

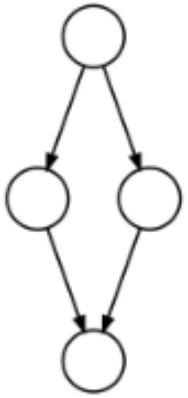
Step 2

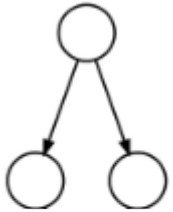
- Application of alteration: Apply a restructuring algorithm to the control-flow graph to produce a structured control-flow graph.

Step 3

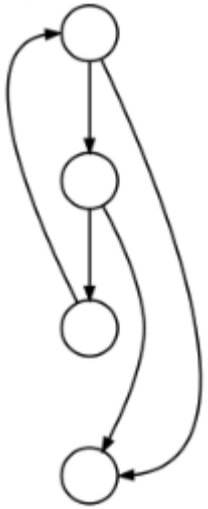
- Application of refinement: Translate the new, structured control-flow graph back into the original programming language.

if-then-else

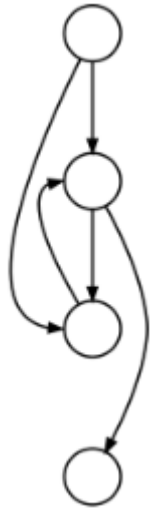




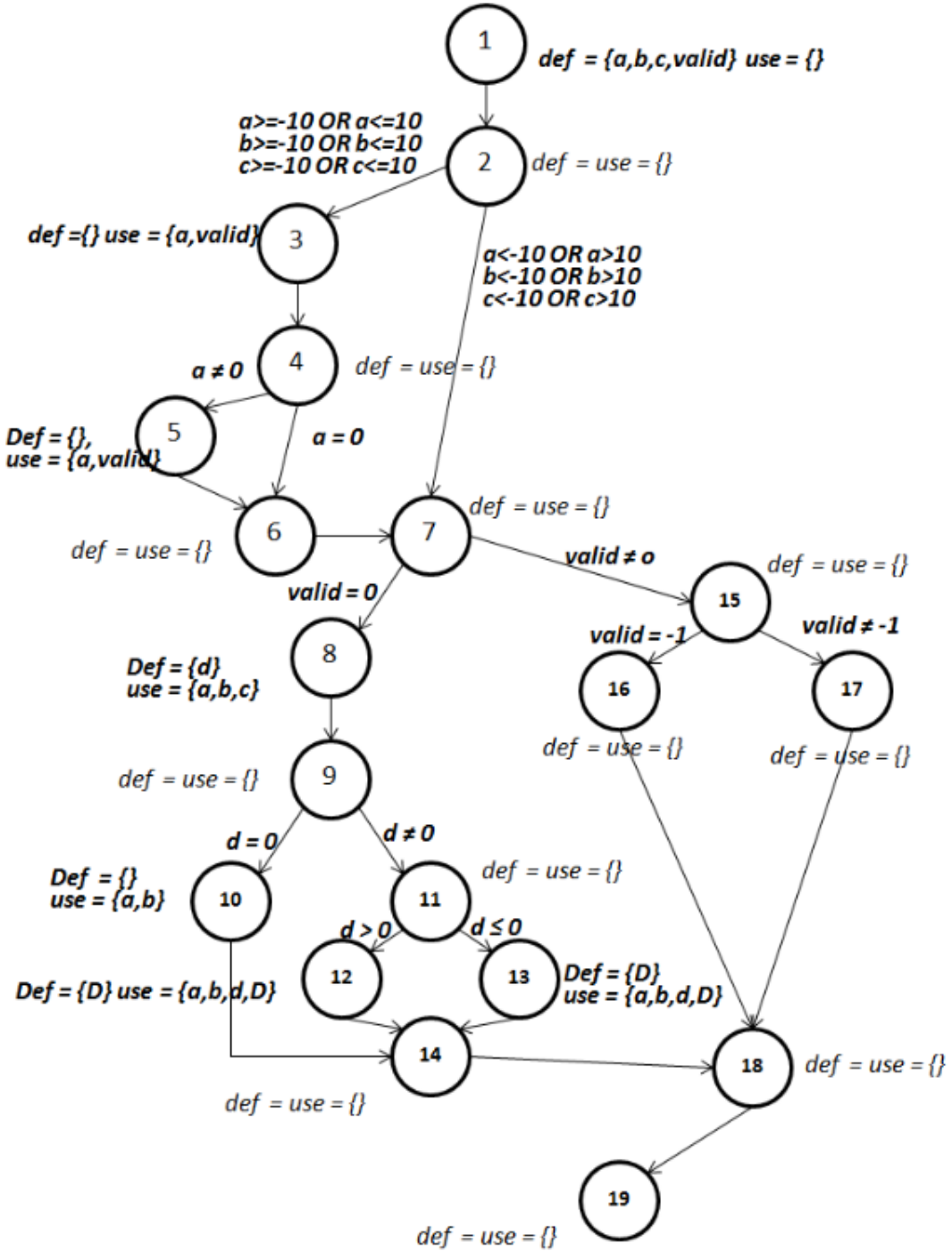
a loop

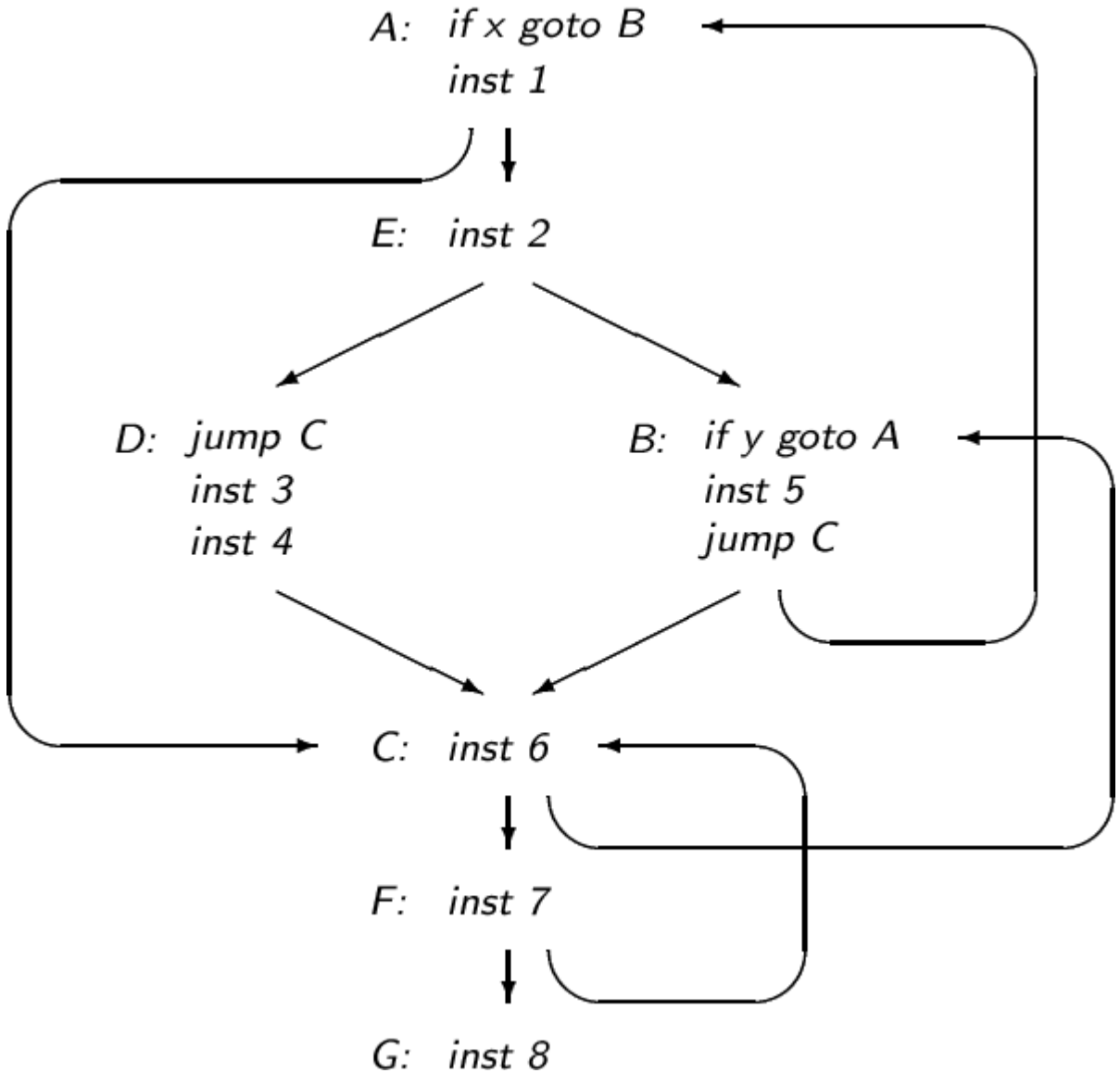


a natural loop with two exits, e.g. while with an if...break in the middle; non-structured but reducible



a loop with two entry points, e.g. goto into a while or for loop


1
def = {a,b,c,valid} use = {}
a>=-10 OR a<=10
b>=-10 OR b<=10
c>=-10 OR c<=10
2
def = use = {}
def ={} use = {a,valid}
3
a<-10 OR a>10
b<-10 OR b>10
c<-10 OR c>10
4
def = use = {}
a ≠ 0
a = 0
5
Def = {},
use = {a,valid}
6
def = use = {}
7
def = use = {}
valid ≠ o
15
def = use = {}
valid = 0
8
Def = {d}
use = {a,b,c}
valid = -1
valid ≠ -1
16
17
def = use = {}
def = use = {}
9
def = use = {}
d = 0
d ≠ 0
Def = {}
use = {a,b}
10
11
def = use = {}
d > 0
d ≤ 0
12
13
Def = {D} use = {a,b,d,D}
Def = {D}
use = {a,b,d,D}
14
18
def = use = {}
def = use = {}
19
def = use = {}


A: if x goto B
inst 1
E: inst 2
D: jump C
inst 3
inst 4
B: if y goto A
inst 5
jump C
C: inst 6
F: inst 7
G: inst 8

CFG

- A **control-flow graph** (**CFG**) is a representation, using graph notation, of all paths that might be traversed through a program during its execution.
- The CFG is use as a static-analysis tools.

- he CFG can be obtained, by starting from the program's (full) flow graph.
- The graph in which every node represents an individual instruction.

- Example :

```
0: t0 = read_num
1: if t0 mod 2 == 0
2: print t0 + " is even."
3: goto 5
4: print t0 + " is odd."
5: end program
```

```
int main(){
int t0;
Scanf("%d", & t0);
if (t0 % 2 == 0)
    printf("%d is even.", t0);
Printf(" %d is odd.", t0);
}
```

Starting Abstraction Level	Type Change	Reengineering Strategy		
		Rewrite	Rework	Replace
Implementation Level	Re-code	Yes	Yes	Yes
	Re-design	Bad	Yes	Yes
	Re-specify	Bad	Yes	Yes
	Re-think	Bad	Yes*	Yes*
Design Level	Re-code	No	No	No
	Re-design	Yes	Yes	Yes
	Re-specify	Bad	Yes	Yes
	Re-think	Bad	Yes*	Yes*
Requirement Level	Re-code	No	No	No
	Re-design	No	No	No
	Re-specify	Yes	Yes	Yes
	Re-think	Bad	Yes*	Yes*
Conceptual Level	Re-code	No	No	No
	Re-design	No	No	No
	Re-specify	No	No	No
	Re-think	Yes	Yes*	Yes*

Table 4.1: Reengineering process variations [5] (©[1992] IEEE).

Yes - One can produce a target system.

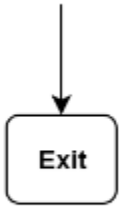
Yes* - Same as Yes, but the starting degree of abstraction is lower than the uppermost degree of abstraction within the conceptual abstraction level.

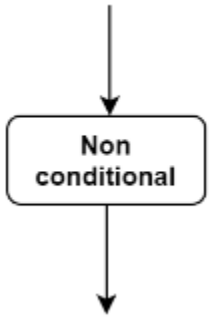
No - One cannot start at abstraction level A, make B type of changes by using strategy C, because the starting abstraction level is higher than the abstraction level required by the particular type of change.

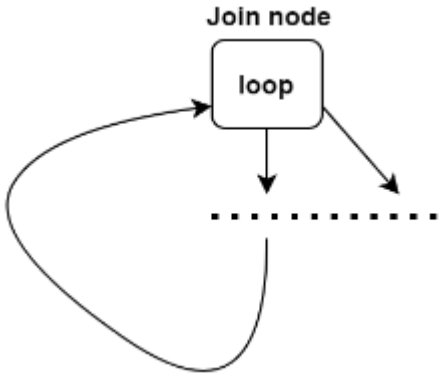
Bad - A target system can be created, but the likelihood of achieving a good result is low.

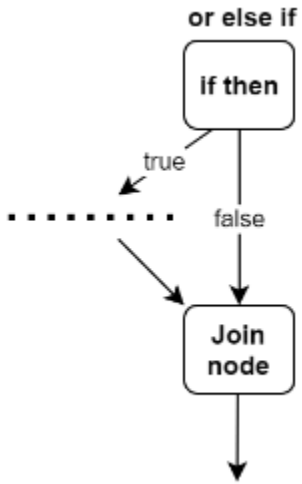
CHEAT SHEET

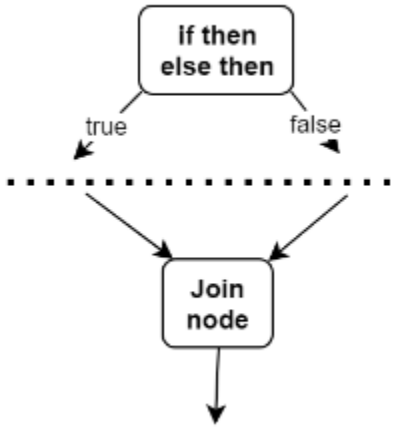

Entry

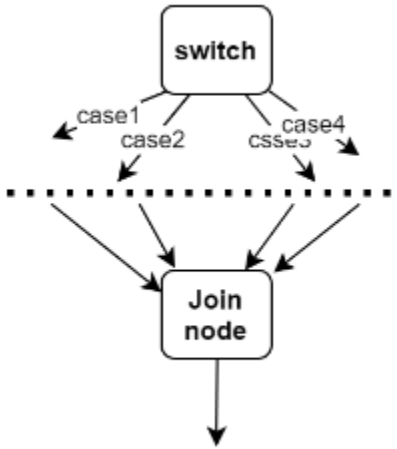

Exit


Non
conditional


Join node
loop


or else if
if then
true
false
Join
node


If then
else then
true
false
Join
node


switch
case1
case2
case4
Join
node

REENGINEERING PROCESS

- An ordered set of activities designed to perform a specific task is called a process.
- For ease of understanding and communication, processes are described by means of process models.
- For example, in the software development domain, the Waterfall process model is widely used in developing well-understood software systems.
- Process models are used to comprehend, evaluate, reason about, and improve processes.

REENGINEERING PROCESS

- Process models are described by means of important relationships among data objects, human roles, activities, and tools.
- We will discuss five process models for software reengineering.
- The five approaches are different in two aspects:

  (i) the extent of reengineering performed, and

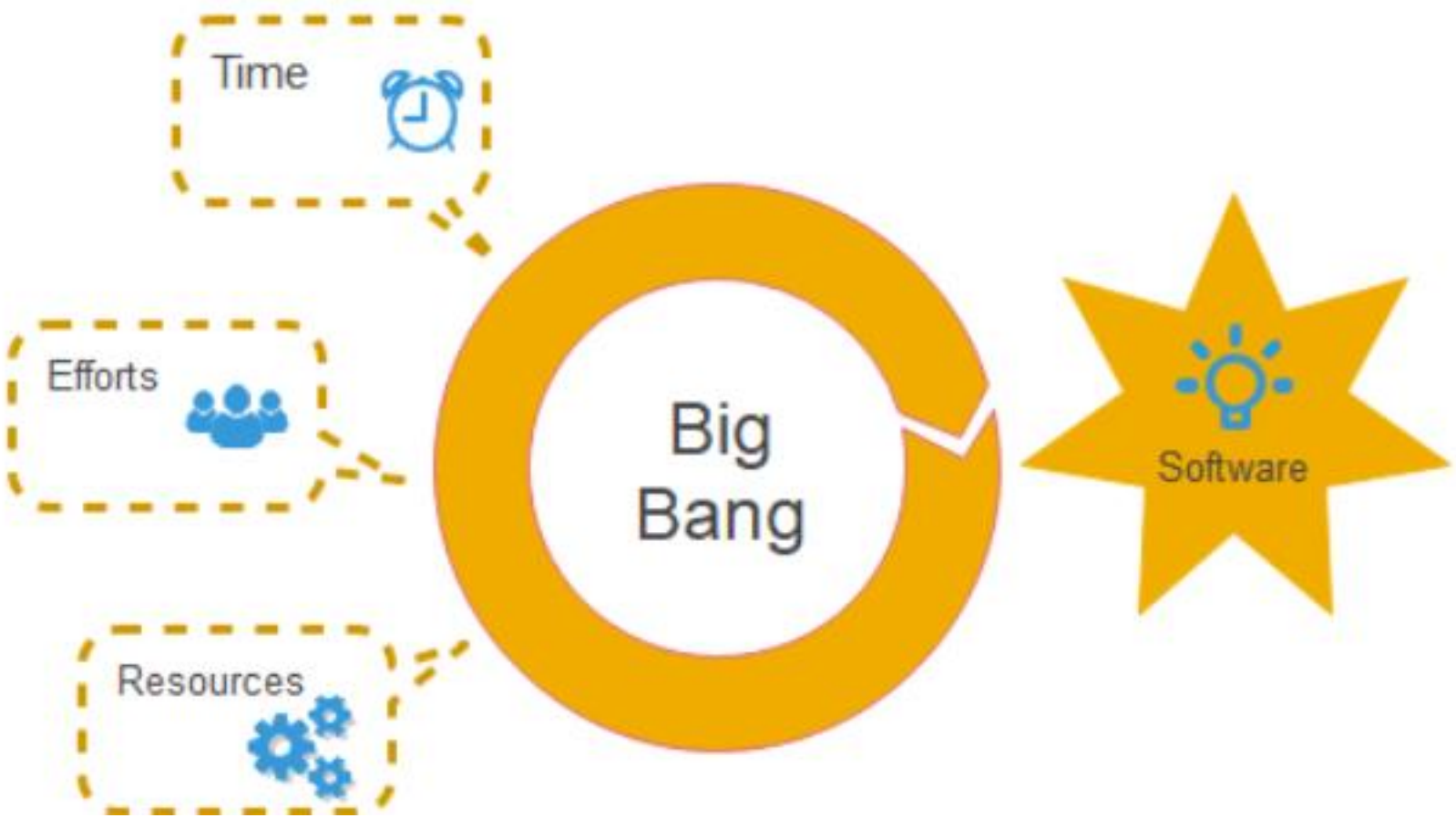
  (ii) the rate of substitution of the operational system with the new one.

BIG BANG APPROACH

- The "**Big Bang**" approach replaces the whole system at once.
- Once a reengineering effort is initiated, it is continued until all the objectives of the project are achieved and the target system is constructed.
- This approach is generally used if reengineering cannot be done in parts.
- For example, if there is a need to move to a different system architecture, then all components affected by such a move must be changed at once.

DISADVANTAGE

- The consequent advantage is that the system is brought into its new environment all at once.
- The disadvantage of Big Bang is that the reengineering project becomes a monolithic task, which may not be desirable in all situations.
- In addition, the Big Bang approach consumes too much resources at once for large systems, and takes a long stretch of time before the new system is visible.


Time
Efforts
Resources
Big
Bang
Software

INCREMENTAL APPROACH

- In this approach a system is reengineered gradually, one step closer to the target system at a time.
- For a large system, several new interim versions are produced and released.
- Successive interim versions satisfy increasingly more project goals than their preceding versions.

- The advantages of this approach are as follows:

  (i) locating errors becomes easier, because one can clearly identify the newly added components, and

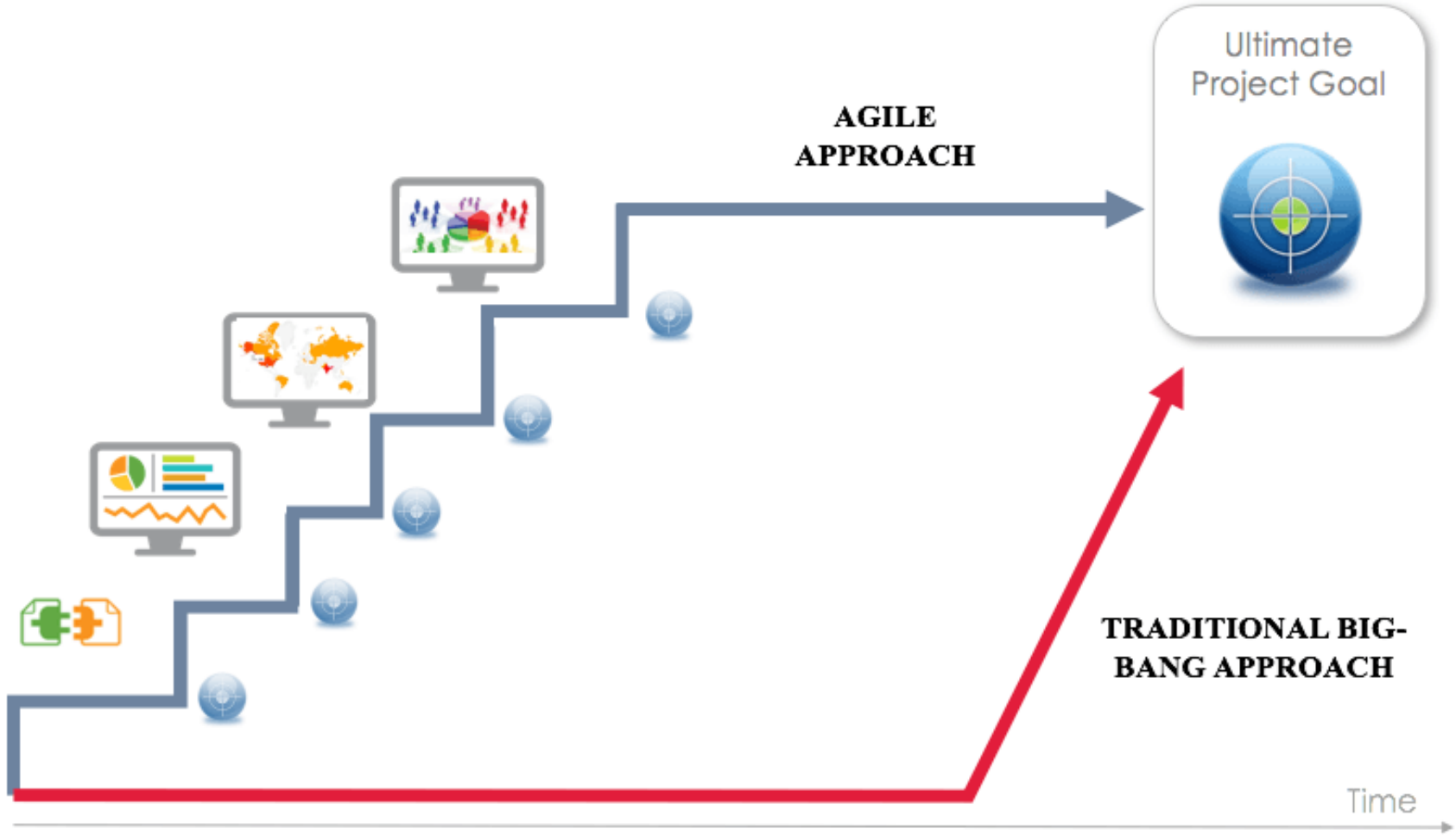
  (ii) It becomes easy for the customer to notice progress, because interim versions are released.

- The disadvantages of the incremental approach are as follows:

  (i) with multiple interim versions and their careful version controls, the incremental approach takes much longer to complete, and

  (ii) even if there is a need, the entire architecture of the system cannot be changed.

INCREMENTAL VS BIG BANG APPROACH



PARTIAL APPROACH

- In this approach, only a part of the system is reengineered and then it is integrated with the non-engineered portion of the system.
- One must decide whether to use a "Big Bang" approach or an "Incremental" approach for the portion to be reengineered.
- The following three steps are followed in the partial approach:
  - In the first step, the existing system is partitioned into two parts: one part is identified to be reengineered and the remaining part to be not reengineered.
  - In the second step, reengineering work is performed using either the "Big Bang" or the "Incremental" approach.
  - In the third step, the two parts, namely, the not-to-be-reengineered part and the reengineered part of the system, are integrated to make up the new system.

- The partial approach has the advantage of reducing the scope of reengineering that is less time and costs less.

- A disadvantage of the partial approach is that modifications are not performed to the interface between the portion modified and the portion not modified.
- Compatibility or performance issue may be faced.

ITERATIVE APPROACH

- The reengineering process is applied on the source code of a few procedures at a time, with each reengineering operation lasting for a short time.
- This process is repeatedly executed on different components in different stages.
- During the execution of the process, ensure that the four types of components can coexist:
  - old components not reengineered,
  - components currently being reengineered,
  - components already reengineered, and
  - new components added to the system.

- There are two advantages of the iterative reengineering process:
  (i) it guarantees the continued operation of the system during the execution of the reengineering process, and
  (ii) the maintainers' and the users' familiarities with the system are preserved.

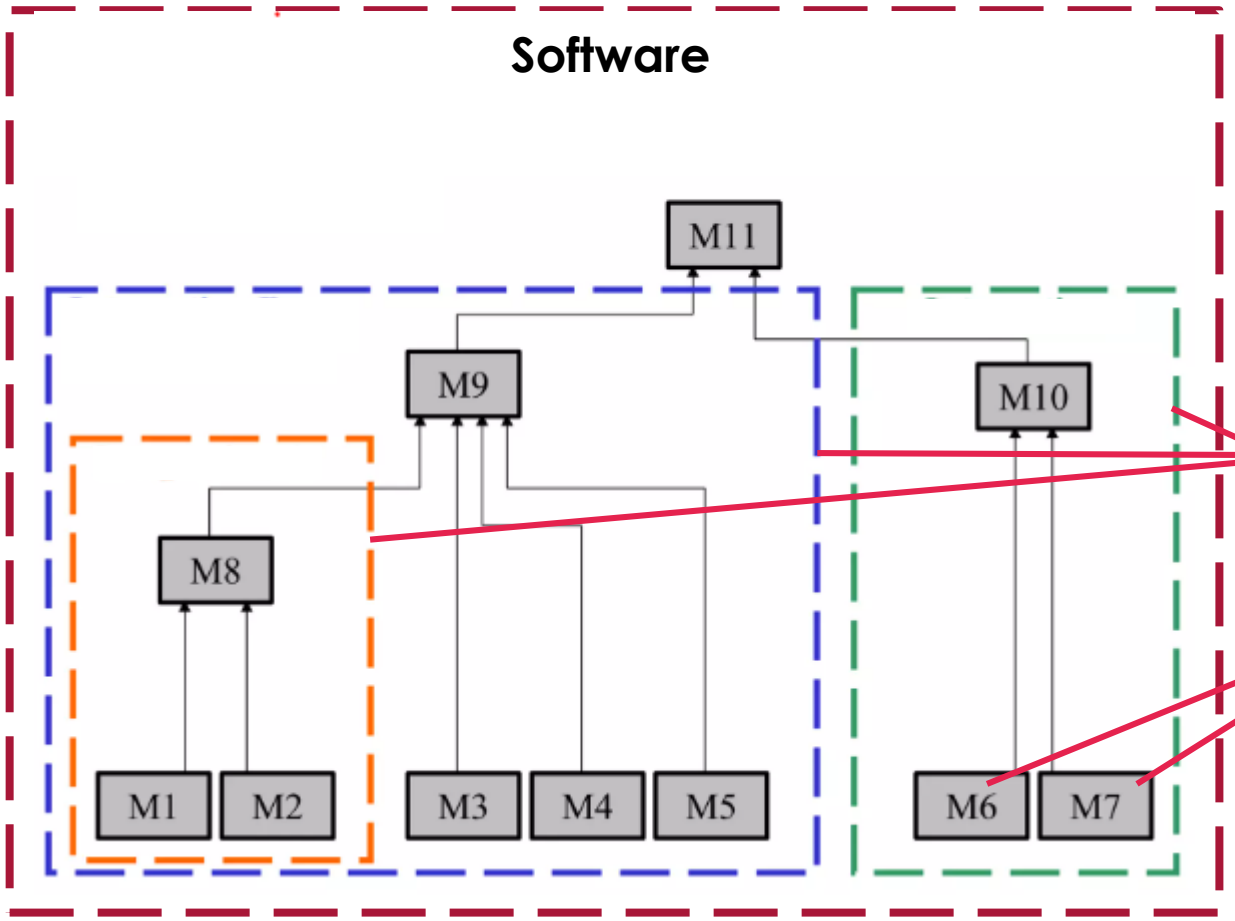
- The disadvantage of this approach is the need to keep track of the four types of components during the reengineering process.
- In addition, both the old and the newly reengineered components need to be maintained.

EVOLUTIONARY APPROACH

- In the "Evolutionary approach" components of the original system are substituted with re-engineered components.
- In this approach, the existing components are grouped by functions and reengineered into new components.
- Software engineers focus their reengineering efforts on identifying functional objects irrespective of the locations of those components within the current system.
- As a result, the new system is built with functionally cohesive components as needed.

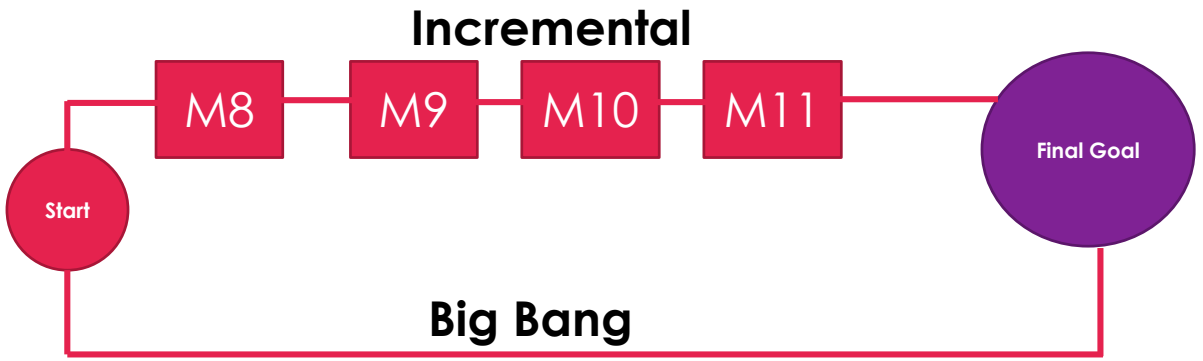
- There are two advantages of the “Evolutionary” approach:
  (i) the resulting design is more cohesive, and
  (ii) the scope of individual components is reduced.

- A major disadvantage:
  (i) all the functions with much similarities must be first identified throughout the operational system.
  (ii) next, those functions are refined as one unit in the new system



Partial

Iterative



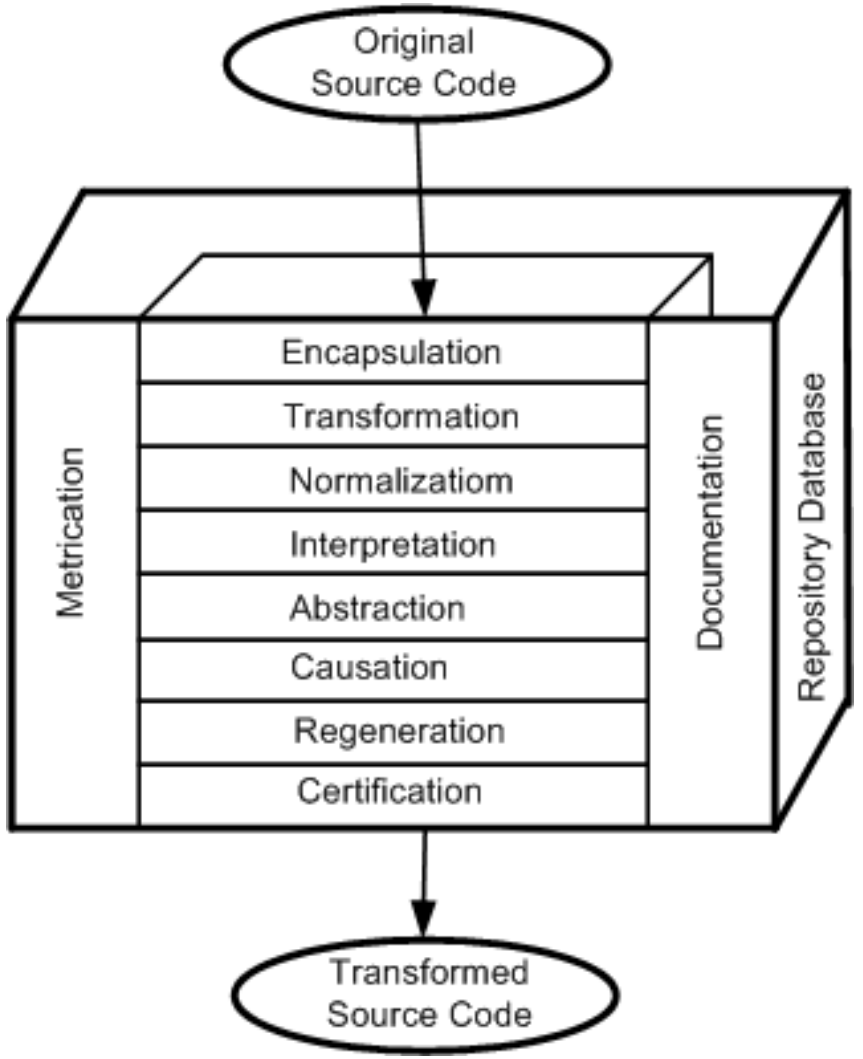
SOURCE CODE REENGINEERING REFERENCE MODEL

- The SCORE/RM model was proposed by Colbrook, Smythe and Darlison.

- The framework consists of four kinds of elements:
  - **function,**
  - **documentation,**
  - **repository database, and**
  - **metrication.**

FUNCTION ELEMENT

- The function element is divided into eight layers, namely:
  - Encapsulation,
  - Transformation,
  - Normalization,
  - Interpretation,
  - Abstraction,
  - Causation,
  - Regeneration, and
  - Certification.



- The eight layers provide a detailed approach to:

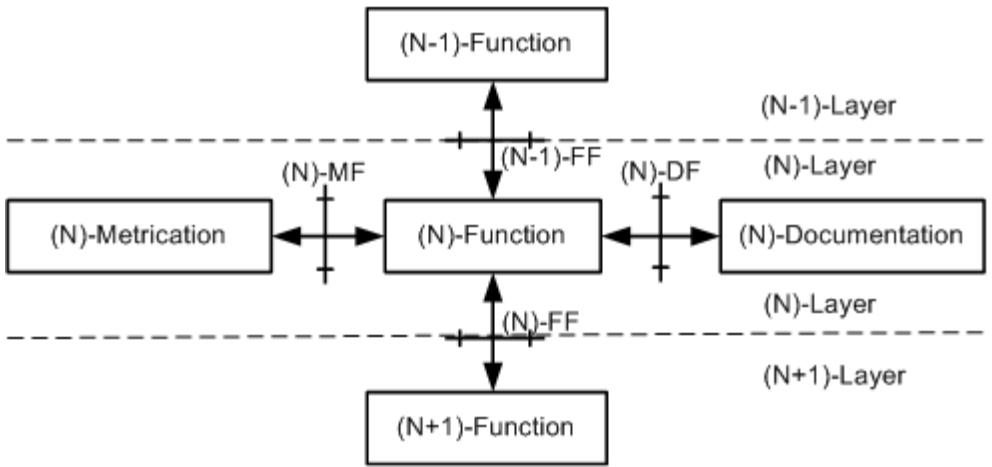
  (i) rationalizing the system to be reengineered by removing redundant data and altering the control flow,

  (ii) comprehending the software's requirements, and

  (iii) reconstructing the software according to established practices.

- The first six of the eight layers together constitute a process for reverse engineering, and the final three a process for forward engineering.
- Improvements in the software as a result of reengineering is quantified by means of the metrication element.
- The metrication element is described in terms of the relevant software metrics before executing a layer and after executing the same layer.
- The repository database is the information store for the entire reengineering process, containing the following kinds of information:
  - metrication,
  - documentation, and
  - both the old and the new source code.

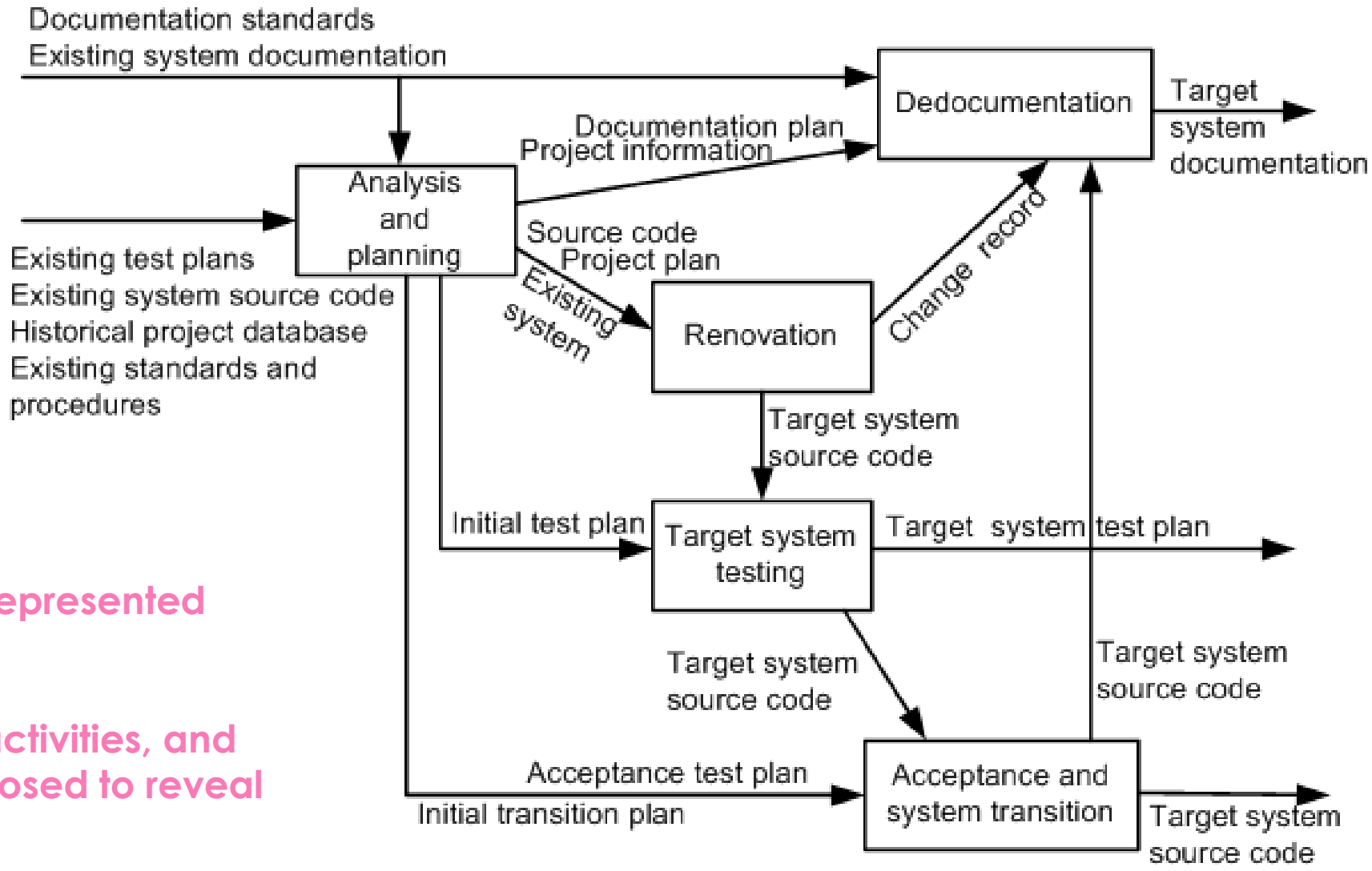
- The interfaces among the elements are shown in Figure.
- For simplicity, any layer is referred to as (N)− layer, while its next lower and next higher layers are referred to as (N − 1)− layer and the (N + 1)− layer, respectively.
- The three types of interfaces are explained as follows:
  - Metrication/Function: (N)-MF – the structures describing the metrics and their values.
  - Documentation/Function: (N)-DF – the structures describing the documentation.
  - Function/Function: (N)-FF – the representation structures for source code passed between the layers.



The interface nomenclature

PHASE REENGINEERING MODEL

- The phase model of software reengineering was originally proposed by Byrne and Gustafson.
- The model comprises five phases: analysis and planning, renovation, target system testing, redocumentation, and acceptance testing and system transition, as depicted in Figure on next slide.
- The labels on the arcs denote the possible information that flows from the tail entities of the arcs to the head entities.



- A major process activity is represented by each phase.

- Tasks represent a phase's activities, and tasks can be further decomposed to reveal the detailed methodologies.

ANALYSIS AND PLANNING

- Analysis addresses three technical and one economic issue.
  - The first technical issue concerns the present state of the system to be reengineered and understanding its properties.
  - The second technical issue concerns the identification of the need for the system to be reengineered.
  - The third technical issue concerns the specification of the characteristics of the new system to be produced.
- The economic issue concerns a cost and benefit analysis of the reengineering project.
- Planning includes:
  - understanding the scope of the work;
  - identifying the required resources;
  - identifying the tasks and milestones;
  - estimating the required effort; and
  - preparing a schedule.

- Task Analysis & Planning Phase

Task	Description
Implementation motivations and objectives	List the motivations for reengineering the system. List the objectives to be achieved.
Analyze environment	Identify the differences between the existing and the target environments. Differences can influence system changes.
Collect inventory	Form a baseline for knowledge about the operational system by locating all program files, documents, test plans, and history of maintenance.
Analyze implementation	Analyze the source code and record the details of the code.
Define approach	Choose an approach to reengineer the system.
Define project procedures and standards	Procedures outline how to perform reviews and report problems. Standards describe the acceptable formats of the outputs of processes.
Identify resources	Determine what resources are going to be used; ensure that resources are ready to be used.
Identify tools	Determine and obtain tools to be used in the reengineering project.
Data conversion planning	Make a plan to effect changes to databases and files.
Test planning	Identify test objectives and test procedures, and evaluate the existing test plan. Design new tests if there is a need.
Define acceptance criteria	By means of negotiations with the customers, identify acceptance criteria for the target system.
Documentation planning	Evaluate the existing documentation. Develop a plan to redocument the target system.
Plan system transition	Develop an end-of-project plan to put the new system into operation and phase out the old one.
Estimation	Estimate the resource requirements of the project: effort, cost, duration, and staffing.
Define organizational structure	Identify personnel for the project, and develop a project organization.
Scheduling	Develop a schedule, including dependencies, for project phases and tasks.

RENOVATION

- An operational system is modified into the target system in the renovation phase.
- Two main aspects of a system are considered in this phase:

  (i) representation of the system.

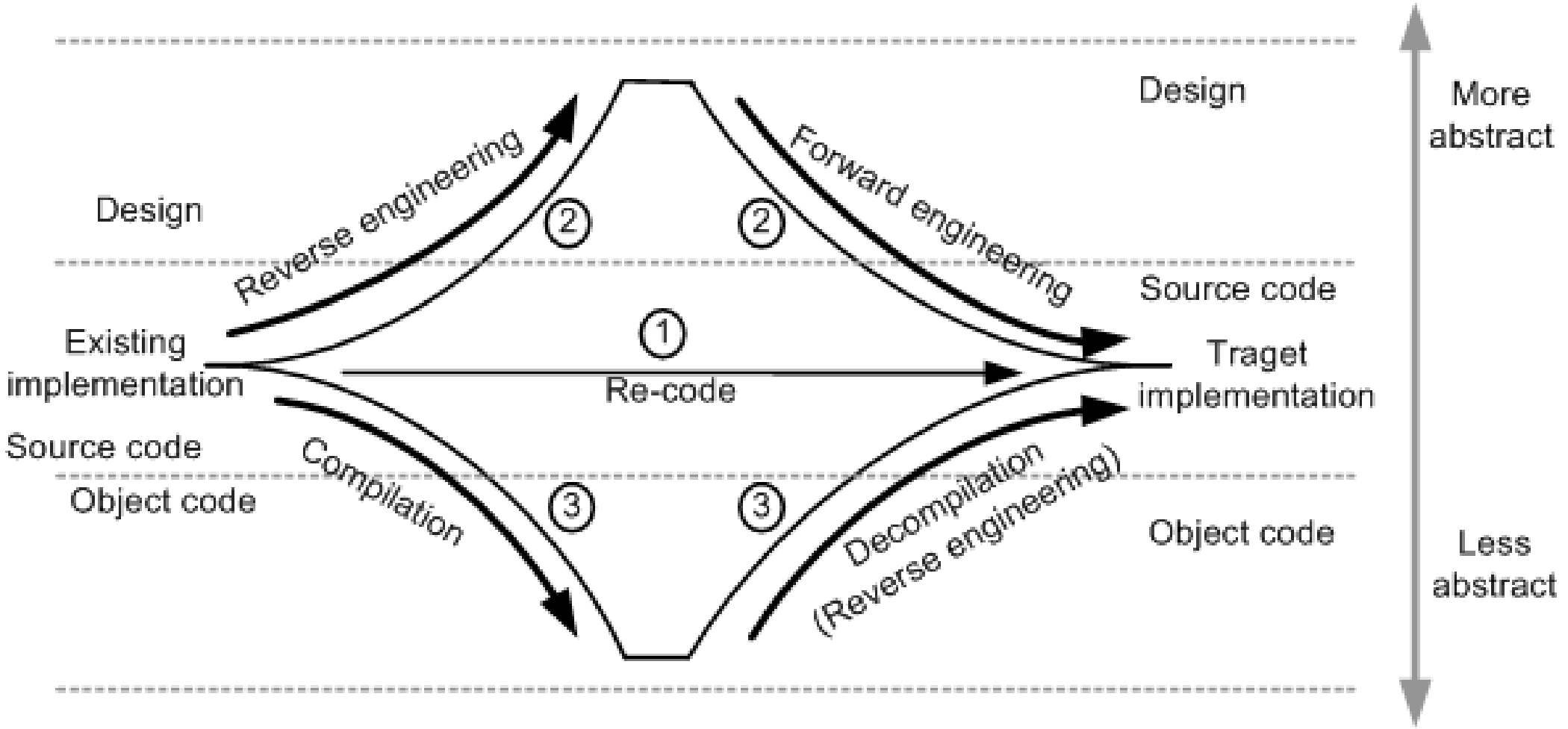
  It refers to source code, but it may include the design model and the requirement specification of the existing system.

  (ii) representation of external data.

  It refers to the database and/or data files used by the system. Often the external data are reengineered, and it is known as data reengineering.
- An operational system can be renovated in many ways, depending upon the objectives of the project, the approach followed, and the starting representation of the system.
- It may be noted that the starting representation can be source code, design, or requirements.
- Table 4.1 discussed earlier recommends several alternatives to renovate a system.

RENOVATION: EXAMPLE

- A project in which the objective is to re-code the system from Fortran to C.
- Figure 4.9 shows the three possible replacement strategies.
- First, to perform source-to-source translation, program migration is used.
- Second, a high-level design is constructed from the operational source code, say, in Fortran, and the resulting design is re-implemented in the target language, C in this case.
- Finally, a mix of compilation and decompilation is used to obtain the system implementation in C


Design
Reverse engineering
Forward engineering
Design
Source code
Existing
implementation
Re-code
Traget
implementation
Source code
Compilation
Decompilation
(Reverse engineering)
Object code
Object code
More
abstract
Less
abstract
1
2
2
3
3

TARGET SYSTEM TESTING

- In this phase for system testing, faults are detected in the target system.
- Those faults might have been introduced during reengineering.
- Fault detection is performed by applying the target system test plan on the target system.
- The same testing strategies, techniques, methods, and tools that are used in software development are used during reengineering.
- For example, apply the existing system-level test cases to both the existing and the new system.
- Assuming that the two systems have identical requirements, the test results from both the scenarios must be the same.

RE-DOCUMENTATION

- In the re-documentation phase, documentations are rewritten, updated, and/or replaced to reflect the target system.
- Documents are revised according to the re-documentation plan.
- The two major tasks within this phase are:
  (i) analyze new source code, and
  (ii) create documentation.
- Documents requiring revision are:
  - requirement specification.
  - design documentation.
  - a report justifying the design decisions, assumptions made in the implementation. configuration.
  - user and reference manuals.
  - on-line help.
  - document describing the differences between the existing and the target system.

ACCEPTANCE AND SYSTEM TRANSITION

- In this final phase, the reengineered system is evaluated by performing acceptance testing.
- Acceptance criteria should already have been established in the beginning of the project.
- Should the reengineered system pass those tests, preparation begins to transition to the new system.
- On the other hand, if the reengineered system fails some tests, the faults must be fixed; in some cases, those faults are fixed after the target system is deployed.
- Upon completion of the acceptance tests, the reengineered system is made operational, and the old system is put out of service.
- System transition is guided by the prior developed transition plan.

SOFTWARE RE-ENGINEERING

Unit 5

DATA RE-ENGINEERING

- So far, most of the discussion of software evolution has focused on the problems of changing programs and software systems.
- However, in many cases, there are associated problems of data evolution.
- The storage, organisation and format of the data processed by legacy programs may have to evolve to reflect changes to the software.
- The process of analysing and re-organising the data structures and, sometimes, the data values in a system to make it more understandable is called *data re-engineering*

- *Data degradation* Over time, the quality of data tends to decline.
  - errors, duplicate values,

- *Inherent limits that are built into the program:* When originally designed, developers of many programs included built-in constraints on the amount of data which could be processed. However, programs are now often required to process much more data than was originally envisaged by their developers.

- *Architectural evolution:* If a centralised system is migrated to a distributed architecture it is essential that the core of that architecture should be a data management system that can be accessed from remote clients. This may require a large data re-engineering effort to move data from separate files into the server database management system.

- Data re-engineering may be required to remove the limitations.
- 1- Data cleanup: The data records and values are analysed to improve their quality.

  -Duplicates are removed.

  -Redundant information is deleted.

  -Consistent format applied to all records.

  -This should not normally require any associated program changes.

- 2- Data extension:

  -Remove limits on the data processing.

  -This may require changes to programs to increase field lengths.

  -Modify upper limits on the tables, etc.

- 3-Data migration:

  -Data is moved into the control of a modern DBMS.

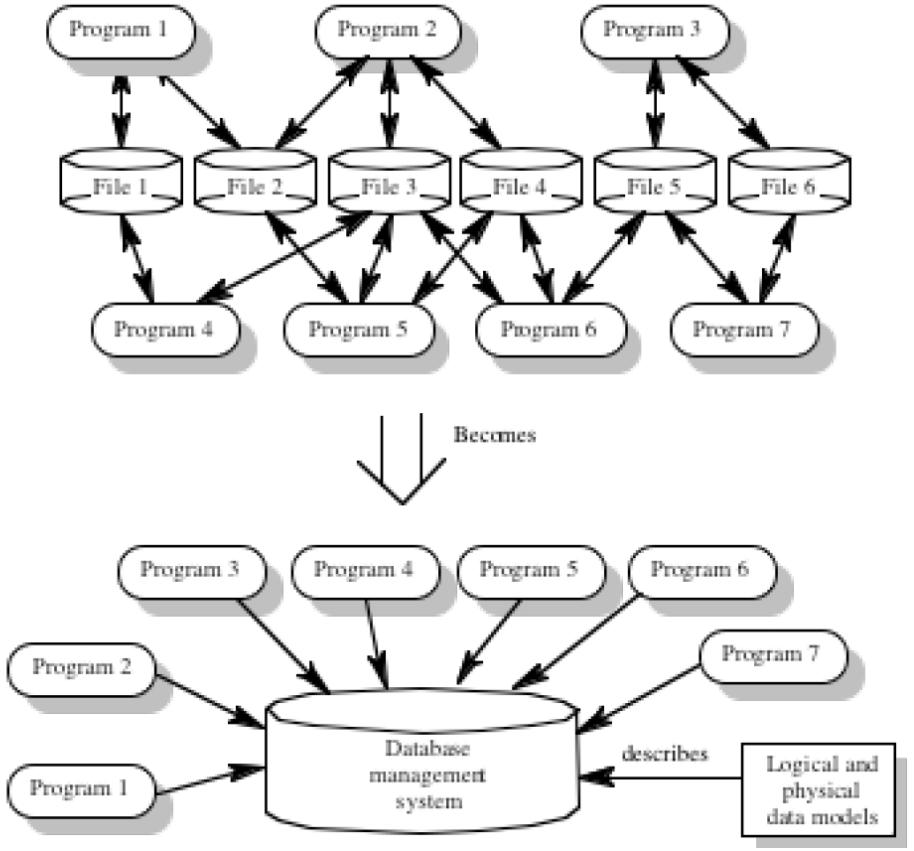
  -The data may be stored in separate files or may be managed by an older type of DBMS.

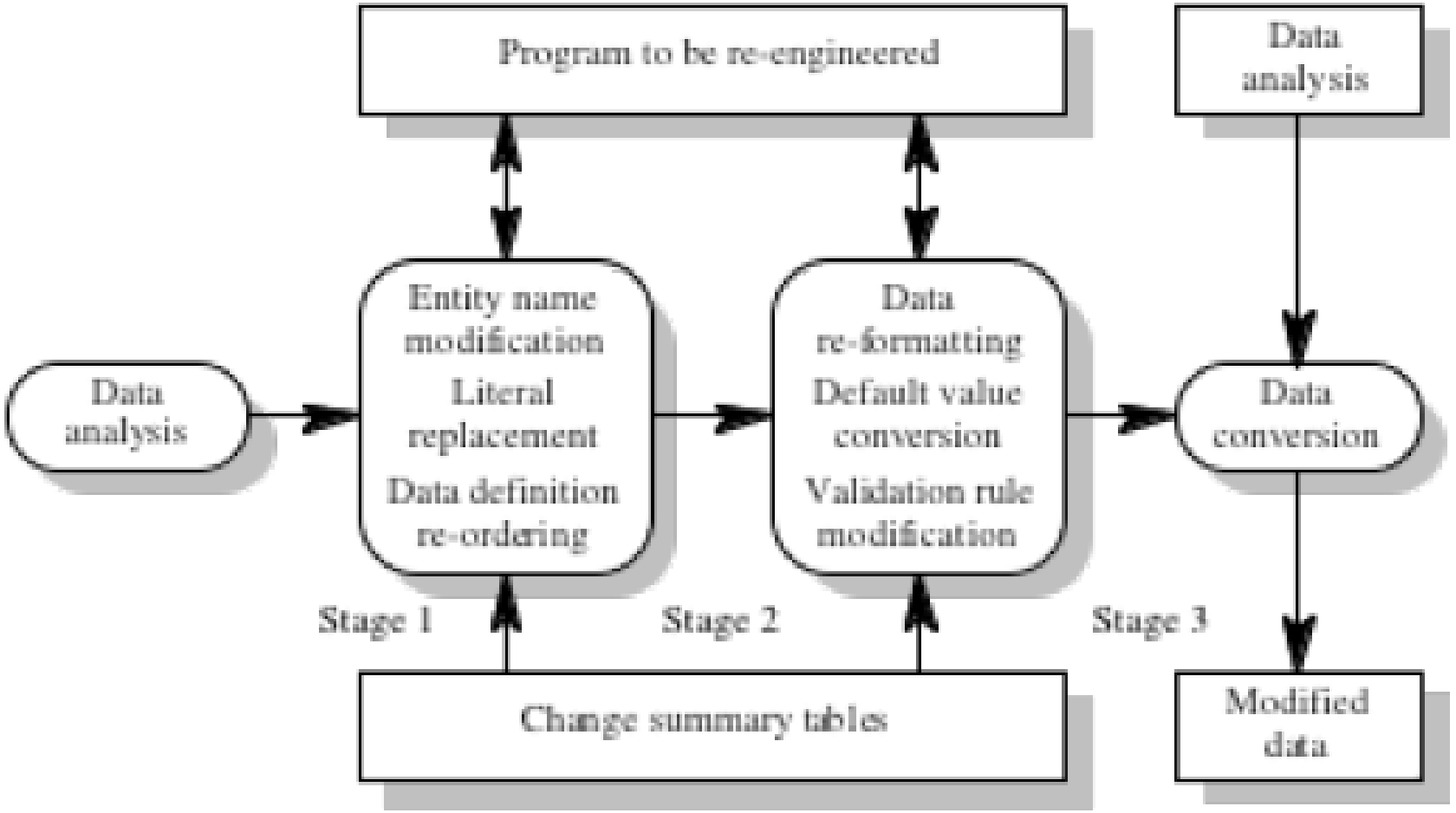
SOME OF THE PROBLEMS WITH DATA

- *Data naming problems* Names may be cryptic and difficult to understand.
  - The same name may be used in different programs to mean different things.
- This is a problem when field lengths in records are explicitly assigned in the program.
- Records representing the same entity may be organised differently in different programs.
- *Hard-coded literals* Literal (absolute) values.
- *No data dictionary*

SOME OF THE PROBLEMS WITH DATA

- Inconsistent data definitions, data values may also be stored in an inconsistent way.
- **Inconsistent Defaults**: *Different programs assign different default values to the same logical data items.*
- **Inconsistent Units.** *(lbs vs kg)*
- **Inconsistent Validation rules:** *Data written by one program may be rejected by another.*
- **Inconsistent representation semantics.**
- **Inconsistent handling of negative values**

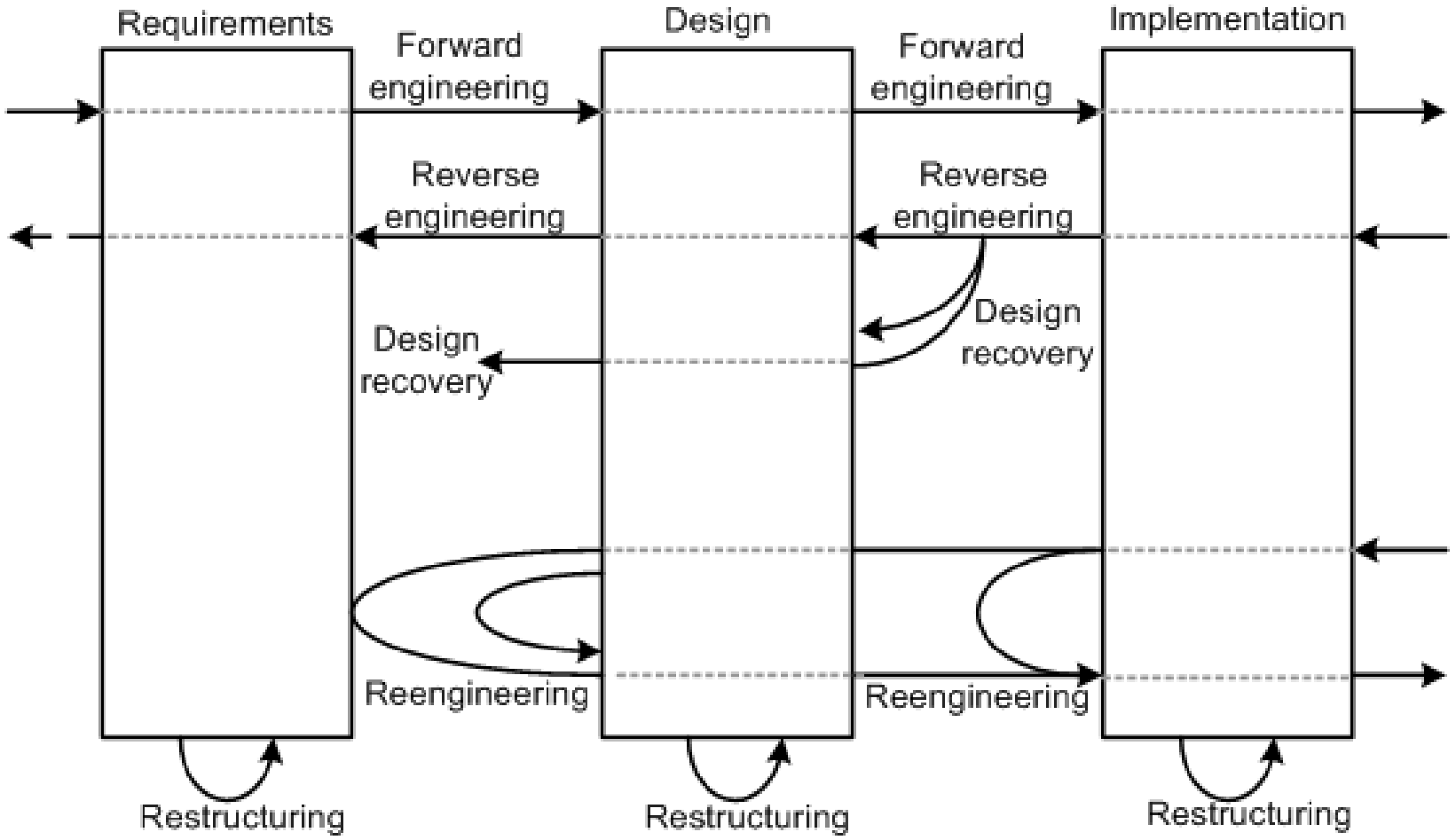



Program to be re-engineered
Data analysis
Entity name modification
Literal replacement
Data definition re-ordering
Data re-formatting
Default value conversion
Validation rule modification
Data analysis
Data conversion
Stage 1
Stage 2
Stage 3
Change summary tables
Modified data

SOFTWARE RE-ENGINEERING

Unit 6

THE RELATIONSHIP BETWEEN FORWARD ENGINEERING, REENGINEERING, AND REVERSE ENGINEERING



CODE REVERSE ENGINEERING

- Six objectives of reverse engineering, as identified by Chikofsky and Cross II:
  - generating alternative views.
  - recovering lost information.
  - synthesizing higher levels of abstractions.
  - detecting side effects.
  - facilitating reuse.
  - coping with complexity.

SIX KEY STEPS IN REVERSE ENGINEERING, THE IEEE STANDARD FOR SOFTWARE MAINTENANCE

Step 1: Partition source code into units.

Step 2: Describe the meanings of those units and identify the functional units.

Step 3: Create the input and output schematics of the units identified before.

Step 4: Describe the connected units.

Step 5: Describe the system application.

Step 6: Create an internal structure of the system.

PROBLEM AREAS

- redocumenting programs
- identifying reusable assets
- discovering design architectures,
- recovering design patterns
- building traceability between code and documentation
- finding objects in procedural programs
- deriving conceptual data models
- detecting duplications and clones
- cleaning up code smells
- aspect-oriented software development
- computing change impact
- transforming binary code into source code
- redesigning user interfaces
- parallelizing largely sequential programs
- translating a program to another language
- migrating data
- extracting business rules
- wrapping legacy code
- auditing security and vulnerability
- extracting protocols of network applications

GMT PARADIGM

- A high level organizational paradigm is found to be useful while setting up a reverse engineering process, as advocated by Benedusi et al.

- The high level paradigm plays two roles:
  - (i) define a framework to use the available methods and tools, and
  - (ii) allow the process to be repetitive.

- The paradigm, namely, Goals/Models/Tools, which partitions a process for reverse engineering into three ordered stages: Goals, Models, and Tools.

GOALS

- In this phase, the reasons for setting up a process for reverse engineering are identified and analyzed.
- Analyses are performed to identify the information needs of the process and the abstractions to be created by the process.
- The team setting up the process first acquires a good understanding of the forward engineering activities and the environment where the products of the reverse engineering process will be used.
- Results of the aforementioned comprehension are used to accurately identify:
  - (i) the information to be generated.
  - (ii) the formalisms to be used to represent the information.

MODELS

- In this phase, the abstractions identified in the Goals stage are analyzed to create representation models.
- Representation models include information required for the generation of abstractions.
- Activities in this phase are:
  - identify the kinds of documents to be generated.
  - to produce those documents, identify the information and their relations to be derived from source code.
  - define the models to be used to represent the information and their relationships extracted from source code.
  - to produce the desired documents from those models, define the abstraction algorithm for reverse engineering.
- The important properties of a reverse engineering model are: expressive power, language independence, compactness, richness of information content, granularity, and support for information preserving transformation.

TOOLS

- In this phase, tools needed for reverse engineering are identified, acquired, and/or developed in-house.

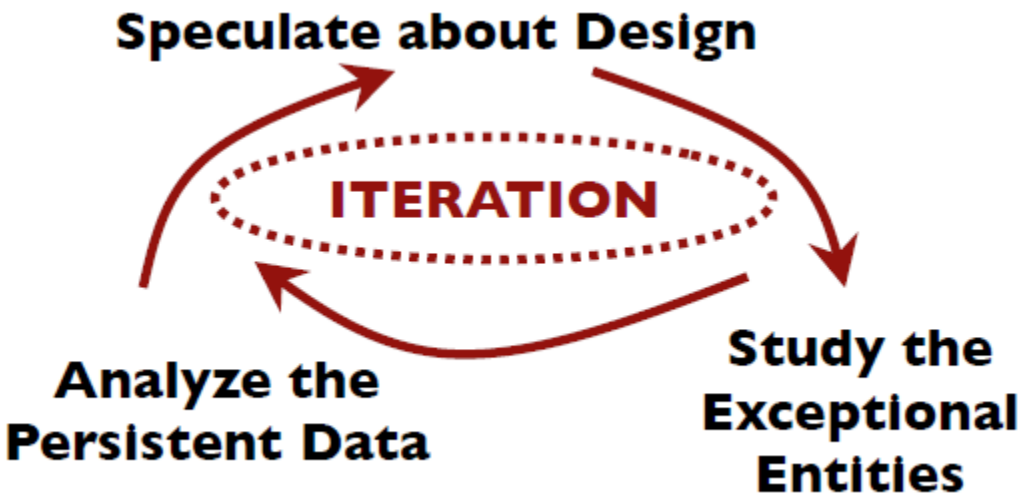
- Those tools are grouped into two categories:
  (i) tools to extract information and generate program representations according to the identified models.
  (ii) tools to extract information and produce the required documents. Extraction tools generally work on source code to reconstruct designdocuments.

- Therefore, those tools are ineffective in producing inputs for an abstraction process aiming to produce high-level design documents.

TECHNIQUES USED FOR REVERSE ENGINEERING

- The well-known analysis techniques that facilitate reverse engineering are:
    1. Lexical analysis.
    2. Control flow analysis.
    3. Data flow analysis.
    4. Program slicing.
    5. **Visualization.**
    6. **Program metrics.**

VISUALIZE DESIGN



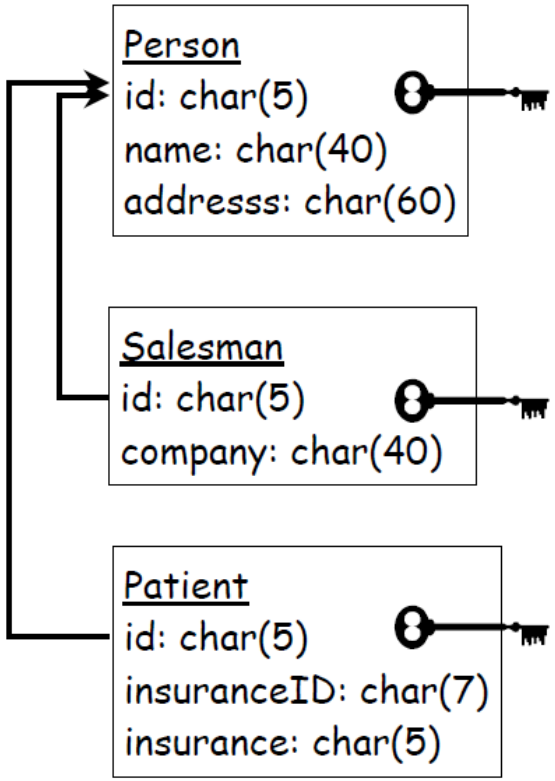
ANALYZE THE PERSISTENT DATA

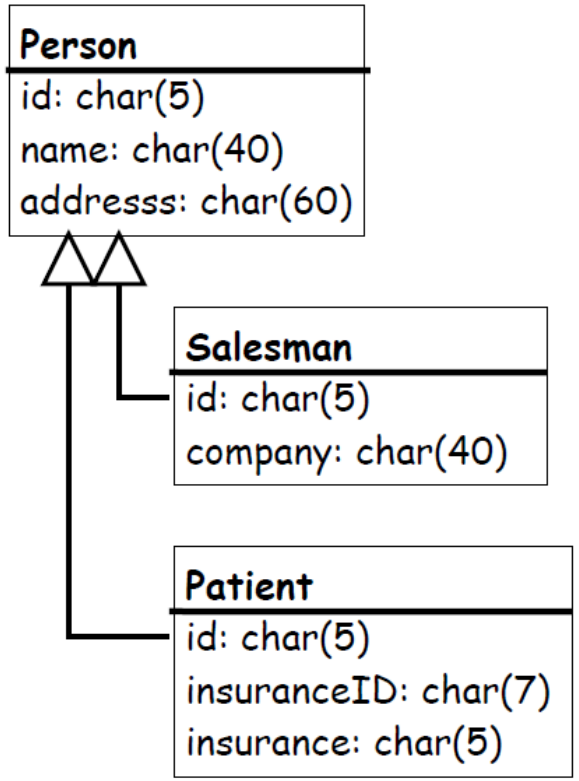
- Problem: Which objects represent valuable data?

Database	**UML**
Table	class
Columns	class attributes
Candidate keys	class associations
Naming conventions	Inheritance
Unique indices	
Foreign keys	

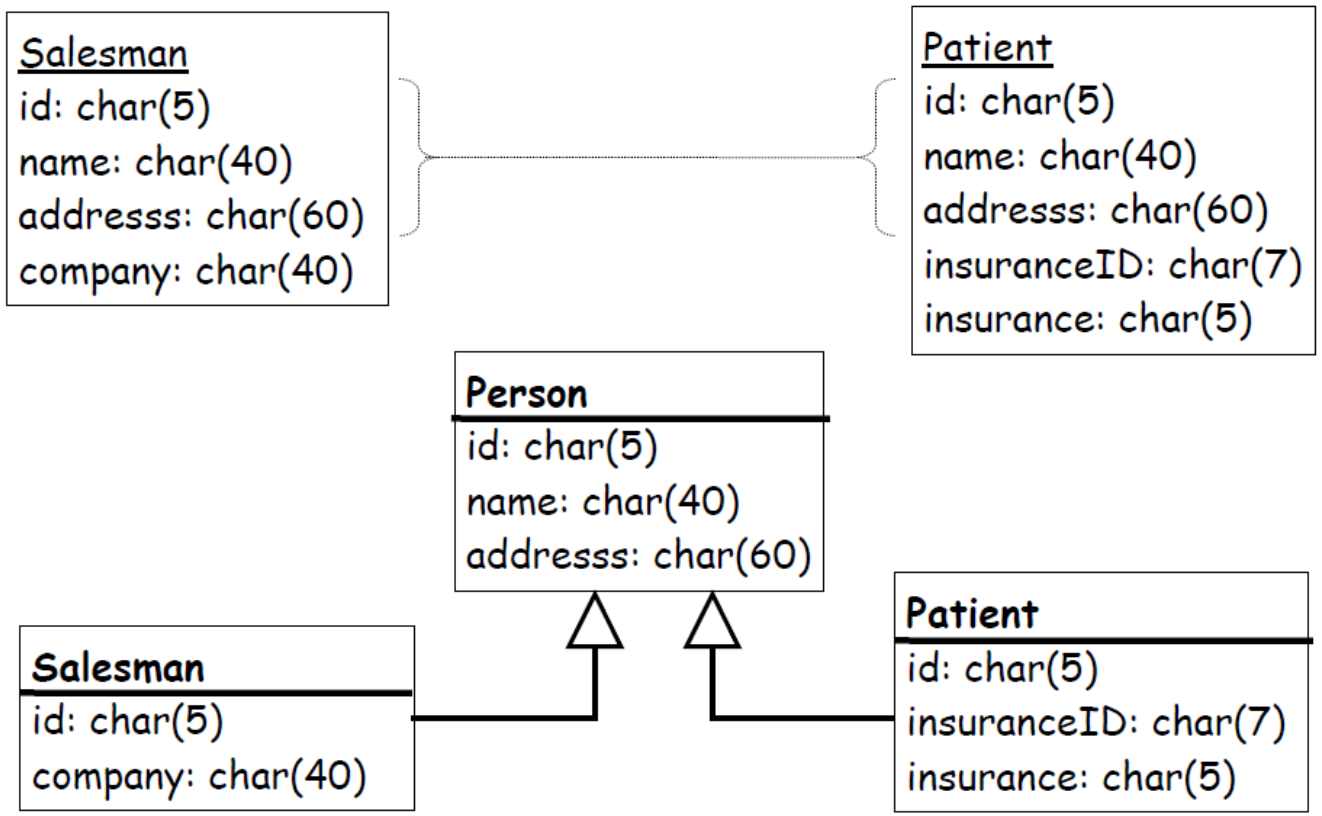
- Use explicit foreign key declarations
- Infer from column types + naming conventions + view declarations + join clauses

ONE TO ONE EXAMPLE





ROLL DOWN EXAMPLE



SPECULATE ABOUT DESIGN

- **Problem:** How do you recover the design from source code?

- **Solution:** Develop hypotheses and check them
  - Develop a plausible class diagram and iteratively check and refine your
  - design against the actual code

- **Variants:**
  - Speculate about Business Objects
  - Speculate about Design Patterns
  - Speculate about Architecture

STUDY THE EXCEPTIONAL ENTITIES

- **Problem:** How can you quickly identify design problems?
- **Solution:** Measure software entities and study the anomalous ones
- Visualize metrics to get an overview
  - Use simple metrics
  - Lines of code
  - Number of methods
  - Length of methods

STEP THROUGH THE EXECUTION

- **Problem:** How do you uncover the run-time architecture?
  - Collaborations are spread throughout the code
  - Polymorphism may hide which classes are instantiated

- **Solution:** Execute scenarios of known use cases and step through the code with a debugger
  - Set breakpoints
  - Change internal state to test alternative paths

LOOK FOR THE CONTRACTS

- **Problem:** What does a class expect from its clients?
  - Interfaces are visible in the code but how to use them?

- **Solution:** Look for common programming idioms
  - Look for "key methods"
    - Method name, parameter types (important type -> important method)
  - Constructor calls
    - Shows which parameters to pass
  - Template/hook methods
    - Shows how to specialize a sub-class

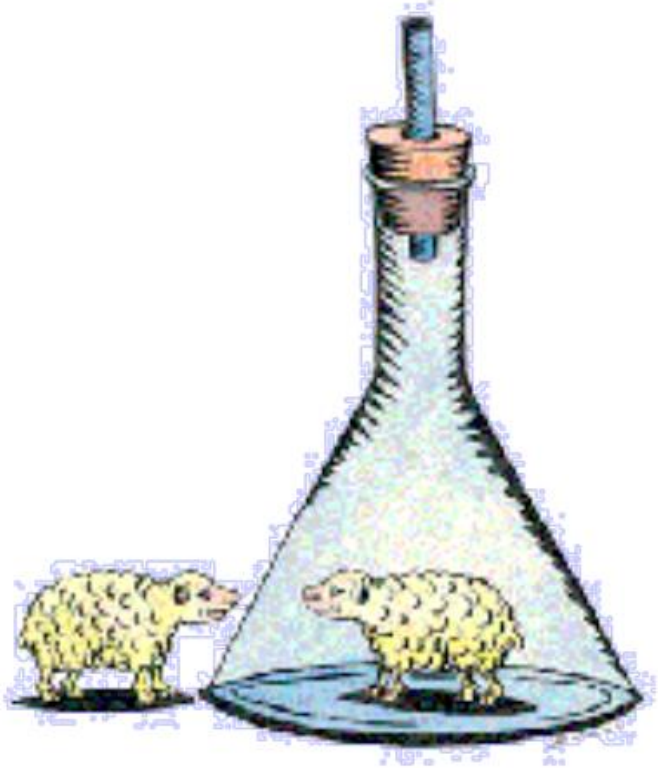
SOFTWARE RE-ENGINEERING

Unit 6

DESIGN PROBLEMS

Unclear & complicated

Duplicated (code clones)



CODE SMELLS

- A code smell is a hint that something has gone wrong somewhere in your code.
- If it stinks, change it
  - Duplicated Code
  - Long Method
  - Large Class
  - Long Parameter List
  - Divergent Change
  - Shotgun Surgery
  - Feature Envy

SMELL 1: LONG METHOD

- The longer a method is, the more difficult it is to understand it.
  - When is a method too long?
  - Heuristic: > 10 LOCs (?)

- How to detect?
  - Visualize LOC metric values of methods
  - “Method Length Distribution View”

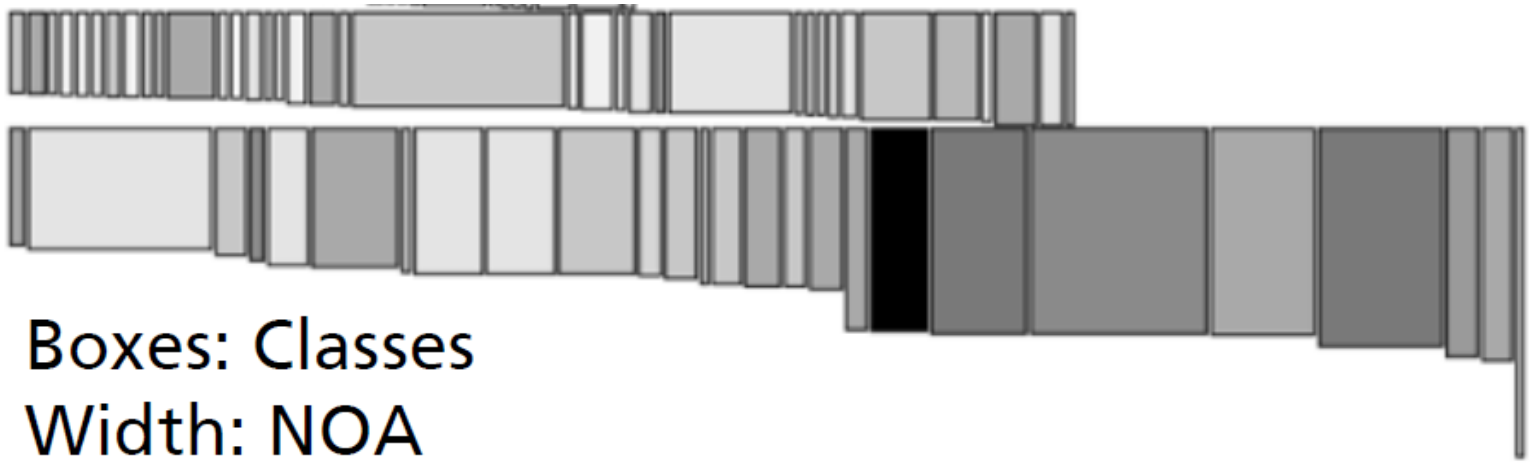
METHOD LENGTH DISTRIBUTION

SMELL 2: SWITCH STATEMENT

- Problem is similar to code duplication
  - Switch statement is scattered in different places
- How to detect?
- Visualize McCabe Cyclomatic Complexity metric to detect complex methods
  - “Method Complexity Distribution View”

SMELL 3: SYSTEM HOTSPOTS

- Classes that contain too much responsibilities
  - When is a class too large?
  - Heuristic: > 20 NOM
- How to detect?
  - Visualize number of methods (NOM) and sum of lines of code of methods (WLOC)
  - "System Hotspots View"

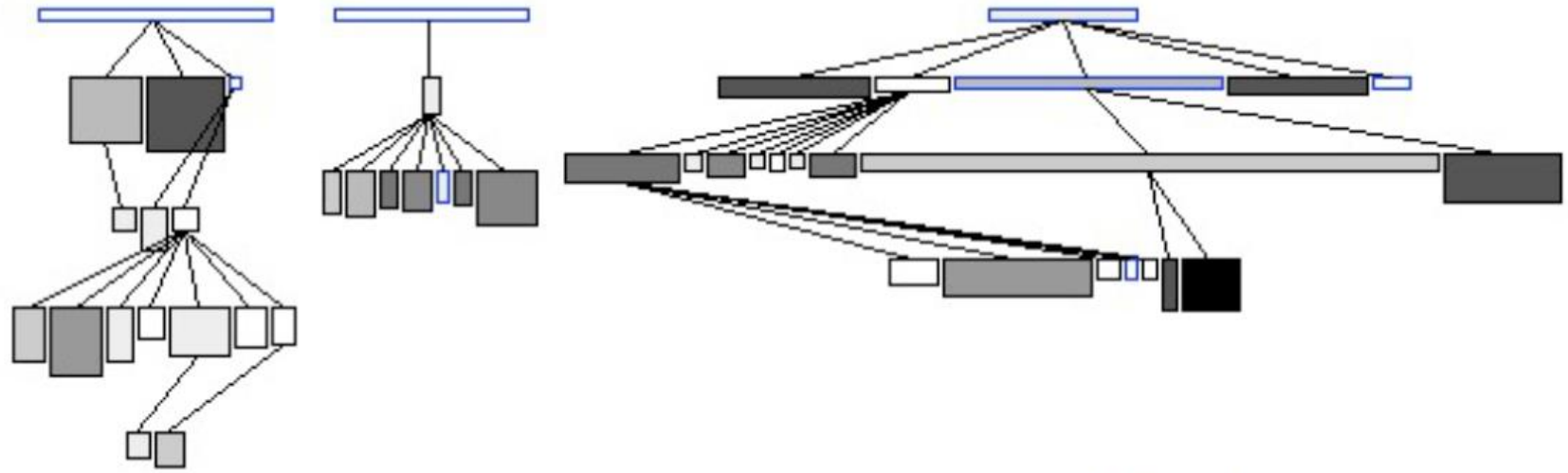


Boxes: Classes
Width: NOA
Height: NOM

SMELL 4: LAZY SUB-CLASS

- A class that is not doing enough to pay for itself should be eliminated
- How to detect?
  - Visualize inheritance structure with number of methods added (NMA),
  - overridden (NMO), and extended (NME)
  - “Inheritance Classification View”

INHERITANCE CLASSIFICATION



Metrics:
Boxes: Classes
Edges: Inheritance
Width: NMA
Height: NMO
Color: NME

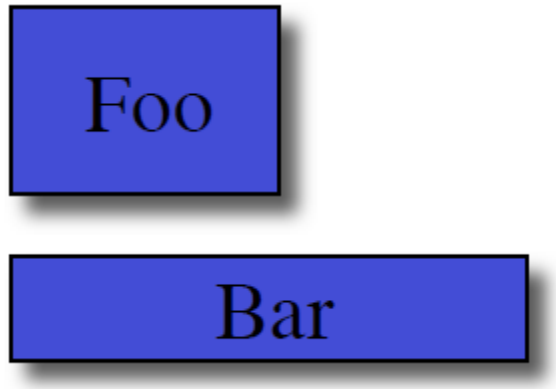
UNDERSTANDING EVOLUTION

- Changes can point to design problems
  - “Evolutionary Smells”
- But
  - Overwhelming complexity
  - How can we detect and understand changes?

- **Solutions**
  - The Evolution Matrix
  - The Kiviat Graphs

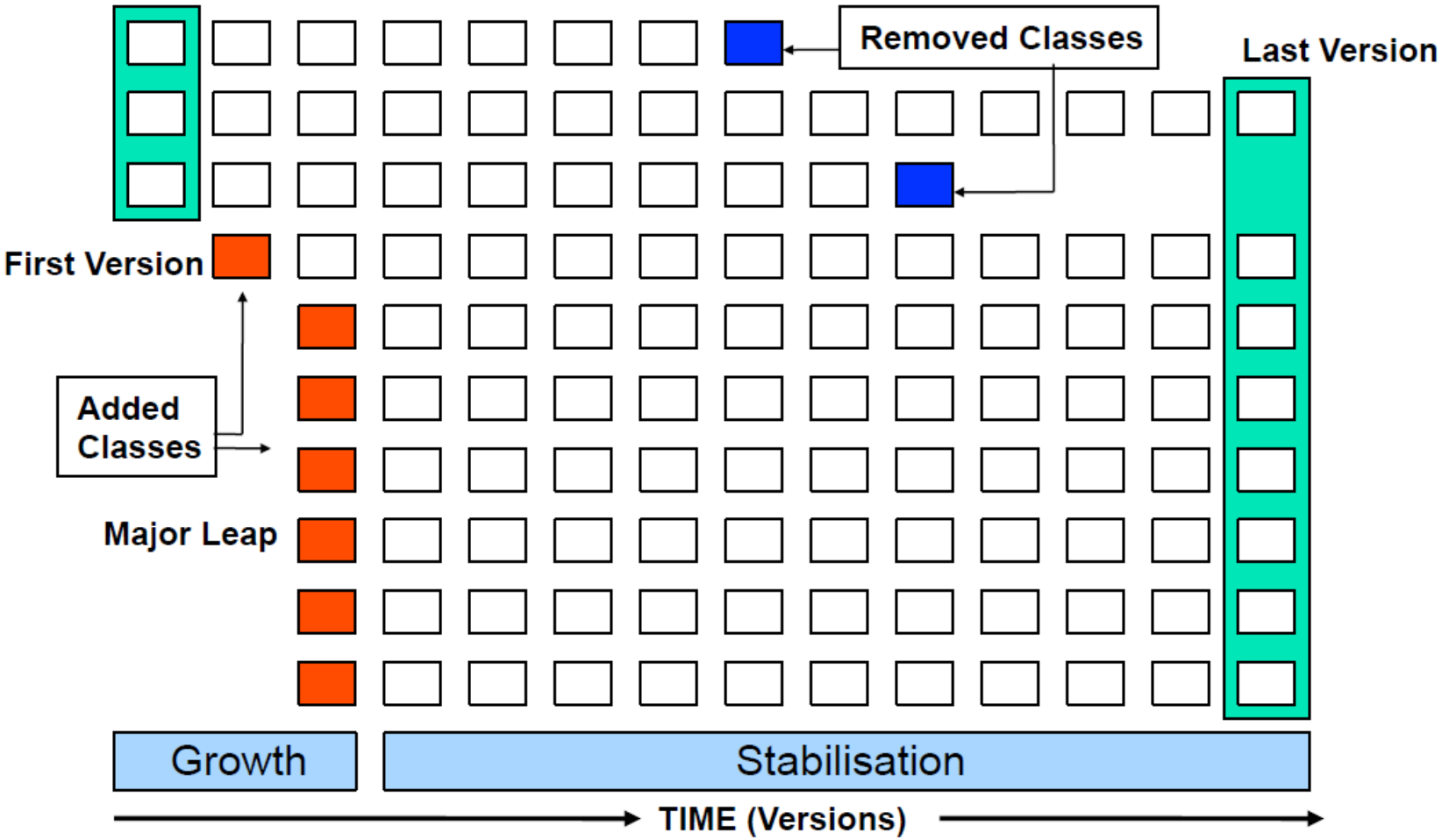
VISUALIZING CLASS EVOLUTION

- Visualize classes as rectangles using for
- width and height the following metrics:
  - NOM (number of methods)
  - NOA (number of attributes)



- The Classes can be categorized according to their “personal evolution” and to their “system evolution”
  - -> Evolution Patterns

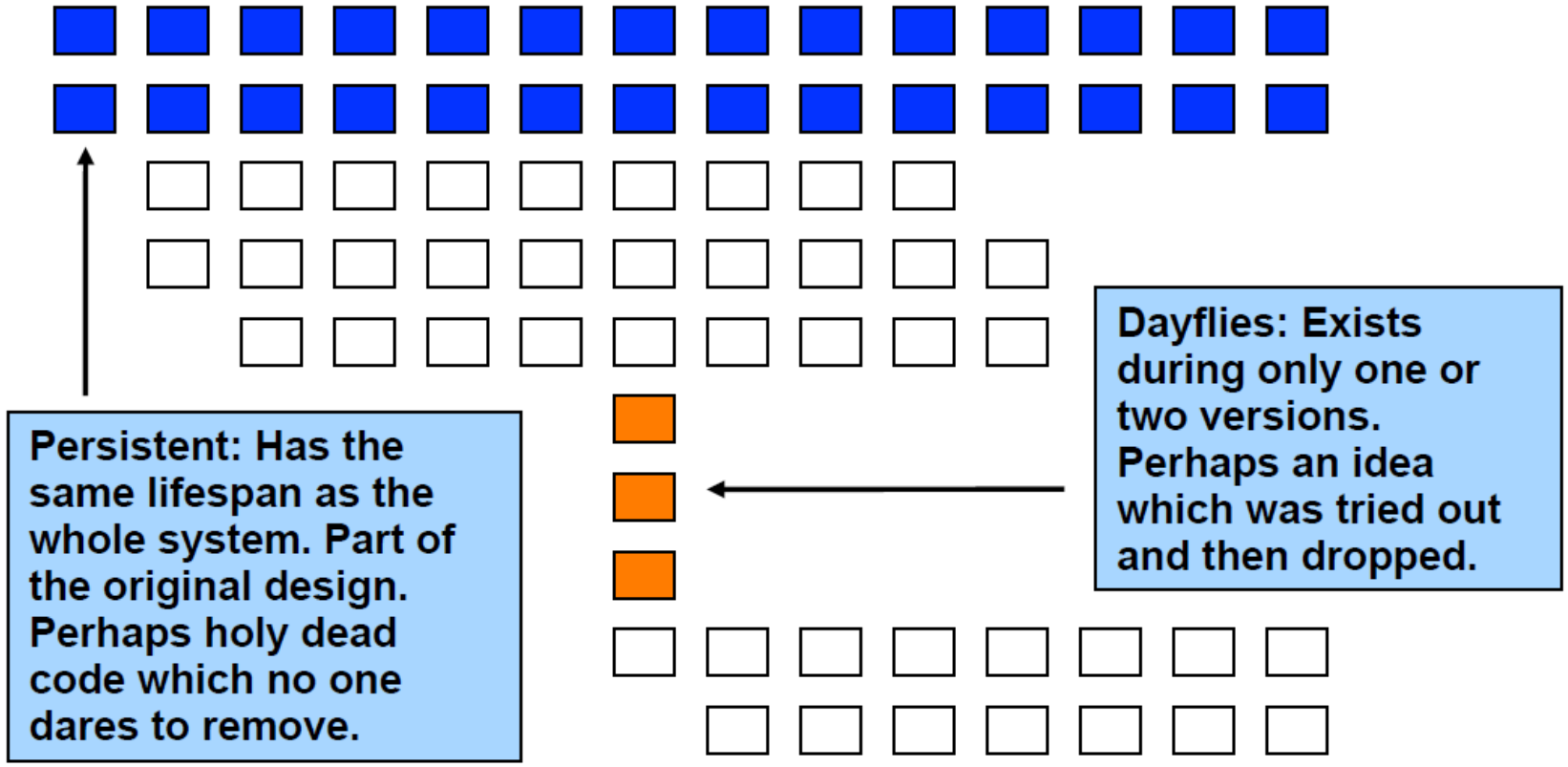
THE EVOLUTION MATRIX



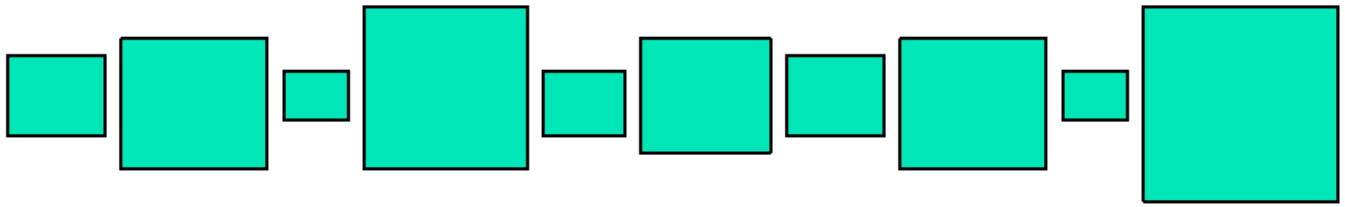
EVOLUTION PATTERNS & SMELLS

- Day-fly (Dead Code)
- Persistent
- Pulsar (Change Prone Entity)
- SupernovaWhite Dwarf (Dead Code)
- Red Giant (Large/God Class)
- Idle (Dead Code)

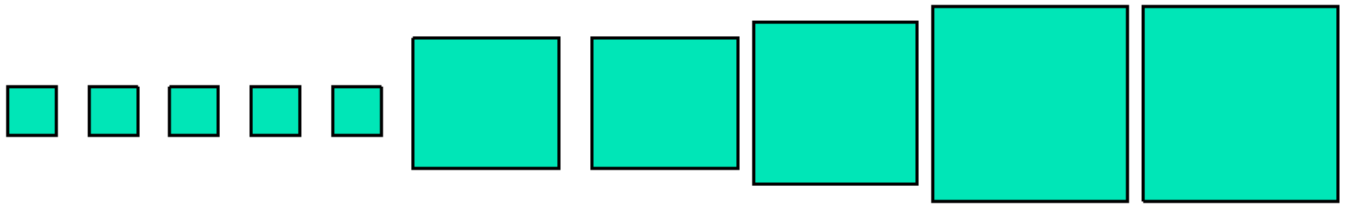
PERSISTENT / DAYFLY



PULSAR / SUPERNOVA

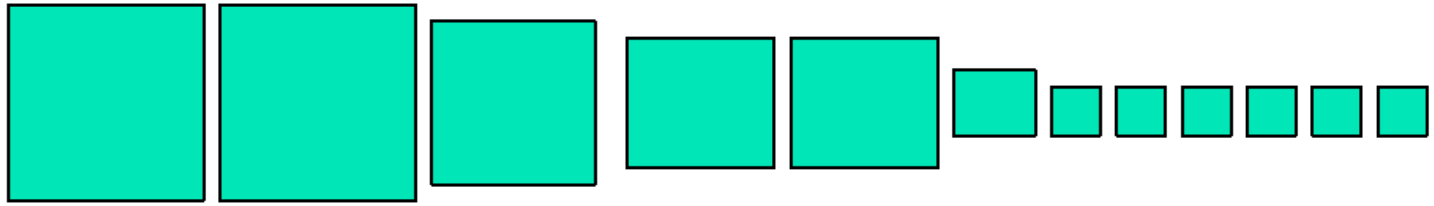


Pulsar: Repeated Modifications make it grow and shrink.
System Hotspot: Every System Version requires changes.

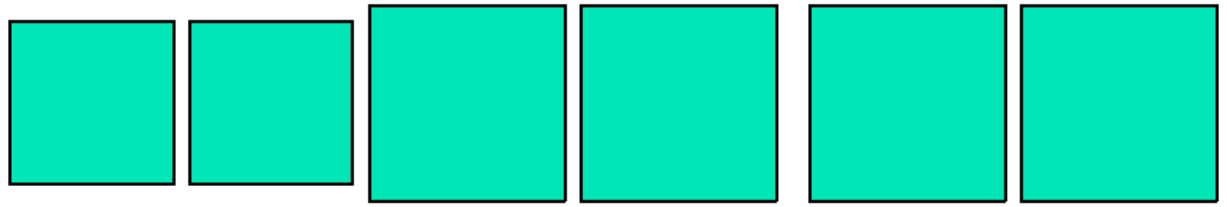


Supernova: Sudden increase in size. Possible Reasons:
- Massive shift of functionality towards a class.
- Data holder class for which it is easy to grow.
- *Sleeper*: Developers knew exactly what to fill in.

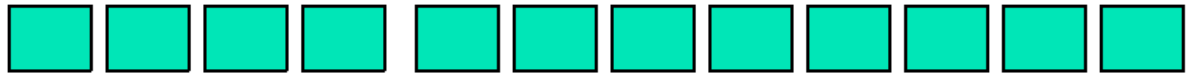
WHITE DWARF / RED GIANT / IDLE



White Dwarf: Lost the functionality it had and now trundles along without real meaning. Possibly dead code -> Lazy Class.



Red Giant: A permanent god (large) class which is always very large.



Idle: Keeps size over several versions. Possibly dead code, possibly good code.

EVALUATION: EVOLUTION MATRIX

- Pros
  - Understand the evolution of a system in terms of size and growth rate
  - Introduction of new classes
  - Remove of classes
  - Detection of Evolution Patterns & Smells
  - Dayflight, Persistent, White Dwarf, ...
- Cons
  - Scalability
  - Limited to 3 metric values per glyph
  - Fragile regarding the renaming of classes
    - What if the name of a class was changed?

EXTENDED POLYMETRIC VIEWS

- Goal:
  - Visualize n metric values of m releases
  - More semantic in graphs
  - More flexibility to combine metric values
- Solution:
  - Kiviat Diagrams (Radar Charts)
  - Each ray represents a metric
  - Encode releases with different colors